Introducción a los algoritmos, segunda edición

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein
La prensa del MIT
Cambridge, Massachusetts Londres, Inglaterra
Compañía de libros McGraw-Hill
Boston Burr Ridge, IL Dubuque, IA Madison, WI Nueva York San Francisco St. Louis
Montreal Toronto

Este libro es uno de una serie de textos escritos por la facultad de Ingeniería Eléctrica y
Departamento de Ciencias de la Computación del Instituto de Tecnología de Massachusetts. Fue editado y
producido por The MIT Press bajo un acuerdo conjunto de producción y distribución con el
Compañía de libros McGraw-Hill.

Información sobre pedidos:

Norteamérica

Los pedidos por mensaje de texto deben dirigirse a McGraw-Hill Book Company. Todos los demás pedidos deben
dirigirse a The MIT Press.

Fuera de América del Norte

Todos los pedidos deben dirigirse a The MIT Press o su distribuidor local.

Primera edición 1990

Este libro fue impreso y encuadernado en los Estados Unidos de América.

Datos de catalogación en publicación de la Biblioteca del Congreso

Introducción a los algoritmos / Thomas H. Cormen ... [et al.] .- 2ª ed.
pags. cm.
Incluye referencias bibliográficas e indice.
ISBN 0-262-03293-7 (hc .: papel alcalino, MIT Press) .- ISBN 0-07-013151-1 (McGraw-Hill)

1. Programación de computadoras. 2. Algoritmos informáticos. I. Título: Algoritmos. II. Cormen, Thomas
H.

QA76.6 I5858 2001

005.1-dc21

2001031277

Prefacio

Este libro ofrece una introducción completa al estudio moderno de la informática.
algoritmos. Presenta muchos algoritmos y los cubre en profundidad considerable, pero hace
su diseño y análisis accesibles a todos los niveles de lectores. Hemos tratado de mantener
explicaciones elementales sin sacrificar la profundidad de cobertura o el rigor matemático.

Cada capítulo presenta un algoritmo, una técnica de diseño, un área de aplicación o un tema relacionado.
Los algoritmos se describen en inglés y en un "pseudocódigo" diseñado para que cualquiera pueda leerlo.
que ha hecho un poco de programación. El libro contiene más de 230 figuras que ilustran cómo
los algoritmos funcionan. Dado que enfatizamos la *eficiencia* como criterio de diseño, incluimos una cuidadosa
análisis de los tiempos de ejecución de todos nuestros algoritmos.

El texto está destinado principalmente para su uso en cursos de pregrado o posgrado en algoritmos o
estructuras de datos. Porque analiza cuestiones de ingeniería en el diseño de algoritmos, así como
aspectos matemáticos, es igualmente adecuado para el autoaprendizaje por parte de profesionales técnicos.

En esta, la segunda edición, hemos actualizado todo el libro. Los cambios van desde el
adición de nuevos capítulos a la reescritura de frases individuales.

Al maestro

Este libro está diseñado para ser versátil y completo. Lo encontrará útil para una variedad
de cursos, desde un curso de pregrado en estructuras de datos hasta un curso de posgrado en
algoritmos. Porque hemos proporcionado considerablemente más material del que puede caber en un típico
curso de un trimestre, debe pensar en el libro como un "buffet" o "mezcla heterogénea" del que
puede elegir el material que mejor se adapte al curso que desea impartir.

Debería resultarle fácil organizar su curso en torno a los capítulos que necesita. Tenemos
hizo capítulos relativamente independientes, por lo que no necesita preocuparse por un inesperado y
dependencia innecesaria de un capítulo de otro. Cada capítulo presenta el material más fácil
primero y el material más difícil después, con límites de sección que marcan la parada natural
puntos. En un curso de pregrado, puede usar solo las secciones anteriores de un capítulo; en
un curso de posgrado, puede cubrir todo el capítulo.

Hemos incluido más de 920 ejercicios y más de 140 problemas. Cada sección termina con
ejercicios, y cada capítulo termina con problemas. Los ejercicios son generalmente preguntas breves.
que prueban el dominio básico del material. Algunos son simples ejercicios mentales de autoevaluación, mientras que
otros son más sustanciales y adecuados como tarea asignada. Los problemas son mas
estudios de casos elaborados que a menudo introducen material nuevo; normalmente constan de varios
preguntas que guían al alumno a través de los pasos necesarios para llegar a una solución.

Hemos destacado (⋆) las secciones y ejercicios que son más adecuados para estudiantes de posgrado
que para los estudiantes universitarios. Una sección destacada no es necesariamente más difícil que una no destacada

Página 3

uno, pero puede requerir una comprensión de matemáticas más avanzadas. Asimismo, destacó
Los ejercicios pueden requerir una formación avanzada o una creatividad superior a la media.

Para el estudiante

Esperamos que este libro de texto le proporcione una introducción agradable al campo de
algoritmos. Hemos intentado hacer que cada algoritmo sea accesible e interesante. Ayudar
cuando se encuentre con algoritmos desconocidos o difíciles, describimos cada uno en un paso a
paso a paso. También proporcionamos explicaciones detalladas de las matemáticas necesarias para comprender
el análisis de los algoritmos. Si ya está familiarizado con un tema, encontrará
los capítulos organizados para que pueda hojear las secciones introductorias y proceder rápidamente a la
material más avanzado.

Este es un libro grande y su clase probablemente cubrirá solo una parte de su material. Nosotros
Sin embargo, he intentado convertir este libro en un libro que le resultará útil ahora como libro de texto del curso.

y también más adelante en su carrera como referencia matemática de escritorio o manual de ingeniería.

¿Cuáles son los requisitos previos para leer este libro?

- Debe tener algo de experiencia en programación. En particular, debes entender procedimientos recursivos y estructuras de datos simples como matrices y listas enlazadas.
- Debería tener cierta facilidad con las demostraciones por inducción matemática. Algunas porciones del libro se basan en algunos conocimientos de cálculo elemental. Más allá de eso, las Partes I y VIII de este libro le enseñará todas las técnicas matemáticas que necesitará.

Al profesional

La amplia gama de temas de este libro lo convierte en un excelente manual sobre algoritmos. Porque cada capítulo es relativamente autónomo, puede concentrarse en los temas que más le interesan.

La mayoría de los algoritmos que discutimos tienen una gran utilidad práctica. Por lo tanto, abordamos preocupaciones de implementación y otros problemas de ingeniería. A menudo ofrecemos alternativas prácticas a los pocos algoritmos que son principalmente de interés teórico.

Si desea implementar alguno de los algoritmos, encontrará la traducción de nuestro pseudocódigo en su lenguaje de programación favorito, una tarea bastante sencilla. los El pseudocódigo está diseñado para presentar cada algoritmo de forma clara y sucinta. En consecuencia, hacemos no abordar el manejo de errores y otros problemas de ingeniería de software que requieren suposiciones sobre su entorno de programación. Intentamos presentar cada algoritmo simple y directamente sin permitir las idiosincrasias de un lenguaje de programación en particular para oscurecer su esencia.

A nuestros compañeros

Hemos proporcionado una extensa bibliografía y sugerencias para la literatura actual. Cada El capítulo termina con un conjunto de "notas del capítulo" que brindan detalles históricos y referencias. los Sin embargo, las notas de capítulo no proporcionan una referencia completa a todo el campo de los algoritmos. Aunque puede ser difícil de creer para un libro de este tamaño, muchos algoritmos interesantes podrían No se incluirá por falta de espacio.

Página 4

A pesar de las innumerables solicitudes de los estudiantes de soluciones a problemas y ejercicios, tenemos elegido como una cuestión de política para no proporcionar referencias para problemas y ejercicios, para eliminar la tentación de los estudiantes de buscar una solución en lugar de encontrarla ellos mismos.

Cambios para la segunda edición

¿Qué ha cambiado entre la primera y la segunda edición de este libro? Dependiendo de como míralo, no mucho o bastante.

Un vistazo rápido a la tabla de contenido muestra que la mayoría de los capítulos y secciones de la primera edición aparecen en la segunda edición. Eliminamos dos capítulos y un puñado de secciones, pero tenemos añadió tres nuevos capítulos y cuatro nuevas secciones además de estos nuevos capítulos. Si tuvieras que juzgar el alcance de los cambios por la tabla de contenido, probablemente concluiría que el los cambios fueron modestos.

Sin embargo, los cambios van mucho más allá de lo que se muestra en la tabla de contenido. En ningún particular orden, aquí hay un resumen de los cambios más significativos para la segunda edición:

- Cliff Stein fue agregado como coautor.
- Se han corregido los errores. Cuantos errores? Digamos solo varios.
- Hay tres nuevos capítulos:
  - El Capítulo 1 analiza el papel de los algoritmos en la informática.
  - El capítulo 5 cubre el análisis probabilístico y los algoritmos aleatorios. Como en el primera edición, estos temas aparecen en todo el libro.
  - El capítulo 29 está dedicado a la programación lineal.
- Dentro de los capítulos que se trasladaron de la primera edición, hay nuevas secciones sobre los siguientes temas:
  - hash perfecto (Sección 11.5),
  - dos aplicaciones de programación dinámica (Secciones 15.1 y 15.5), y
  - algoritmos de aproximación que utilizan aleatorización y programación lineal

(Sección 35.4).

- Para permitir que aparezcan más algoritmos al principio del libro, tres de los capítulos sobre los antecedentes matemáticos se han trasladado de la Parte I al Apéndice, que es la Parte VIII .
- Hay más de 40 problemas nuevos y más de 185 ejercicios nuevos.
- Hemos hecho explícito el uso de invariantes de bucle para demostrar la corrección. Nuestro primer El invariante de bucle aparece en el Capítulo 2 , y lo usamos un par de docenas de veces a través del libro.
- Muchos de los análisis probabilísticos se han reescrito. En particular, usamos en un docena de lugares la técnica de "indicadores de variables aleatorias", que simplifican análisis probabilísticos, especialmente cuando las variables aleatorias son dependientes.
- Hemos ampliado y actualizado las notas de los capítulos y la bibliografía. La bibliografía ha crecido más del 50% y hemos mencionado muchos resultados algorítmicos nuevos que aparecieron después de la impresión de la primera edición.

También hemos realizado los siguientes cambios:

- El capítulo sobre resolución de recurrencias ya no contiene el método de iteración. En cambio, en Sección 4.2 , hemos "promovido" árboles de recursividad para que constituyan un método en su propio Derecha. Hemos descubierto que extraer árboles de recursividad es menos propenso a errores que iterar

Página 5

recurrencias. Sin embargo, señalamos que los árboles de recursividad se utilizan mejor como una forma de generar conjeturas que luego se verifican mediante el método de sustitución.

- El método de partición utilizado para la ordenación rápida (Sección 7.1) y el tiempo lineal esperado El algoritmo estadístico de orden (Sección 9.2) es diferente. Ahora usamos el método desarrollado por Lomuto, que, junto con las variables aleatorias del indicador, permite una cierta análisis más sencillo. El método de la primera edición, debido a Hoare, aparece como un problema en el Capítulo 7 .
- Hemos modificado la discusión del hash universal en la Sección 11.3.3 para que se integra en la presentación de hash perfecto.
- Existe un análisis mucho más simple de la altura de un árbol de búsqueda binario construido aleatoriamente en Sección 12.4 .
- Las discusiones sobre los elementos de la programación dinámica (Sección 15.3) y la Los elementos de algoritmos codiciosos (Sección 16.2) se expanden significativamente. los exploración del problema de selección de actividad, que comienza con los algoritmos codiciosos capítulo, ayuda a aclarar la relación entre la programación dinámica y codiciosos algoritmos.
- Hemos reemplazado la prueba del tiempo de ejecución de la estructura de datos disjoint-set-union en la Sección 21.4 con una prueba que usa el método potencial para derivar un límite ajustado.
- La prueba de la corrección del algoritmo para componentes fuertemente conectados en La sección 22.5 es más simple, más clara y más directa.
- El Capítulo 24 , sobre las rutas más cortas de una sola fuente, se ha reorganizado para mover las pruebas de las propiedades esenciales a su propia sección. La nueva organización nos permite enfocarnos anteriormente en algoritmos.
- La sección 34.5 contiene una descripción general ampliada de NP-completitud, así como nuevas NP-pruebas de completitud para el ciclo hamiltoniano y los problemas de suma de subconjuntos.

Finalmente, prácticamente todas las secciones han sido editadas para corregir, simplificar y aclarar explicaciones. y pruebas.

Sitio web

Otro cambio con respecto a la primera edición es que este libro ahora tiene su propio sitio web: http://mitpress.mit.edu/algorithms/ . Puede utilizar el sitio web para informar errores, obtener una lista de errores conocidos o hacer sugerencias; nos gustaría saber de usted. Damos la bienvenida particularmente ideas para nuevos ejercicios y problemas, pero por favor incluya soluciones.

Lamentamos no poder responder personalmente a todos los comentarios.

Agradecimientos por la primera edición

Muchos amigos y colegas han contribuido enormemente a la calidad de este libro. Agradecemos a todos de ustedes por su ayuda y críticas constructivas.

El Laboratorio de Ciencias de la Computación del MIT ha proporcionado un entorno de trabajo ideal. Nuestra

colegas del Grupo de Teoría de la Computación del laboratorio han sido de
y tolerante con nuestras incesantes peticiones de valoración crítica de los capítulos. Agradecemos específicamente
Baruch Awerbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David
Shmoys y Éva Tardos. Gracias a William Ang, Sally Bemus, Ray Hirschfeld y Mark
Reinhold por mantener nuestras máquinas (DEC Microvaxes, Apple Macintosh y Sun

Página 6

Sparcstations) en ejecución y para recompilar cada vez que excedimos un límite de tiempo de compilación.
Thinking Machines Corporation brindó apoyo parcial para que Charles Leiserson trabajara en
este libro durante un permiso de ausencia del MIT.

Muchos colegas han utilizado borradores de este texto en cursos en otras escuelas. Han sugerido
numerosas correcciones y revisiones. En particular, queremos agradecer a Richard Beigel, Andrew
Goldberg, Joan Lucas, Mark Overmars, Alan Sherman y Diane Souvaine.

Muchos profesores asistentes en nuestros cursos han hecho contribuciones significativas a la
desarrollo de este material. Agradecemos especialmente a Alan Baratz, Bonnie Berger, Aditi Dhagat,
Burt Kaliski, Arthur Lent, Andrew Moulton, Marios Papaefthymiou, Cindy Phillips, Mark
Reinhold, Phil Rogaway, Flavio Rose, Arie Rudich, Alan Sherman, Cliff Stein, Susmita Sur,
Gregory Troxel y Margaret Tuttle.

Muchas personas proporcionaron una valiosa asistencia técnica adicional. Denise Sergent
Pasé muchas horas en las bibliotecas del MIT investigando referencias bibliográficas. Maria Sensale,
la bibliotecaria de nuestra sala de lectura, siempre fue alegre y servicial. Acceso a Albert Meyer's
La biblioteca personal ahorró muchas horas de tiempo en la biblioteca al preparar las notas del capítulo. Shlomo
Kipnis, Bill Niehaus y David Wilson corrigieron viejos ejercicios, desarrollaron nuevos y
escribió notas sobre sus soluciones. Marios Papaefthymiou y Gregory Troxel contribuyeron a la
indexación. A lo largo de los años, nuestras secretarias Inna Radzihovsky, Denise Sergent, Gayle Sherman,
y especialmente Be Blackburn brindó un apoyo inagotable en este proyecto, por lo que agradecemos
ellos.

Los estudiantes informaron de muchos errores en los primeros borradores. Agradecemos especialmente a Bobby
Blumofe, Bonnie Eisenberg, Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge,
John Pezaris, Steve Ponzio y Margaret Tuttle por sus cuidadosas lecturas.

Los colegas también han proporcionado revisiones críticas de capítulos específicos o información sobre
algoritmos, por los que estamos agradecidos. Agradecemos especialmente a Bill Aiello, Alok Aggarwal, Eric
Bach, Vašek Chvátal, Richard Cole, Johan Hastad, Alex Ishii, David Johnson, Joe Kilian,
Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plambeck, Hershel Safer, Jeff
Shallit, Cliff Stein, Gil Strang, Bob Tarjan y Paul Wang. Varios de nuestros colegas también
gentilmente nos proporcionó problemas; agradecemos particularmente a Andrew Goldberg, Danny
Sleator y Umesh Vazirani.

Ha sido un placer trabajar con The MIT Press y McGraw-Hill en el desarrollo de
este texto. Agradecemos especialmente a Frank Satlow, Terry Ehling, Larry Cohen y Lorrie Lejeune
de The MIT Press y David Shapiro de McGraw-Hill por su aliento, apoyo y
paciencia. Estamos particularmente agradecidos a Larry Cohen por su excelente corrección de estilo.

Agradecimientos por la segunda edición

Cuando le pedimos a Julie Sussman, PPA, que sirviera como correctora técnica para el segundo
edición, no sabíamos qué buen negocio estábamos obteniendo. Además de editar la
contenido técnico, Julie editó con entusiasmo nuestra prosa. Es humillante pensar en cuántos
errores que Julie encontró en nuestros borradores anteriores, aunque teniendo en cuenta cuántos errores encontró en el
primera edición (después de que se imprimiera, lamentablemente), no es de extrañar. Además, Julie sacrificó
su propio horario para adaptarse al nuestro, incluso trajo capítulos con ella en un viaje al
¡Islas Virgenes! Julie, no podemos agradecerte lo suficiente por el increíble trabajo que hiciste.

Página 7

El trabajo para la segunda edición se realizó mientras los autores eran miembros del Departamento de Ciencias de la Computación en Dartmouth College y el Laboratorio de Ciencias de la Computación en MIT. Ambos fueron entornos estimulantes en los que trabajar, y agradecemos a nuestros compañeros su apoyo.

Amigos y colegas de todo el mundo han brindado sugerencias y opiniones que guiaron nuestra escritura. Muchas gracias a Sanjeev Arora, Javed Aslam, Guy Blelloch, Avrim Blum, Scot Drysdale, Hany Farid, Hal Gabow, Andrew Goldberg, David Johnson, Yanlin Liu y Nicolas Schabanel, Alexander Schrijver, Sasha Shen, David Shmoys, Dan Spielman y Gerald Jay Sussman, Bob Tarjan, Mikkel Thorup y Vijay Vazirani.

Muchos profesores y colegas nos han enseñado mucho sobre algoritmos. Nosotros particularmente reconocer a nuestros maestros Jon L. Bentley, Bob Floyd, Don Knuth, Harold Kuhn, HT Kung, Richard Lipton, Arnold Ross, Larry Snyder, Michael I. Shamos, David Shmoys y Ken Steiglitz, Tom Szymanski, Éva Tardos, Bob Tarjan y Jeffrey Ullman.

Agradecemos el trabajo de los numerosos profesores asistentes de los cursos de algoritmos del MIT y Dartmouth, incluidos Joseph Adler, Craig Barrack, Bobby Blumofe, Roberto De Prisco, Matteo Frigo, Igal Galperin, David Gupta, Raj D. Iyer, Nabil Kahale, Sarfraz Khurshid, Stavros Kolliopoulos, Alain Leblanc, Yuan Ma, Maria Minkoff, Dimitris Mitsouras, Alin Popescu, Harald Prokop, Sudipta Sengupta, Donna Slonim, Joshua A. Tauber, Sivan Toledo, Elisheva Werner-Reiss, Lea Wittie, Qiang Wu y Michael Zhang.

William Ang, Scott Blomquist y Greg Shomo en MIT proporcionaron soporte informático y por Wayne Cripps, John Konkle y Tim Tregubov en Dartmouth. Gracias también a Be Blackburn, Don Dailey, Leigh Deacon, Irene Sebeda y Cheryl Patton Wu en el MIT y para Phyllis Bellmore, Kelly Clark, Delia Mauceli, Sammie Travis, Deb Whiting y Beth Young en Dartmouth para apoyo administrativo. Michael Fromberger, Brian Campbell y Amanda Eubanks, Sung Hoon Kim y Neha Narula también brindaron apoyo oportuno en Dartmouth.

Mucha gente tuvo la amabilidad de informar errores en la primera edición. Agradecemos a los siguientes personas, cada una de las cuales fue la primera en informar un error de la primera edición: Len Adleman, Selim Akl, Richard Anderson, Juan Andrade-Cetto, Gregory Bachelis, David Barrington, Paul Beame, Richard Beigel, Margrit Betke, Alex Blakemore, Bobby Blumofe, Alexander Brown, Xavier Cazin, Jack Chan, Richard Chang, Chienhua Chen, Ien Cheng, Hoon Choi, Drue Coles, Christian Collberg, George Collins, Eric Conrad, Peter Csaszar, Paul Dietz, Martin Dietzfelbinger, Scot Drysdale, Patricia Ealy, Yaakov Eisenberg, Michael Ernst, Michael Formann, Nedim Fresko, Hal Gabow, Marek Galecki, Igal Galperin, Luisa Gargano, John Gately, Rosario Genario, Mihaly Gereb, Ronald Greenberg, Jerry Grossman, Stephen Guattery, Alexander Hartemik, Anthony Hill, Thomas Hofmeister, Mathew Hostetter, Yih-Chun Hu, Dick Johnsonbaugh, Marcin Jurdzinki, Nabil Kahale, Fumiaki Kamiya y Anand Kanagala, Mark Kantrowitz, Scott Karlin, Dean Kelley, Sanjay Khanna, Haluk Konuk y Dina Kravets, Jon Kroger, Bradley Kuszmaul, Tim Lambert, Hang Lau, Thomas Lengauer, George Madrid, Bruce Maggs, Victor Miller, Joseph Muskat, Tung Nguyen, Michael Orlov, James Parque, Parque Seongbin, Ioannis Paschalidis, Boaz Patt-Shamir, Leonid Peshkin, Patricio Poblete, Ira Pohl, Stephen Ponzio, Kjell Post, Todd Poynor, Colin Prepscius, Sholom Rosen, Dale Russell, Hershel Safer, Karen Seidel, Joel Seiferas, Erik Seligman, Stanley Selkow, Jeffrey Shallit, Greg Shannon, Micha Sharir, Sasha Shen, Norman Shulman, Andrew Singer, Daniel Sleator, Bob Sloan, Michael Sofka, Volker Strumpen, Lon Sunshine, Julie Sussman, Asterio Tanaka, Clark Thomborson, Nils Thommesen, Homer Tilton, Martin Tompa, Andrei

Página 8

Toom, Felzer Torsten, Hirendu Vaishnav, M. Veldhorst, Luca Venuti, Jian Wang, Michael Wellman, Gerry Wiener, Ronald Williams, David Wolfe, Jeff Wong, Richard Woundy y Neal Young, Huaiyuan Yu, Tian Yuxing, Joe Zachary, Steve Zhang, Florian Zschoke y Uri Zwick.

Muchos de nuestros colegas proporcionaron revisiones detalladas o completaron una encuesta larga. Nosotros agradecemos revisores Nancy Amato, Jim Aspnes, Kevin Compton, William Evans, Peter Gacs, Michael Goldwasser, Andrzej Proskurowski, Vijaya Ramachandran y John Reif. También agradecemos al siguientes personas por enviar la encuesta: James Abello, Josh Benaloh, Bryan Beresford-Smith, Kenneth Blaha, Hans Bodlaender, Richard Borie, Ted Brown, Domenico Cantone, M. Chen, Robert Cimikowski, William Clocksin, Paul Cull, Rick Decker, Matthew Dickerson, Robert Douglas, Margaret Fleck, Michael Goodrich, Susanne Hambrusch, Dean Hendrix, Richard Johnsonbaugh, Kyriakos Kalorkoti, Srinivas Kankanahalli, Hikyoo Koh, Steven

Lindell, Errol Lloyd, Andy Lopez, Dian Rae Lopez, George Lucker, David Maier, Charles Martel, Xiannong Meng, David Mount, Alberto Policriti, Andrzej Proskurowski, Kirk Pruhs, Yves Robert, Guna Seetharaman, Stanley Selkow, Robert Sloan, Charles Steele, Gerard Tel, Murali Varanasi, Bernd Walter y Alden Wright. Ojalá pudiéramos haber realizado todo tus sugerencias. El único problema es que si lo hubiéramos tenido, la segunda edición habría sido aproximadamente 3000 páginas de largo!

La segunda edición se produjo en . Michael Downes convirtió las macros de "clásico" a , y convirtió los archivos de texto para usar estas nuevas macros. David Jones también proporcionado apoyo. Las cifras de la segunda edición fueron elaboradas por los autores. utilizando MacDraw Pro. Como en la primera edición, el índice se compiló usando Windex, un C programa escrito por los autores, y la bibliografía se preparó utilizando . Ayorkor Mills-Tettey y Rob Leathern ayudaron a convertir las figuras a MacDraw Pro, y Ayorkor también Consulte nuestra bibliografía.

Al igual que en la primera edición, trabajar con The MIT Press y McGraw-Hill ha sido un deleite. Nuestros editores, Bob Prior de The MIT Press y Betsy Jones de McGraw-Hill, publicaron con nuestras travesuras y nos mantuvo en marcha con zanahorias y palos.

Finalmente, agradecemos a nuestras esposas Nicole Cormen, Gail Rivest y Rebecca Ivry, nuestros hijos Ricky, William y Debby Leiserson; Alex y Christopher Rivest; y Molly, Noah y Benjamin Stein, y nuestros padres, Renee y Perry Cormen, Jean y Mark Leiserson, Shirley y Lloyd Rivest, e Irene e Ira Stein, por su amor y apoyo durante la redacción de este libro. La paciencia y el ánimo de nuestras familias hicieron posible este proyecto. Nosotros Les dedico con cariño este libro.

THOMAS H. CORMEN
Hannover, Nueva Hampshire

CHARLES E. LEISERSON
Cambridge, massachusetts

RONALD L. RIVEST
Cambridge, massachusetts

CLIFFORD STEIN
Hannover, Nueva Hampshire

Página 9

Mayo de 2001

Parte I: Fundamentos

Lista de capítulos

Capítulo 1: El papel de los algoritmos en la informática
Capítulo 2: Introducción
Capítulo 3: Crecimiento de funciones
Capítulo 4: Recurrencias
Capítulo 5: Análisis probabilístico y algoritmos aleatorios

Introducción

Esta parte lo ayudará a comenzar a pensar en diseñar y analizar algoritmos. Es pretende ser una introducción suave a cómo especificamos algoritmos, parte del diseño estrategias que usaremos a lo largo de este libro, y muchas de las ideas fundamentales utilizadas en análisis de algoritmos. Las partes posteriores de este libro se basarán en esta base.

El capítulo 1 es una descripción general de los algoritmos y su lugar en los sistemas informáticos modernos. Esta El capítulo define qué es un algoritmo y enumera algunos ejemplos. También argumenta que Los algoritmos son una tecnología, al igual que el hardware rápido, las interfaces gráficas de usuario, sistemas orientados y redes.

En el capítulo 2 , vemos nuestros primeros algoritmos, que resuelven el problema de ordenar una secuencia de n números. Están escritos en un pseudocódigo que, aunque no se puede traducir directamente a ningún

lenguaje de programación convencional, transmite la estructura del algoritmo con suficiente claridad que un programador competente pueda implementarlo en el lenguaje de su elección. La clasificación los algoritmos que examinamos son la ordenación por inserción, que utiliza un enfoque incremental, y la combinación sort, que utiliza una técnica recursiva conocida como "divide y vencerás". Aunque el tiempo cada uno requiere aumentos con el valor de n , la tasa de aumento difiere entre los dos algoritmos. Determinamos estos tiempos de ejecución en el Capítulo 2 y desarrollamos una notación útil para expresarlos.

El capítulo 3 define con precisión esta notación, que llamamos notación asintótica. Empieza por definir varias notaciones asintóticas, que usamos para delimitar los tiempos de ejecución del algoritmo desde arriba y / o abajo. El resto del Capítulo 3 es principalmente una presentación de matemáticas notación. Su propósito es más asegurar que el uso de la notación coincida con el de este libro que para enseñarte nuevos conceptos matemáticos.

El capítulo 4 profundiza en el método de divide y vencerás presentado en el capítulo 2 . En En particular, el Capítulo 4 contiene métodos para resolver recurrencias, que son útiles para describiendo los tiempos de ejecución de los algoritmos recursivos. Una técnica poderosa es el "maestro método ", que se puede utilizar para resolver las recurrencias que surgen de divide y vencerás algoritmos. Gran parte del capítulo 4 está dedicado a demostrar la exactitud del método maestro, aunque esta prueba puede omitirse sin daño.

Página 10

El capítulo 5 presenta el análisis probabilístico y los algoritmos aleatorios. Normalmente usamos análisis probabilístico para determinar el tiempo de ejecución de un algoritmo en los casos en los que, debido a la presencia de una distribución de probabilidad inherente, el tiempo de ejecución puede diferir en diferentes entradas del mismo tamaño. En algunos casos, asumimos que las entradas se ajustan a un conocido distribución de probabilidad, de modo que estamos promediando el tiempo de ejecución sobre todas las entradas posibles. En En otros casos, la distribución de probabilidad no proviene de las entradas sino de elecciones aleatorias. realizado durante el transcurso del algoritmo. Un algoritmo cuyo comportamiento está determinado no solo por su entrada sino por los valores producidos por un generador de números aleatorios es un algoritmo. Podemos usar algoritmos aleatorios para hacer cumplir una distribución de probabilidad en el insumos, asegurando así que ningún insumo en particular siempre cause un rendimiento deficiente, o incluso limitar la tasa de error de los algoritmos que pueden producir resultados incorrectos en un base.

Los apéndices AC contienen otro material matemático que le resultará útil a medida que lea este libro. Es probable que haya visto mucho del material de los capítulos del apéndice antes. haber leído este libro (aunque las convenciones de notación específicas que usamos pueden diferir en algunos casos de lo que ha visto en el pasado), por lo que debe pensar en los Apéndices como material de referencia. Por otro lado, probablemente no hayas visto la mayoría de los el material en la parte I . Todos los capítulos de la Parte I y los Apéndices están escritos con un tutorial sabor.

Capítulo 1: El papel de los algoritmos en Informática

¿Qué son los algoritmos? ¿Por qué vale la pena el estudio de algoritmos? ¿Cuál es el papel de algoritmos relativos a otras tecnologías utilizadas en computadoras? En este capítulo, responderemos estas preguntas.

1.1 Algoritmos

De manera informal, un *algoritmo* es cualquier procedimiento computacional bien definido que toma algún valor, o conjunto de valores, como *entrada* y produce algún valor, o conjunto de valores, como *salida* . Un algoritmo es por tanto, una secuencia de pasos computacionales que transforman la entrada en la salida.

También podemos ver un algoritmo como una herramienta para resolver un *problema computacional* bien especificado . El enunciado del problema especifica en términos generales la relación entrada / salida deseada. El algoritmo describe un procedimiento computacional específico para lograr esa entrada / salida relación.

Por ejemplo, es posible que deba ordenar una secuencia de números en orden no decreciente. Esta

El problema surge con frecuencia en la práctica y proporciona un terreno fértil para introducir muchos técnicas de diseño estándar y herramientas de análisis. Así es como definimos formalmente la clasificación problema :

- **Entrada:** una secuencia de n números $a_1, a_2, ..., a_n$.
- **Salida:** una permutación (reordenamiento) de la secuencia de entrada de modo que .

Página 11

Por ejemplo, dada la secuencia de entrada 31, 41, 59, 26, 41, 58, un algoritmo de clasificación devuelve como salida la secuencia 26, 31, 41, 41, 58, 59. Esta secuencia de entrada se denomina instancia del problema de clasificación. En general, una instancia de un problema consiste en la entrada (que satisface las restricciones impuestas en el planteamiento del problema) necesarias para calcular una solución a el problema.

La clasificación es una operación fundamental en la informática (muchos programas la utilizan como paso intermedio) y, como resultado, se ha desarrollado un gran número de buenos algoritmos de clasificación. desarrollado. Qué algoritmo es mejor para una aplicación determinada depende, entre otros factores, la cantidad de elementos que se ordenarán, la medida en que los elementos ya están algo ordenados, posibles restricciones en los valores de los elementos y el tipo de dispositivo de almacenamiento que se utilizará: principal memoria, discos o cintas.

Se dice que un algoritmo es correcto si, para cada instancia de entrada, se detiene con la salida correcta. Decimos que un algoritmo correcto resuelve el problema computacional dado. Una incorrecta Es posible que el algoritmo no se detenga en absoluto en algunas instancias de entrada, o puede que se detenga con una respuesta que el deseado. Al contrario de lo que cabría esperar, los algoritmos incorrectos a veces pueden ser útil, si se puede controlar su tasa de error. Veremos un ejemplo de esto en el capítulo 31. cuando estudiamos algoritmos para encontrar números primos grandes. Normalmente, sin embargo, estaremos preocupado únicamente por los algoritmos correctos.

Un algoritmo se puede especificar en inglés, como un programa de computadora o incluso como un hardware. diseño. El único requisito es que la especificación debe proporcionar una descripción precisa del procedimiento computacional a seguir.

¿Qué tipo de problemas se resuelven mediante algoritmos?

La clasificación no es de ninguna manera el único problema computacional para el que se han desarrollado. (Probablemente sospechaba tanto cuando vio el tamaño de este libro). Las aplicaciones de algoritmos son ubicuas e incluyen los siguientes ejemplos:

- El Proyecto Genoma Humano tiene el objetivo de identificar todos los 100.000 genes en ADN humano, determinando las secuencias de los 3 mil millones de pares de bases químicas que hacen hasta el ADN humano, almacenando esta información en bases de datos y desarrollando herramientas para datos análisis. Cada uno de estos pasos requiere algoritmos sofisticados. Mientras que las soluciones para Los diversos problemas involucrados están más allá del alcance de este libro, ideas de muchos Los capítulos de este libro se utilizan en la solución de estos problemas biológicos, por lo que permitiendo a los científicos realizar tareas utilizando los recursos de manera eficiente. Los ahorros están en el tiempo, tanto humanos como mecánicos, y en dinero, ya que se puede obtener más información extraído de técnicas de laboratorio.
- Internet permite a personas de todo el mundo acceder rápidamente y recuperar grandes cantidades de información. Para ello, se emplean algoritmos inteligentes para gestionar y manipular este gran volumen de datos. Ejemplos de problemas que deben resolverse incluyen encontrar buenas rutas por las que viajarán los datos (técnicas para resolver tales problemas aparecen en el Capítulo 24) y el uso de un motor de búsqueda para encontrar rápidamente páginas en qué información particular reside (las técnicas relacionadas se encuentran en los Capítulos 11 y 32).
- El comercio electrónico permite negociar e intercambiar bienes y servicios electrónicamente. La capacidad de mantener información como números de tarjetas de crédito, contraseñas, y los extractos bancarios privados son esenciales si el comercio electrónico se va a utilizar ampliamente.

Pagina 12

La criptografía de clave pública y las firmas digitales (tratadas en el Capítulo 31) se encuentran entre las tecnologías centrales utilizadas y se basan en algoritmos numéricos y teoría de números.

- En la fabricación y otros entornos comerciales, a menudo es importante asignar los escasos recursos de la manera más beneficiosa. Una empresa petrolera tal vez desee saber dónde colocar sus pozos para maximizar su beneficio esperado. Un candidato a la presidencia de los Estados Unidos puede querer determinar dónde gastar dinero comprando campañas publicidad para maximizar las posibilidades de ganar una elección. Una aerolínea puede desea asignar tripulaciones a los vuelos de la manera menos costosa posible, asegurándose de que cada vuelo está cubierto y que las regulaciones gubernamentales con respecto a la programación de la tripulación son reunió. Es posible que un proveedor de servicios de Internet desee determinar dónde colocar recursos para atender a sus clientes de manera más eficaz. Todos estos son ejemplos de problemas que pueden resolverse mediante programación lineal, que estudiaremos en Capítulo 29 .

Si bien algunos de los detalles de estos ejemplos están más allá del alcance de este libro, ofrecemos técnicas subyacentes que se aplican a estos problemas y áreas problemáticas. También mostramos cómo resolver muchos problemas concretos en este libro, incluidos los siguientes:

- Se nos da un mapa de carreteras en el que la distancia entre cada par de adyacentes intersecciones están marcadas, y nuestro objetivo es determinar la ruta más corta de una intersección a otra. La cantidad de rutas posibles puede ser enorme, incluso si no permitir rutas que se crucen sobre sí mismas. ¿Cómo elegimos cuál de todos los posibles rutas es la más corta? Aquí, modelamos la hoja de ruta (que es en sí misma un modelo de carreteras reales) como un gráfico (que veremos en el Capítulo 10 y el Apéndice B), y desea encontrar el camino más corto de un vértice a otro en el gráfico. Veremos cómo resolver este problema de manera eficiente en el Capítulo 24 .
- Se nos da una secuencia $A_1, A_2, ..., A_n$ de n matrices, y deseamos determinar su producto $A_1 A_2 A_n$. Como la multiplicación de matrices es asociativa, hay varios Órdenes legales de multiplicación. Por ejemplo, si $n = 4$, podríamos realizar la matriz multiplicaciones como si el producto estuviera entre paréntesis en cualquiera de los siguientes órdenes: $(A_1 (A_2 (A_3 A_4))), (A_1 ((A_2 A_3) A_4)), ((A_1 A_2) (A_3 A_4)), ((A_1 (A_2 A_3)) A_4)$, o $(((A_1 A_2$ todas estas matrices son cuadradas (y por lo tanto del mismo tamaño), el orden de multiplicación será no afecta el tiempo que tardan las multiplicaciones de matrices. Sin embargo, si estas matrices son de diferentes tamaños (pero sus tamaños son compatibles para la multiplicación de matrices), entonces el El orden de multiplicación puede marcar una gran diferencia. El número de posibles órdenes de multiplicación es exponencial en n , por lo que intentar todos los órdenes posibles puede tomar un mucho tiempo. Veremos en el capítulo 15 cómo utilizar una técnica general conocida como programación dinámica para resolver este problema de manera mucho más eficiente.
- Se nos da una ecuación $ax \equiv b \pmod n$, donde un , b y n son números enteros, y deseamos para encontrar todos los enteros x , módulo n , que satisfacen la ecuación. Puede haber cero, uno, o más de una de estas soluciones. Simplemente podemos intentar $x = 0, 1, ..., n - 1$ en orden, pero El capítulo 31 muestra un método más eficiente.
- Se nos dan n puntos en el plano y deseamos encontrar el casco convexo de estos puntos. El casco convexo es el polígono convexo más pequeño que contiene los puntos. Intuitivamente, podemos pensar en cada punto como representado por un clavo que sobresale de una tabla. El casco convexo estaría representado por una banda de goma apretada que rodea todas las uñas. Cada clavo alrededor del cual gira la goma elástica es un vértice del casco convexo. (Consulte la Figura 33.6 en la página 948 para ver un ejemplo.) Cualquiera de los 2^n subconjuntos de los puntos pueden ser los vértices del casco convexo. Saber qué puntos son vértices del casco convexo tampoco es suficiente, ya que también necesitamos saber el orden en

Página 13

que aparecen. Hay muchas opciones, por lo tanto, para los vértices del convexo cáscara. El capítulo 33 ofrece dos buenos métodos para encontrar el casco convexo.

Estas listas están lejos de ser exhaustivas (como probablemente haya deducido de nuevo a partir de la heft), pero exhiben dos características que son comunes a muchos algoritmos interesantes.

1. Hay muchas soluciones candidatas, la mayoría de las cuales no son las que queremos. Encontrar uno que queremos puede presentar un gran desafío.
2. Hay aplicaciones prácticas. De los problemas de la lista anterior, los caminos más cortos proporciona los ejemplos más fáciles. Una empresa de transporte, como camiones o ferrocarriles. empresa, tiene un interés financiero en encontrar los caminos más cortos a través de una carretera o ferrocarril

red porque tomar caminos más cortos resulta en menores costos de mano de obra y combustible. O una ruta
nodo en Internet puede necesitar encontrar la ruta más corta a través de la red para poder
enviar un mensaje rápidamente.

Estructuras de datos

Este libro también contiene varias estructuras de datos. Una *estructura de datos* es una forma de almacenar y organizar los datos para facilitar el acceso y modificaciones. Ninguna estructura de datos única funciona bien para todos los propósitos, por lo que es importante conocer las fortalezas y limitaciones de varios ellos.

Técnica

Aunque puede usar este libro como un "libro de cocina" para algoritmos, es posible que algún día encuentre un problema para el cual no puede encontrar fácilmente un algoritmo publicado (muchos de los ejercicios y problemas en este libro, por ejemplo!). Este libro le enseñará técnicas de diseño de algoritmos y análisis para que pueda desarrollar algoritmos por su cuenta, demuestre que brindan la responder y comprender su eficacia.

Problemas difíciles

La mayor parte de este libro trata sobre algoritmos eficientes. Nuestra medida habitual de eficiencia es la velocidad, es decir, cuánto tarda un algoritmo en producir su resultado. Sin embargo, existen algunos problemas para que no se conoce una solución eficaz. El capítulo 34 estudia un subconjunto interesante de estos problemas, que se conocen como NP-completo.

¿Por qué son interesantes los problemas NP-completos? Primero, aunque no hay un algoritmo eficiente para un NP-
Si alguna vez se ha encontrado un problema completo, nadie ha probado nunca que un algoritmo eficiente para uno no puede existir. En otras palabras, se desconoce si existen o no algoritmos eficientes para Problemas NP-completos. En segundo lugar, el conjunto de problemas NP-completos tiene la propiedad notable que si existe un algoritmo eficiente para cualquiera de ellos, entonces existen algoritmos eficientes para todos de ellos. Esta relación entre los problemas NP-completos hace que la falta de eficiencia soluciones aún más tentadoras. En tercer lugar, varios problemas NP-completos son similares, pero no idéntico, a problemas para los que conocemos algoritmos eficientes. Un pequeño cambio en el El planteamiento del problema puede provocar un gran cambio en la eficacia del algoritmo más conocido.

Es valioso conocer los problemas NP-completos porque algunos de ellos surgen sorprendentemente a menudo en aplicaciones reales. Si se le pide que produzca un algoritmo eficiente para un NP-problema completo, es probable que pase mucho tiempo en una búsqueda infructuosa. Si puedes mostrar

Página 14

que el problema es NP-completo, puede dedicar su tiempo a desarrollar una algoritmo que ofrece una buena, pero no la mejor solución posible.

Como ejemplo concreto, considere una empresa de camiones con un almacén central. Cada día, carga el camión en el almacén y lo envía a varios lugares para hacer entregas. Al final del día, el camión debe regresar al almacén para que esté listo para ser cargado para el día siguiente. Para reducir costos, la empresa desea seleccionar un pedido de paradas de entrega que producen la distancia total más baja recorrida por el camión. Este problema es el conocido "problema del viajante-vendedor" y es NP-completo. No tiene eficiencia conocida algoritmo. Sin embargo, bajo ciertos supuestos, existen algoritmos eficientes que dan una distancia total que no está muy por encima de la más pequeña posible. El capítulo 35 analiza tales "algoritmos de aproximación".

Ejercicios 1.1-1

Dé un ejemplo del mundo real en el que aparezca uno de los siguientes problemas de cálculo: ordenar, determinar el mejor orden para multiplicar matrices o encontrar el casco convexo.

Ejercicios 1.1-2

Aparte de la velocidad, ¿qué otras medidas de eficiencia se pueden utilizar en un entorno del mundo real?

Ejercicios 1.1-3

Seleccione una estructura de datos que haya visto anteriormente y analice sus fortalezas y limitaciones.

Ejercicios 1.1-4

¿En qué se parecen los problemas del camino más corto y del vendedor ambulante antes mencionados? Como son
son diferentes?

Ejercicios 1.1-5

Piense en un problema del mundo real en el que solo servirá la mejor solución. Entonces sube
con uno en el que una solución que sea "aproximadamente" la mejor es suficientemente buena.

1.2 Algoritmos como tecnología

Página 15

Supongamos que las computadoras fueran infinitamente rápidas y la memoria de la computadora estuviera libre. ¿Tendrías algo?
razón para estudiar algoritmos? La respuesta es sí, aunque no sea por otra razón que la que todavía
quisiera demostrar que su método de solución termina y lo hace con la respuesta correcta.

Si las computadoras fueran infinitamente rápidas, cualquier método correcto para resolver un problema sería suficiente. Tú
probablemente querría que su implementación estuviera dentro de los límites de un buen software
práctica de ingeniería (es decir, bien diseñada y documentada), pero usaría con mayor frecuencia
Cualquiera que sea el método más fácil de implementar.

Por supuesto, las computadoras pueden ser rápidas, pero no infinitamente rápidas. Y la memoria puede ser barata
pero no es gratis. Por tanto, calcular el tiempo es un recurso limitado, al igual que el espacio en la memoria.
Estos recursos deben usarse con prudencia y los algoritmos que sean eficientes en términos de tiempo o
el espacio te ayudará a hacerlo.

Eficiencia

Los algoritmos diseñados para resolver el mismo problema a menudo difieren dramáticamente en su eficiencia.
Estas diferencias pueden ser mucho más significativas que las diferencias debidas al hardware y
software.

Como ejemplo, en el Capítulo 2 , veremos dos algoritmos para ordenar. El primero, conocido como
ordenación por inserción , lleva un tiempo aproximadamente igual ac_1n_2 para ordenar n elementos, donde c_1 es una constante que no
no depende de n . Es decir, lleva un tiempo aproximadamente proporcional an_2 . El segundo, tipo de fusión ,
toma un tiempo aproximadamente igual $ac_2n \lg n$, donde lg n representa $\log_2 n$ y c_2 es otra constante
que tampoco depende de n . La ordenación por inserción generalmente tiene un factor constante más pequeño que la combinación
ordenar, de modo que $c_1 < c_2$. Veremos que los factores constantes pueden ser mucho menos significativos en el
tiempo de ejecución que la dependencia del tamaño de entrada n . Donde la ordenación por fusión tiene un factor de lg n en
su tiempo de ejecución, la ordenación por inserción tiene un factor de n , que es mucho mayor. Aunque el tipo de inserción
suele ser más rápido que el ordenamiento combinado para tamaños de entrada pequeños, una vez que el tamaño de entrada n *se* vuelve grande
suficiente, la ventaja del tipo de fusión de lg n frente a n compensará con creces la diferencia en
factores constantes. No importa cuánto más pequeño sea c_1 que c_2 , siempre habrá un cruce
punto más allá del cual la ordenación por combinación es más rápida.

Para un ejemplo concreto, enfrentemos una computadora más rápida (computadora A) que ejecuta la ordenación por inserción
contra una computadora más lenta (computadora B) que ejecuta la clasificación por combinación. Cada uno debe ordenar una serie de
un millón de números. Suponga que la computadora A ejecuta mil millones de instrucciones por segundo
y la computadora B ejecuta solo diez millones de instrucciones por segundo, por lo que la computadora A es 100
veces más rápido que el ordenador B en potencia de cálculo bruta. Para marcar la diferencia aún más
dramático, supongamos que el programador más astuto del mundo ordena la inserción de códigos en la máquina
idioma para la computadora A, y el código resultante requiere $2n_2$ instrucciones para ordenar n números.
(Aquí, $c_1 = 2$.) La ordenación por fusión, por otro lado, está programada para la computadora B por un promedio

programador usando un lenguaje de alto nivel con un compilador ineficiente, con el código resultante tomando 50 n lg n instrucciones (de modo que $c_2 = 50$). Para ordenar un millón de números, la computadora A toma

mientras que la computadora B toma

Página 16

Al utilizar un algoritmo cuyo tiempo de ejecución crece más lentamente, incluso con un compilador deficiente, la computadora B corre 20 veces más rápido que la computadora A! La ventaja de la ordenación combinada es aún mayor pronunciado cuando ordenamos diez millones de números: donde la ordenación por inserción toma aproximadamente 2,3 días, la ordenación combinada tarda menos de 20 minutos. En general, a medida que aumenta el tamaño del problema, también lo hace ventaja relativa de la ordenación por fusión.

Algoritmos y otras tecnologías

El ejemplo anterior muestra que los algoritmos, como el hardware informático, son una tecnología . Total El rendimiento del sistema depende tanto de la elección de algoritmos eficientes como de la elección rápida. hardware. Así como se están logrando rápidos avances en otras tecnologías informáticas, también se hace en algoritmos.

Podría preguntarse si los algoritmos son realmente tan importantes en las computadoras contemporáneas en a la luz de otras tecnologías avanzadas, como

- hardware con altas velocidades de reloj, canalización y arquitecturas superescalares,
- interfaces gráficas de usuario (GUI) intuitivas y fáciles de usar,
- sistemas orientados a objetos, y
- Redes de área local y de área extensa.

La respuesta es sí. Aunque hay algunas aplicaciones que no requieren explícitamente contenido algorítmico a nivel de aplicación (por ejemplo, algunas aplicaciones simples basadas en web), la mayoría también requieren un grado de contenido algorítmico por sí mismos. Por ejemplo, considere un sitio web servicio que determina cómo viajar de un lugar a otro. (Varios de estos servicios existía en el momento de escribir este artículo.) Su implementación se basaría en hardware rápido, un interfaz gráfica de usuario, redes de área amplia y, posiblemente, también sobre orientación a objetos. Sin embargo, también requeriría algoritmos para ciertas operaciones, como encontrar rutas (probablemente usando un algoritmo de ruta más corta), renderizando mapas e interpolando direcciones.

Además, incluso una aplicación que no requiera contenido algorítmico en la aplicación El nivel se basa en gran medida en algoritmos. ¿La aplicación depende de hardware rápido? los El diseño de hardware utilizó algoritmos. ¿La aplicación se basa en interfaces gráficas de usuario? los El diseño de cualquier GUI se basa en algoritmos. ¿La aplicación depende de la red? Enrutamiento en las redes se basan en gran medida en algoritmos. ¿La solicitud estaba escrita en un idioma diferente al ¿codigo de maquina? Luego fue procesado por un compilador, intérprete o ensamblador, todos los cuales hacer un uso extensivo de algoritmos. Los algoritmos son el núcleo de la mayoría de las tecnologías utilizadas en computadoras contemporáneas.

Además, con las capacidades cada vez mayores de las computadoras, las usamos para resolver problemas que nunca. Como vimos en la comparación anterior entre ordenación por inserción y fusionar ordenación, es en problemas de mayor tamaño que las diferencias en la eficiencia entre algoritmos se vuelven particularmente prominentes.

Tener una base sólida de conocimiento y técnica algorítmica es una característica que separa a los programadores verdaderamente hábiles de los novatos. Con la informática moderna

Página 17

tecnología, puede realizar algunas tareas sin saber mucho sobre algoritmos, pero con una buena experiencia en algoritmos, puede hacer mucho, mucho más.

Ejercicios 1.2-1

Dé un ejemplo de una aplicación que requiera contenido algorítmico a nivel de aplicación, y discutir la función de los algoritmos involucrados.

Ejercicios 1.2-2

Suponga que estamos comparando implementaciones de ordenación por inserción y ordenación por fusión en el mismo máquina. Para entradas de tamaño n , la ordenación por inserción se ejecuta en $8n^2$ pasos, mientras que la ordenación por combinación se ejecuta en $64n \lg n$ pasos. ¿Para qué valores de n *la* ordenación por inserción con la combinación por combinación?

Ejercicios 1.2-3

¿Cuál es el valor más pequeño de n tal que un algoritmo cuyo tiempo de ejecución es $100n^2$ se ejecuta más rápido? que un algoritmo cuyo tiempo de ejecución es 2^n en la misma máquina?

Problemas 1-1: Comparación de tiempos de ejecución

Para cada función $f(n)$ y tiempo t en la siguiente tabla, determine el tamaño más grande n de un problema que se puede resolver en el tiempo t , asumiendo que el algoritmo para resolver el problema requiere $f(n)$ microsegundos.

	1 segundo	1 minuto	1 hora	1 día	1 mes	1 año	1 siglo
$\lg n$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Página 18

Notas del capítulo

Hay muchos textos excelentes sobre el tema general de los algoritmos, incluidos los de Aho, Hopcroft y Ullman [5 , 6] , Baase y Van Gelder [26] , Brassard y Bratley [46 , 47] , Goodrich y Tamassia [128] , Horowitz, Sahni y Rajasekaran [158] , Kingston [179] , Knuth [182 , 183 , 185] , Kozen [193] , Manber [210] , Mehlhorn [217 , 218 , 219] , Purdom y Brown [252] , Reingold, Nievergelt y Deo [257] , Sedgewick [269] , Skiena [280] y Wilf [315] . Bentley analiza algunos de los aspectos más prácticos del diseño de algoritmos [39 , 40] y Gonnet [126] . Las encuestas del campo de los algoritmos también se pueden encontrar en el Manual

of Theoretical Computer Science, Volumen A [302] y el Manual CRC sobre Algoritmos y Teoría de la Computación [24] . Descripción general de los algoritmos utilizados en biología computacional puede encontrarse en libros de texto de Gusfield [136] , Pevzner [240] , Setubal y Medinas [272] , y Waterman [309] .

Capítulo 2: Introducción

Este capítulo lo familiarizará con el marco que usaremos a lo largo del libro para Piense en el diseño y análisis de algoritmos. Es autónomo, pero incluye Varias referencias al material que se presentará en los Capítulos 3 y 4 . (También contiene varias sumas, que el Apéndice A muestra cómo resolver).

Comenzamos examinando el algoritmo de ordenación por inserción para resolver el problema de ordenación introducido en Capítulo 1 . Definimos un "pseudocódigo" que debería resultar familiar a los lectores que hayan hecho programación de computadoras y usarlo para mostrar cómo especificaremos nuestros algoritmos. Teniendo especificado el algoritmo, luego argumentamos que ordena correctamente y analizamos su tiempo de ejecución. El análisis introduce una notación que se centra en cómo aumenta ese tiempo con el número de elementos a ordenar. Después de nuestra discusión sobre la ordenación por inserción, presentamos la división y conquistar el enfoque para el diseño de algoritmos y utilizarlo para desarrollar un algoritmo llamado fusión ordenar. Terminamos con un análisis del tiempo de ejecución del tipo de combinación.

2.1 Orden de inserción

Nuestro primer algoritmo, la ordenación por inserción, resuelve el *problema de ordenación* presentado en el Capítulo 1 :

- **Entrada:** una secuencia de n números $a_1, a_2, \ldots, una_n$.
- **Salida:** una permutación (reordenamiento) de la secuencia de entrada de modo que .

Los números que deseamos ordenar también se conocen como *claves* .

En este libro, describiremos típicamente algoritmos como programas escritos en un *pseudocódigo* que es similar en muchos aspectos a C, Pascal o Java. Si ha sido presentado a alguno de estos idiomas, no debería tener problemas para leer nuestros algoritmos. Que separa el pseudocódigo del código "real" es que en pseudocódigo, empleamos cualquier método expresivo que sea más claro y conciso para especificar un algoritmo dado. A veces, el método más claro es el inglés, así que no Sorpréndase si encuentra una frase u oración en inglés incrustada en una sección de código "real". Otra diferencia entre el pseudocódigo y el código real es que el pseudocódigo no es normalmente se ocupa de cuestiones de ingeniería de software. Problemas de abstracción de datos,

Página 19

modularidad y manejo de errores a menudo se ignoran para transmitir la esencia de la algoritmo de forma más concisa.

Comenzamos con la *ordenación por inserción* , que es un algoritmo eficiente para ordenar una pequeña cantidad de elementos. La clasificación por inserción funciona de la misma manera que muchas personas clasifican una mano de naipes. Empezamos con la mano izquierda vacía y las cartas boca abajo sobre la mesa. Luego retiramos una tarjeta a la tiempo de la mesa e insértelo en la posición correcta en la mano izquierda. Para encontrar el correcto posición para una carta, la comparamos con cada una de las cartas que ya están en la mano, de derecha a izquierda, como se ilustra en la Figura 2.1 . En todo momento, las cartas que se encuentran en la mano izquierda se ordenan y estas cartas eran originalmente las cartas superiores de la pila sobre la mesa.

Figura 2.1: Clasificación de una mano de cartas mediante clasificación por inserción.

Nuestro pseudocódigo para la ordenación por inserción se presenta como un procedimiento llamado INSERTION-SORT, que toma como parámetro una matriz A [1 n] que contiene una secuencia de longitud n que debe ser ordenados. (En el código, el número n de elementos en A se denota por la *longitud* [A].) La entrada Los números se ordenan en su lugar : los números se reordenan dentro de la matriz A , con como máximo un número constante de ellos almacenados fuera de la matriz en cualquier momento. La matriz de entrada A contiene el secuencia de salida ordenada cuando finaliza INSERTION-SORT.

```
INSERCIÓN-CLASIFICACIÓN ( A )
1 para j ← 2 a la longitud [ A ]
2 tecla hacer ← A [ j ]
3                ▸ Inserte A [ j ] en la secuencia ordenada A [1 j - 1].
4                i ← j - 1
5                mientras que i > 0 y A [ i ]> clave
6                        hacer A [ i + 1] ← A [ i ]
7                                yo ← yo - 1
8                Una tecla [ i + 1] ←
```

Invariantes de bucle y corrección de la ordenación por inserción

La figura 2.2 muestra cómo funciona este algoritmo para A = 5, 2, 4, 6, 1, 3. El índice j indica la "tarjeta actual" se inserta en la mano. Al comienzo de cada iteración del bucle **for** "externo" , que está indexado por j , el subarreglo que consta de elementos A [1 j - 1] constituyen la mano actualmente clasificada, y los elementos A [j + 1 n] corresponden a la pila de cartas todavía en la mesa. De hecho, los elementos A [1 j - 1] son los elementos *originalmente* en las posiciones 1 hasta j - 1, pero ahora en orden. Declaramos estas propiedades de A [1 j -1] formalmente como un invariante de bucle :

Página 20

- Al comienzo de cada iteración del bucle **for** de las líneas 1-8, el subarreglo A [1 j - 1] consta de los elementos originalmente en A [1 j - 1] pero en orden ordenado.

Figura 2.2: La operación de INSERTION-SORT en el arreglo A = 5, 2, 4, 6, 1, 3. Formación Los índices aparecen encima de los rectángulos y los valores almacenados en las posiciones de la matriz aparecen dentro del rectángulos. *(a)* - *(e)* Las iteraciones del bucle *for* de las líneas 1-8. En cada iteración, el negro rectángulo contiene la clave tomada de A [j], que se compara con los valores en sombreado rectángulos a su izquierda en la prueba de la línea 5. Las flechas sombreadas muestran los valores de la matriz movidos una posición a la derecha en la línea 6, y las flechas negras indican donde la clave se mueve en la línea 8. *(f)* La matriz ordenada final.

Usamos invariantes de bucle para ayudarnos a comprender por qué un algoritmo es correcto. Debemos mostrar tres cosas sobre un ciclo invariante:

- **Inicialización:** es cierto antes de la primera iteración del bucle.
- **Mantenimiento:** si es verdadero antes de una iteración del bucle, permanece verdadero antes de la próxima iteración.
- **Terminación:** cuando el ciclo termina, el invariante nos da una propiedad útil que ayuda a mostrar que el algoritmo es correcto.

Cuando se cumplen las dos primeras propiedades, el invariante de bucle es verdadero antes de cada iteración del lazo. Tenga en cuenta la similitud con la inducción matemática, donde para demostrar que una propiedad es válida, probar un caso base y un paso inductivo. Aquí, mostrando que el invariante se mantiene antes del primer iteración es como el caso base, y mostrar que el invariante se mantiene de una iteración a otra es como el paso inductivo.

La tercera propiedad es quizás la más importante, ya que estamos usando el ciclo invariante para mostrar corrección. También difiere del uso habitual de la inducción matemática, en el que la el paso inductivo se usa infinitamente; aquí, detenemos la "inducción" cuando termina el ciclo.

Veamos cómo se mantienen estas propiedades para la ordenación por inserción.

- **Inicialización:** Empezamos mostrando que el invariante del ciclo se mantiene antes del primer ciclo. iteración, cuando j = 2. [1] El subarreglo A [1 j - 1], por lo tanto, consiste solo en el

elemento A [1], que de hecho es el elemento original en A [1]. Además, este subarreglo es ordenados (trivialmente, por supuesto), lo que muestra que el invariante de bucle se mantiene antes de la primera iteración del bucle.

- **Mantenimiento: a** continuación, abordamos la segunda propiedad: mostrar que cada iteración mantiene el ciclo invariante. De manera informal, el cuerpo del bucle **for** externo funciona mediante moviendo A [j - 1], A [j - 2], A [j - 3], y así sucesivamente una posición a la derecha hasta que Se encuentra la posición adecuada para A [j] (líneas 4-7), en cuyo punto se inserta el valor de A [j] (línea 8). Un tratamiento más formal de la segunda propiedad requeriría que declaramos y mostrar una invariante de bucle para el "interior" **, mientras que** bucle. En este punto, sin embargo, preferimos no empantanarse en tal formalismo, por lo que confiamos en nuestro análisis informal para muestre que la segunda propiedad es válida para el bucle exterior.

Página 21

- **Terminación:** Finalmente, examinamos qué sucede cuando termina el ciclo. por ordenación por inserción, el bucle **for** externo termina cuando j excede n , es decir, cuando $j = n + 1$. Sustituyendo $n + 1$ por j en la redacción del ciclo invariante, tenemos que el subarreglo A [1 n] consta de los elementos originalmente en A [1 n], pero en orden ordenado. Pero el subarreglo A [1 n] es el arreglo completo! Por lo tanto, se ordena toda la matriz, lo que significa que el algoritmo es correcto.

Usaremos este método de invariantes de bucle para mostrar la corrección más adelante en este capítulo y en otros capítulos también.

Convenciones de pseudocódigo

Usamos las siguientes convenciones en nuestro pseudocódigo.

1. La sangría indica la estructura del bloque. Por ejemplo, el cuerpo del bucle **for** que comienza en la línea 1 consiste en líneas 2-8, y el cuerpo de la **mientras** bucle que comienza en la línea 5 contiene las líneas 6-7 pero no la línea 8. Nuestro estilo de sangría se aplica a **if-then-else** declaraciones también. Usando sangría en lugar de indicadores convencionales de bloque estructura, como las declaraciones de **inicio** y **finalización** , reduce en gran medida el desorden al tiempo que conserva, o incluso mejorando la claridad. [2]
2. Las construcciones de bucle **while** , **for** y **repeat** y las construcciones condicionales **si** , **entonces** , y **además** tienen interpretaciones similares a las de Pascal. [3] Hay uno sutil diferencia con respecto a los bucles **for** , sin embargo: en Pascal, el valor del contador de bucles La variable no está definida al salir del ciclo, pero en este libro, el contador de ciclo retiene su valor después de salir del bucle. Por lo tanto, inmediatamente después de un bucle **for** , el contador de bucles valor es el valor que primero excedió el límite del ciclo **for** . Usamos esta propiedad en nuestro argumento de corrección para la ordenación por inserción. El encabezado del bucle **for** en la línea 1 es **para** $j \leftarrow 2$ **a** *longitud* [A], por lo que cuando este bucle termina, $j =$ *longitud* [A] +1 (o, de manera equivalente, $j = n$ +1, ya que $n =$ *longitud* [A]).
3. El símbolo " ▸ " indica que el resto de la línea es un comentario.
4. Una asignación múltiple de la forma $i \leftarrow j \leftarrow e$ asigna a ambas variables i y j el valor de expresión e ; debe tratarse como equivalente a la asignación $j \leftarrow e$ seguida de la asignación $i \leftarrow j$.
5. Las variables (como i , j y *clave*) son locales para el procedimiento dado. No usaremos variables globales sin indicación explícita.
6. Se accede a los elementos de la matriz especificando el nombre de la matriz seguido del índice en corchetes. Por ejemplo, A [i] indica el i -ésimo elemento de la matriz A . los la notación "" se utiliza para indicar un rango de valores dentro de una matriz. Por tanto, A [1 j] indica el subarreglo de A *que* consta de los j elementos A [1], A [2],. . . , A [j].
7. Los datos compuestos generalmente se organizan en objetos , que se componen de atributos o campos . Se accede a un campo en particular utilizando el nombre del campo seguido del nombre de su objeto entre corchetes. Por ejemplo, tratamos una matriz como un objeto con la *longitud del* atributo que indica cuántos elementos contiene. Para especificar el número de elementos en una matriz A , escribimos *longitud* [A]. Aunque usamos corchetes para ambos indexación de matrices y atributos de objeto, generalmente quedará claro a partir del contexto qué se pretende la interpretación.

   Una variable que representa una matriz u objeto se trata como un puntero a los datos. que representa la matriz u objeto. Para todos los campos f de un objeto x , establecer $y \leftarrow x$ causa f [y] = f [x]. Además, si ahora establecemos f [x] ← 3, luego no solo es f [x] = 3, sino f [y] =

Página 22

3 también. En otras palabras, x y y punto a ("son") el mismo objeto después de la asignación $y \leftarrow x$.

A veces, un puntero no se referirá a ningún objeto. En este caso, le damos el especial valor NULO.

8. Los parámetros se pasan a un procedimiento **por valor** : el procedimiento llamado recibe su propio copia de los parámetros, y si asigna un valor a un parámetro, el cambio *no se* ve por el procedimiento de llamada. Cuando se pasan objetos, el puntero a los datos que representan el objeto se copia, pero los campos del objeto no. Por ejemplo, si x es un parámetro de un procedimiento llamado, la asignación $x \leftarrow y$ dentro del procedimiento llamado no es visible para el procedimiento de llamada. Sin embargo, la asignación $f[x] \leftarrow 3$ es visible.
9. Los operadores booleanos "y" y "o" están en **cortocircuito** . Es decir, cuando evaluamos la "expresión x e y " evaluamos primero x . Si x *se* evalúa como FALSO, entonces el expresión no se puede evaluar como VERDADERO, por lo que no evaluamos y . Si por el otro mano, x *se* evalúa como VERDADERO, debemos evaluar y para determinar el valor de la totalidad expresión. De manera similar, en la expresión " x o y " evaluamos la expresión y solo si x se evalúa como FALSO. Los operadores de cortocircuito nos permiten escribir expresiones booleanas como " $x \neq$ NIL $y f[x] = y$ " sin preocuparnos por lo que sucede cuando intentamos evalúe $f[x]$ cuando x es NULO.

Ejercicios 2.1-1

Usando la Figura 2.2 como modelo, ilustre la operación de INSERTION-SORT en el arreglo $A =$ 31, 41, 59, 26, 41, 58.

Ejercicios 2.1-2

Vuelva a escribir el procedimiento INSERTION-SORT para ordenar en no creciente en lugar de orden no decreciente.

Ejercicios 2.1-3

Considere el **problema de la búsqueda** :

- **Entrada:** una secuencia de n números $A = a_1, a_2, \ldots, una_n$ y un valor v .
- **Salida:** Un índice i tal que $v = A[i]$ o el valor especial NIL si v no aparece en *Una* .

Escriba un pseudocódigo para **búsqueda lineal** , que escanea a través de la secuencia, buscando v . Usando un invariante de bucle, demuestre que su algoritmo es correcto. Asegúrese de que su ciclo invariante cumpla las tres propiedades necesarias.

Página 23

Ejercicios 2.1-4

Considere el problema de la adición de dos n enteros binarios -bit, almacenados en dos n -elemento matrices A y B . La suma de los dos números enteros debe almacenarse en forma binaria en un elemento ($n + 1$)

array C . Enuncie el problema formalmente y escriba un pseudocódigo para sumar los dos enteros.

[1] Cuando el ciclo es un ciclo **for** , el momento en el que comprobamos el invariante del ciclo justo antes de la primera iteración es inmediatamente después de la asignación inicial a la variable de contador de bucle y justo antes de la primera prueba en el encabezado del bucle. En el caso de INSERTION-SORT, este tiempo es después de asignar 2 a la variable j pero antes de la primera prueba de si $j \leq longitud$ [A].

[2] En lenguajes de programación reales, generalmente no es aconsejable usar sangría sola para indican la estructura del bloque, ya que los niveles de sangría son difíciles de determinar cuando se divide el código en las páginas.

[3] La mayoría de los lenguajes estructurados en bloques tienen construcciones equivalentes, aunque la sintaxis exacta puede difieren de la de Pascal.

2.2 Analizar algoritmos

Analizar un algoritmo ha llegado a significar predecir los recursos que el algoritmo requiere. En ocasiones, recursos como la memoria, el ancho de banda de comunicación o la computadora El hardware es una preocupación principal, pero la mayoría de las veces es el tiempo computacional lo que queremos medida. Por lo general, al analizar varios algoritmos candidatos para un problema, el método más eficiente uno puede identificarse fácilmente. Dicho análisis puede indicar más de un candidato viable, pero varios algoritmos inferiores generalmente se descartan en el proceso.

Antes de que podamos analizar un algoritmo, debemos tener un modelo de la tecnología de implementación que se utilizará, incluido un modelo para los recursos de esa tecnología y sus costos. por la mayor parte de este libro, asumiremos una máquina genérica de acceso aleatorio (RAM) de un procesador modelo de computación como nuestra tecnología de implementación y entendemos que nuestros algoritmos se implementará como programas de computadora. En el modelo RAM, las instrucciones se ejecutan uno tras otro, sin operaciones concurrentes. En capítulos posteriores, sin embargo, tendremos ocasión para investigar modelos para hardware digital.

Estrictamente hablando, se deben definir con precisión las instrucciones del modelo RAM y su costos. Sin embargo, hacerlo sería tedioso y proporcionaría poca información sobre el algoritmo. diseño y análisis. Sin embargo, debemos tener cuidado de no abusar del modelo RAM. Por ejemplo, qué si una RAM tuviera una instrucción que ordena? Entonces podríamos clasificar en una sola instrucción. Tal La RAM sería poco realista, ya que las computadoras reales no tienen tales instrucciones. Nuestra guía, por lo tanto, así se diseñan las computadoras reales. El modelo RAM contiene instrucciones comúnmente encontrado en computadoras reales: aritmética (sumar, restar, multiplicar, dividir, restar, piso, techo), movimiento de datos (carga, almacenamiento, copia) y control (condicional e incondicional bifurcación, llamada de subrutina y retorno). Cada una de estas instrucciones requiere una cantidad constante de tiempo.

Página 24

Los tipos de datos en el modelo RAM son enteros y de coma flotante. Aunque normalmente no nos preocupamos por la precisión en este libro, en algunas aplicaciones la precisión es crucial. Nosotros también suponga un límite en el tamaño de cada palabra de datos. Por ejemplo, cuando se trabaja con entradas de tamaño n , normalmente asumimos que los enteros están representados por c lg n bits para alguna constante $c \geq 1$. Requerimos $c \geq 1$ para que cada palabra pueda contener el valor de n , lo que nos permite indexar elementos de entrada individuales, y restringimos c para que sea una constante para que el tamaño de la palabra no crecer arbitrariamente. (Si el tamaño de la palabra pudiera crecer arbitrariamente, podríamos almacenar grandes cantidades de datos en una palabra y operar en todo en un tiempo constante, claramente un escenario poco realista).

Las computadoras reales contienen instrucciones que no se enumeran arriba, y tales instrucciones representan un gris área en el modelo RAM. Por ejemplo, ¿es la exponenciación una instrucción de tiempo constante? En el caso general, no; se tarda varias instrucciones para calcular x^y cuando X y Y son números reales. En En situaciones restringidas, sin embargo, la exponenciación es una operación de tiempo constante. Muchas computadoras tener una instrucción de "desplazamiento a la izquierda", que en tiempo constante desplaza los bits de un entero en k posiciones a la izquierda. En la mayoría de las computadoras, cambiar los bits de un número entero en una posición izquierda es equivalente a multiplicar por 2. Desplazar los bits k posiciones a la izquierda es equivalente a la multiplicación por 2^k . Por lo tanto, tales computadoras pueden calcular 2^k en un tiempo constante instrucción desplazando el número entero 1 en k posiciones a la izquierda, siempre que k no sea mayor que el número de bits en una palabra de computadora. Nos esforzaremos por evitar tales áreas grises en la RAM modelo, pero trataremos el cálculo de 2^k como una operación de tiempo constante cuando k es un pequeño suficiente entero positivo.

En el modelo de RAM, no intentamos modelar la jerarquía de memoria que es común en computadoras contemporáneas. Es decir, no modelamos cachés ni memoria virtual (que es más a menudo implementado con paginación de demanda). Varios modelos computacionales intentan dar cuenta para efectos de jerarquía de memoria, que a veces son importantes en programas reales en máquinas. Un puñado de problemas en este libro examinan los efectos de la jerarquía de la memoria, pero para En su mayor parte, los análisis de este libro no los considerarán. Modelos que incluyen la memoria La jerarquía es un poco más compleja que el modelo RAM, por lo que puede ser difícil trabajar con. Además, los análisis del modelo RAM suelen ser excelentes predictores del rendimiento. en máquinas reales.

Analizar incluso un algoritmo simple en el modelo de RAM puede ser un desafío. El matemático Las herramientas necesarias pueden incluir combinatoria, teoría de la probabilidad, destreza algebraica y capacidad para identificar los términos más significativos en una fórmula. Porque el comportamiento de un El algoritmo puede ser diferente para cada entrada posible, necesitamos un medio para resumir que comportamiento en fórmulas sencillas y fáciles de entender.

Aunque normalmente seleccionamos solo un modelo de máquina para analizar un algoritmo dado, todavía enfrentamos muchas opciones para decidir cómo expresar nuestro análisis. Nos gustaría una forma que sea simple de escribir y manipular, muestra las características importantes del recurso de un algoritmo requisitos y suprime los tediosos detalles.

Análisis de clasificación de inserción

El tiempo que tarda el procedimiento INSERTION-SORT depende de la entrada: ordenar mil números lleva más tiempo que ordenar tres números. Además, INSERTION-SORT puede tomar diferentes cantidades de tiempo para clasificar dos secuencias de entrada del mismo tamaño dependiendo de cómo casi ordenados ya lo están. En general, el tiempo que tarda un algoritmo crece con el tamaño de la entrada, por lo que es tradicional describir el tiempo de ejecución de un programa en función de la

Página 25

tamaño de su entrada. Para hacerlo, debemos definir los términos "tiempo de ejecución" y "tamaño de la entrada". mas cuidadosamente.

La mejor noción de tamaño de entrada depende del problema que se esté estudiando. Para muchos problemas como clasificar o calcular transformadas discretas de Fourier, la medida más natural es la *número de elementos en la entrada* , por ejemplo, el tamaño de la matriz n para ordenar. Para muchos otros problemas, como multiplicar dos números enteros, la mejor medida del tamaño de entrada es el *número total de bits* necesarios para representar la entrada en notación binaria ordinaria. A veces, es más apropiado para describir el tamaño de la entrada con dos números en lugar de uno. Por ejemplo, si la entrada a un algoritmo es un gráfico, el tamaño de la entrada se puede describir por el número de vértices y aristas en el gráfico. Indicaremos qué medida de tamaño de entrada se está utilizando con cada problema que estudiamos.

El tiempo de ejecución de un algoritmo en una entrada particular es el número de operaciones primitivas o "pasos" ejecutados. Es conveniente definir la noción de paso para que sea como máquina independiente como sea posible. Por el momento, adoptemos el siguiente punto de vista. Una cantidad constante Se requiere de tiempo para ejecutar cada línea de nuestro pseudocódigo. Una línea puede tomar una diferente cantidad de tiempo que otra línea, pero asumiremos que cada ejecución de la i- ésima línea toma tiempo c_i , donde c_i es una constante. Este punto de vista está en consonancia con el modelo RAM, y también refleja cómo se implementaría el pseudocódigo en la mayoría de las computadoras reales. [4]

En la siguiente discusión, nuestra expresión para el tiempo de ejecución de INSERTION-SORT será evolucionar de una fórmula desordenada que usa todos los costos de declaración c_i a una notación mucho más simple que es más conciso y más fácil de manipular. Esta notación más simple también lo hará más fácil para determinar si un algoritmo es más eficiente que otro.

Comenzamos presentando el procedimiento INSERTION-SORT con el "costo" de tiempo de cada instrucción y el número de veces que se ejecuta cada instrucción. Para cada $j = 2, 3, \ldots, n$, donde $n = longitud[A]$, dejamos que t_j sea el número de veces **que** se ejecuta la prueba del ciclo while en la línea 5 para ese valor de j . Cuando una **por** o **mientras que** las salidas de bucle de la forma habitual (es decir, debido a la prueba en el bucle header), la prueba se ejecuta una vez más que el cuerpo del bucle. Asumimos que los comentarios son declaraciones no ejecutables, por lo que no toman tiempo.

```
INSERCIÓN-CLASIFICACIÓN ( A )                          costo     veces
1 para j ← 2 a la longitud [ A ]                       c 1       norte
2 tecla hacer ← A [ j ]                                c 2       n - 1
3                ▸ Inserte A [ j ] en el ordenado
```

4	$i \leftarrow j$ - 1 secuencia A [1 j - 1]. 0 n - 1	c_4	n - 1
5	**mientras que** $i > 0$ y A [i]> clave	c_5	
6	**hacer** A [i + 1] ← A [i]	c_6	
7	$yo \leftarrow yo$ - 1	c_7	
8	*Una tecla* [i + 1] ←	c_8	n - 1

El tiempo de ejecución del algoritmo es la suma de los tiempos de ejecución de cada instrucción ejecutada; una declaración que toma c_i pasos para ejecutarse y se ejecuta n veces contribuirá $c_i n$ al total tiempo de ejecución. [5] Para calcular $T(n)$, el tiempo de ejecución de INSERTION-SORT, sumamos el productos de las columnas de *costos* y *tiempos* , obteniendo

Página 26

Incluso para entradas de un tamaño dado, el tiempo de ejecución de un algoritmo puede depender de *qué* entrada de se da ese tamaño. Por ejemplo, en INSERTION-SORT, el mejor caso ocurre si la matriz es ya ordenado. Para cada $j = 2, 3, \ldots, n$, entonces encontramos que $A[i] \leq clave$ en la línea 5 cuando i tiene su valor inicial de j - 1. Por lo tanto, $t_j = 1$ para $j = 2, 3, \ldots, n$, y el mejor tiempo de ejecución es

$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 (n - 1) + c_8 (n - 1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8).$$

Este tiempo de funcionamiento se puede expresar como $an + b$ para *las constantes* a y b que dependen de la costos de declaración c_i ; por tanto, es una función lineal de n .

Si la matriz está ordenada en orden inverso, es decir, en orden decreciente, se produce el peor de los casos. Nosotros debe comparar cada elemento $A[j]$ con cada elemento en todo el subarreglo ordenado $A[1 j - 1]$, y entonces $t_j = j$ para $j = 2, 3, \ldots, n$. Señalando que

y

(vea el Apéndice A para una revisión de cómo resolver estas sumas), encontramos que en el peor caso, el tiempo de ejecución de INSERTION-SORT es

Esta peor de los casos el tiempo de funcionamiento se puede expresar como $an^2 + bn + c$ para las constantes a , b , y c que nuevamente dependerán de los costos del estado de cuenta c_i ; por tanto, es una función cuadrática de n .

Normalmente, como en la ordenación por inserción, el tiempo de ejecución de un algoritmo es fijo para una entrada determinada, aunque en capítulos posteriores veremos algunos algoritmos "aleatorios" interesantes cuyos el comportamiento puede variar incluso para una entrada fija.

Análisis de el peor caso y caso promedio

Página 27

En nuestro análisis de ordenación por inserción, analizamos el mejor de los casos, en el que la matriz de entrada era ya ordenado, y el peor de los casos, en el que la matriz de entrada se ordenó al revés. Para el
En el resto de este libro, sin embargo, normalmente nos concentraremos en encontrar sólo el peor de los casos tiempo de ejecución , es decir, el tiempo de ejecución más largo para *cualquier* entrada de tamaño n . Damos tres razones para esta orientación.

- El peor tiempo de ejecución de un algoritmo es un límite superior en el tiempo de ejecución para cualquier entrada. Saberlo nos da la garantía de que el algoritmo nunca tomará ninguna más. No necesitamos hacer algunas conjeturas sobre el tiempo de ejecución y esperar que nunca empeora mucho.
- Para algunos algoritmos, el peor de los casos ocurre con bastante frecuencia. Por ejemplo, al buscar un base de datos para un dato particular, el peor caso del algoritmo de búsqueda a menudo ocurre cuando la información no está presente en la base de datos. En alguna búsqueda aplicaciones, las búsquedas de información ausente pueden ser frecuentes.
- El "caso medio" es a menudo tan malo como el peor de los casos. Supongamos que nosotros elija aleatoriamente n números y aplique la ordenación por inserción. ¿Cuánto tiempo se tarda en determinar dónde en el subarreglo A [1 j - 1] insertar el elemento A [j]? En promedio, la mitad de los elementos en A [1 j - 1] son menores que A [j], y la mitad de los elementos son mayores. En promedio, por lo tanto, verificamos la mitad del subarreglo A [1 j - 1], entonces $t_j = j / 2$. Si trabajamos el tiempo de ejecución promedio resultante del caso, resulta ser una función cuadrática de el tamaño de entrada, al igual que el peor tiempo de ejecución.

En algunos casos particulares, nos interesará el caso promedio o el tiempo de ejecución esperado de un algoritmo; En el capítulo 5 , veremos la técnica del análisis probabilístico , mediante la cual determinamos los tiempos de ejecución esperados. Un problema al realizar un caso promedio Sin embargo, el análisis es que puede no ser evidente qué constituye un insumo "promedio" para un Problema particular. A menudo, asumiremos que todas las entradas de un tamaño dado son igualmente probables. En la práctica, este supuesto puede ser violada, pero a veces se puede utilizar una aleatorizado algoritmo , que hace elecciones aleatorias, para permitir un análisis probabilístico.

Orden de crecimiento

Usamos algunas abstracciones simplificadoras para facilitar nuestro análisis de INSERTION-SORT procedimiento. Primero, ignoramos el costo real de cada enunciado, usando las constantes c_i para representan estos costos. Luego, observamos que incluso estas constantes nos dan más detalles de lo que realmente necesita: tiempo del peor caso de ejecución es $an^2 + bn + c$ para algunas constantes de a , b , y c que dependen de los costos del estado de cuenta c_i . Por lo tanto, ignoramos no solo los costos de declaración reales, sino también los costos abstractos c_i .

Ahora haremos una abstracción más simplificadora. Es la tasa de crecimiento , o el orden de crecimiento , del tiempo de ejecución que realmente nos interesa. Por lo tanto, consideramos solo los principales término de una fórmula (por ejemplo, an^2), ya que los términos de orden inferior son relativamente insignificantes para grandes n . También ignoramos el coeficiente constante del término principal, ya que los factores constantes son menos significativo que la tasa de crecimiento en la determinación de la eficiencia computacional para grandes insumos. Por lo tanto, escribimos que la ordenación por inserción, por ejemplo, tiene un tiempo de ejecución en el peor de los casos de $\Theta(n^2)$ (pronunciado "theta de n- cuadrado"). Usaremos la notación Θ de manera informal en este capítulo; va a definirse con precisión en el Capítulo 3 .

Por lo general, consideramos que un algoritmo es más eficiente que otro si se ejecuta en el peor de los casos. el tiempo tiene un orden de crecimiento más bajo. Debido a factores constantes y términos de orden inferior, esto

Página 28

la evaluación puede ser errónea para entradas pequeñas. Pero para entradas lo suficientemente grandes, un algoritmo $\Theta(n^2)$, para ejemplo, se ejecutará más rápidamente en el peor de los casos que un algoritmo $\Theta(n^3)$.

Ejercicios 2.2-1

Expresar la función $n^3 / 1-100 n^2 - 100 n + 3$ en términos de Θ-notación.

Ejercicios 2.2-2

Considere ordenar n números almacenados en la matriz A encontrando primero el elemento más pequeño de A y intercambiándolo con el elemento en A [1]. Luego encuentre el segundo elemento más pequeño de A , y cámbielo por A [2]. Continúe de esta manera para la primera n - 1 elementos de A . Escribir pseudocódigo para este algoritmo, que se conoce como clasificación de selección . ¿Qué ciclo invariante hace mantener este algoritmo? ¿Por qué necesita ejecutarse solo para los primeros n - 1 elementos, en lugar de para todos los n elementos? Indique los tiempos de ejecución del mejor y del peor caso de clasificación de selección en Θ-notación.

Ejercicios 2.2-3

Considere la búsqueda lineal nuevamente (vea el ejercicio 2.1-3). Cuántos elementos de la secuencia de entrada deben comprobarse en promedio, asumiendo que el elemento que se busca es igualmente probable que haya algún elemento en la matriz? ¿Qué tal en el peor de los casos? ¿Cuáles son los casos promedio y tiempos de ejecución en el peor de los casos de búsqueda lineal en notación Θ? Justifica tus respuestas.

Ejercicios 2.2-4

¿Cómo podemos modificar casi cualquier algoritmo para tener un buen tiempo de ejecución en el mejor de los casos?

[4] Aquí hay algunas sutilezas. Los pasos computacionales que especificamos en inglés son a menudo variantes de un procedimiento que requiere más que una cantidad constante de tiempo. Por ejemplo, más adelante en este libro podríamos decir "ordenar los puntos por la coordenada x ", que, como veremos, toma más que una cantidad constante de tiempo. Además, tenga en cuenta que una declaración que llama a una subrutina toma tiempo constante, aunque la subrutina, una vez invocada, puede tardar más. Es decir, separamos el proceso de llamar a la subrutina - pasarle parámetros, etc. - desde el proceso de ejecución la subrutina.

[5] Esta característica no es necesariamente válida para un recurso como la memoria. Una declaración que hace referencia a m palabras de memoria y se ejecuta n veces no necesariamente consume mn palabras de memoria en total.

2.3 Diseñar algoritmos

Hay muchas formas de diseñar algoritmos. La ordenación por inserción utiliza un enfoque incremental : habiendo ordenado el subarreglo A [1 j - 1], insertamos el elemento A [j] en su lugar apropiado, produciendo el subarreglo ordenado A [1 j].

En esta sección, examinamos un enfoque de diseño alternativo, conocido como "divide y vencerás". Usaremos divide y vencerás para diseñar un algoritmo de clasificación cuyo tiempo de ejecución en el peor de los casos es mucho menor que el del tipo de inserción. Una ventaja de los algoritmos de divide y vencerás es que sus tiempos de ejecución a menudo se determinan fácilmente utilizando técnicas que se introducirán en Capítulo 4 .

2.3.1 El enfoque de divide y vencerás

Muchos algoritmos útiles tienen una estructura recursiva : para resolver un problema dado, llaman

ellos mismos de forma recursiva una o más veces para hacer frente a subproblemas estrechamente relacionados. Estas
Los algoritmos suelen seguir un enfoque de *divide y vencerás* : dividen el problema en
varios subproblemas que son similares al problema original pero de menor tamaño, resuelva el
subproblemas de forma recursiva, y luego combinar estas soluciones para crear una solución al original
problema.

El paradigma de divide y vencerás implica tres pasos en cada nivel de la recursividad:

- **Divida** el problema en varios subproblemas.
- **Conquista** los subproblemas resolviéndolos de forma recursiva. Si los tamaños de los subproblemas son lo suficientemente pequeño, sin embargo, simplemente resuelva los subproblemas de una manera sencilla.
- **Combine** las soluciones de los subproblemas en la solución del problema original.

El algoritmo de *ordenación por fusión* sigue de cerca el paradigma de divide y vencerás. Intuitivamente,
opera de la siguiente manera.

- **Dividir:** divide la secuencia de n elementos que se ordenarán en dos subsecuencias de $n / 2$ elementos cada uno.
- **Conquista:** ordena las dos subsecuencias de forma recursiva utilizando la ordenación por combinación.
- **Combinar: combine** las dos subsecuencias ordenadas para producir la respuesta ordenada.

La recursividad "toca fondo" cuando la secuencia que se va a ordenar tiene una longitud de 1, en cuyo caso
no hay trabajo por hacer, ya que cada secuencia de longitud 1 ya está ordenada.

La operación clave del algoritmo de clasificación por fusión es la fusión de dos secuencias ordenadas en el
paso "combinar". Para realizar la fusión, usamos un procedimiento auxiliar MERGE (A , p , q , r),
donde A es una matriz y p , q , y r son índices elementos de la matriz de tal manera que la numeración $p \le q < r$. El procedimiento supone que los subarreglos A [pq] y A [$q + 1$ r] están ordenados.
Los *fusiona* para formar un único subarreglo ordenado que reemplaza al actual subarreglo A [pr].

Página 30

Nuestro procedimiento MERGE toma tiempo $\Theta (n)$, donde $n = r - p + 1$ es el número de elementos que
combinado, y funciona de la siguiente manera. Volviendo a nuestro motivo de juego de cartas, supongamos que tenemos dos
montones de cartas boca arriba sobre una mesa. Cada pila está ordenada, con las cartas más pequeñas en la parte superior. Deseamos
para fusionar las dos pilas en una sola pila de salida ordenada, que debe estar boca abajo sobre la mesa.
Nuestro paso básico consiste en elegir la más pequeña de las dos cartas en la parte superior de las pilas boca arriba,
sacándola de su pila (que expone una nueva carta superior) y colocando esta carta boca abajo
en la pila de salida. Repetimos este paso hasta que una pila de entrada esté vacía, momento en el que simplemente
tome la pila de entrada restante y colóquela boca abajo sobre la pila de salida. Computacionalmente,
cada paso básico lleva un tiempo constante, ya que solo estamos comprobando dos cartas superiores. Desde que nosotros
realizar como máximo n pasos básicos, la fusión lleva $\Theta (n)$ tiempo.

El siguiente pseudocódigo implementa la idea anterior, pero con un giro adicional que evita
tener que comprobar si alguna pila está vacía en cada paso básico. La idea es ponerse el
parte inferior de cada pila una carta *centinela* , que contiene un valor especial que usamos para simplificar nuestra
código. Aquí, usamos ∞ como valor centinela, de modo que siempre que una carta con ∞ esté expuesta,
no puede ser la carta más pequeña a menos que ambas pilas tengan expuestas sus cartas de centinela. Pero una vez que
sucede, todas las cartas que no son centinelas ya se han colocado en la pila de salida. Desde que nosotros
saber de antemano que exactamente $r - p + 1$ cartas se colocarán en la pila de salida, podemos detener
una vez que hayamos realizado tantos pasos básicos.

```
FUSIONAR ( A , p , q , r )
 1 n1 ← q - p + 1
 2 n2 ← r - q
 3 crear matrices L [1 n1 + 1] y R [1 n2 + 1]
 4 para i ← 1 a n1
 5 hacer L [ i ] ← A [ p + i - 1]
 6 para j ← 1 a n2
 7 haz R [ j ] ← A [ q + j ]
 8 L [ n1 + 1] ← ∞
 9 R [ n2 + 1] ← ∞
10 i ← 1
11 j ← 1
12 para k ← p a r
13 hacer si L [ i ] ≤ R [ j ]
14                    luego A [ k ] ← L [ i ]
15                          yo ← yo + 1
dieciséis                 más A [ k ] ← R [ j ]
17                          j ← j + 1
```

En detalle, el procedimiento MERGE funciona de la siguiente manera. La línea 1 calcula la longitud n_1 del subarreglo $A[p..q]$, y la línea 2 calcula la longitud n_2 del subarreglo $A[q+1..r]$. Nosotros creamos matrices L y R ("izquierda" y "derecha"), de longitudes $n_1 + 1$ y $n_2 + 1$, respectivamente, en la línea 3. La **para** El bucle de las líneas 4-5 copia el subarreglo $A[p..q]$ en $L[1..n_1]$, y el bucle **for** de las líneas 6-7 copia el subarreglo $A[q+1..r]$ en $R[1..n_2]$. Las lneas 8-9 colocan a los centinelas en los extremos del matrices L y R. Las líneas 10-17, ilustradas en la Figura 2.3, realizan los pasos básicos $r - p + 1$ mediante manteniendo el siguiente ciclo invariante:

- Al comienzo de cada iteración del bucle **for** de las líneas 12-17, el subarreglo $A[p..k-1]$ contiene los $k - p$ elementos más pequeños de $L[1..n_1+1]$ y $R[1..n_2+1]$, en orden orden. Además, $L[i]$ y $R[j]$ son los elementos más pequeños de sus matrices que no tienen sido copiada de nuevo en *una* .

Página 31

Figura 2.3: La operación de las líneas 10-17 en la llamada MERGE (A , 9, 12, 16), cuando el subarreglo $A[9..16]$ contiene la secuencia 2, 4, 5, 7, 1, 2, 3, 6. Después de copiar e insertar centinelas, la matriz L contiene 2, 4, 5, 7, ∞, y la matriz R contiene 1, 2, 3, 6, ∞. Las posiciones ligeramente sombreadas en A contienen sus valores finales, y las posiciones ligeramente sombreadas en L y R contiene valores que aún no se han copiado de nuevo en *una* . Tomados en conjunto, los ligeramente sombreados las posiciones siempre comprenden los valores originalmente en $A[9..16]$, junto con los dos centinelas. Las posiciones muy sombreadas en A contienen valores que se copiarán y están muy sombreados posiciones en L y R contienen valores que ya han sido copiados de vuelta a A . *(a)* - *(h)* El matrices A, L y R , y sus respectivos índices k, i , y j antes de cada iteración del ciclo de líneas 12-17. *(i)* Las matrices e índices al final. En este punto, el subarreglo en $A[9..16]$ está ordenado, y los dos centinelas en L y R son los únicos dos elementos en estas matrices que tienen no se han copiado en *una* .

Debemos mostrar que este invariante de ciclo se mantiene antes de la primera iteración del ciclo **for** de líneas 12-17, que cada iteración del bucle mantiene el invariante y que el invariante proporciona un propiedad útil para mostrar la corrección cuando termina el ciclo.

- **Inicialización:** antes de la primera iteración del ciclo, tenemos $k = p$, de modo que el el subarreglo $A[p..k-1]$ está vacío. Este subarreglo vacío contiene el $k - p = 0$ más pequeño elementos de L y R , y como $i = j = 1$, tanto $L[i]$ como $R[j]$ son los elementos más pequeños de sus matrices que no han sido copiadas de nuevo en *una* .

- **Mantenimiento:** para ver que cada iteración mantiene el ciclo invariante, primero suponga que $L[i] \le R[j]$. Entonces $L[i]$ es el elemento más pequeño aún no copiada de nuevo en *una* . Dado que $A[p..k-1]$ contiene los $k - p$ elementos más pequeños, después de la línea 14 copia $L[i]$ en $A[k]$, el subarreglo $A[p..k]$ contendrá los $k - p + 1$ elementos más pequeños. Incrementando

k (en la actualización del ciclo **for**) e *i* (en la línea 15) restablece el invariante del ciclo para el siguiente

Página 32

iteración. Si en cambio $L[i] > R[j]$, entonces las líneas 16-17 realizan la acción apropiada para mantener el bucle invariante.

- **Terminación:** En la terminación, $k = r + 1$. Por el ciclo invariante, el subarreglo $A[pk - 1]$, que es $A[pr]$, contiene los $k - p = r - p + 1$ elementos más pequeños de $L[1 n_1 + 1]$ y $R[1 n_2 + 1]$, en orden clasificado. Las matrices L y R juntas contienen $n_1 + n_2 + 2 = r - p + 3$ elementos. Todos menos los dos más grandes se han copiado de nuevo en A , y estos dos los elementos más grandes son los centinelas.

Para ver que el procedimiento MERGE se ejecuta en $\Theta(n)$ tiempo, donde $n = r - p + 1$, observe que cada uno de las líneas 1-3 y 8-11 toman un tiempo constante, los bucles **for** de las líneas 4-7 toman $\Theta(n_1 + n_2) = \Theta(n)$ tiempo, [6] y hay n iteraciones del ciclo **for** de las líneas 12-17, cada una de las cuales toma constante hora.

Ahora podemos usar el procedimiento MERGE como una subrutina en el algoritmo de clasificación por fusión. los El procedimiento MERGE-SORT (A, p, r) ordena los elementos del subarreglo $A[pr]$. Si $p \geq r$, el subarreglo tiene como máximo un elemento y, por lo tanto, ya está ordenado. De lo contrario, el paso de dividir simplemente calcula un índice q que divide $A[pr]$ en dos subarreglos: $A[pq]$, que contiene $\lceil n / 2 \rceil$ elementos, y $A[q + 1 r]$, que contiene $\lfloor n / 2 \rfloor$ elementos. [7]

```
FUSIÓN-CLASIFICACIÓN ( A , p , r )
1 si p < r
2 luego q ← ⌊ ( p + r ) / 2⌋
3              FUSIÓN-CLASIFICACIÓN ( A , p , q )
4              FUSIÓN-CLASIFICACIÓN ( A , q + 1, r )
5              FUSIONAR ( A , p , q , r )
```

Para ordenar la secuencia completa $A = A[1], A[2],. . . , A[n]$, hacemos la llamada inicial MERGE-SORT (A , 1, *longitud* [A]), donde una vez más *longitud* [A] = n . La figura 2.4 ilustra el funcionamiento de el procedimiento de abajo hacia arriba cuando n es una potencia de 2. El algoritmo consiste en fusionar pares de Secuencias de 1 elemento para formar secuencias ordenadas de longitud 2, fusionando pares de secuencias de longitud 2 para formar secuencias ordenadas de longitud 4, y así sucesivamente, hasta que se fusionen dos secuencias de longitud $n / 2$ para formar la secuencia final ordenada de longitud n .

Figura 2.4: La operación de clasificación por fusión en la matriz $A = 5, 2, 4, 7, 1, 3, 2, 6$. Las longitudes de las secuencias ordenadas que se fusionan aumentan a medida que el algoritmo avanza de abajo hacia arriba.

2.3.2 Análisis de algoritmos de divide y vencerás

Página 33

Cuando un algoritmo contiene una llamada recursiva a sí mismo, su tiempo de ejecución a menudo se puede describir por una ecuación de recurrencia o recurrencia , que describe el tiempo de ejecución general en un problema del tamaño n en términos del tiempo de ejecución en entradas más pequeñas. Entonces podemos usar

herramientas matemáticas para resolver la recurrencia y proporcionar límites en el desempeño de la algoritmo.

Una recurrencia para el tiempo de ejecución de un algoritmo de divide y vencerás se basa en los tres pasos del paradigma básico. Como antes, dejamos que $T(n)$ sea el tiempo de ejecución de un problema de tamaño n. Si el tamaño del problema es lo suficientemente pequeño, digamos $n \leq c$ para alguna constante c, el sencillo La solución toma un tiempo constante, que escribimos como $\Theta(1)$. Supongamos que nuestra división del problema produce *un* subproblema, cada uno de los cuales es $1/b$ del tamaño del original. (Para ordenar por fusión, tanto a como b son 2, pero veremos muchos algoritmos de divide y vencerás en los que $a \neq b$.) Si tomamos $D(n)$ tiempo para dividir el problema en subproblemas y $C(n)$ tiempo para combinar los soluciones a los subproblemas en la solución al problema original, obtenemos la recurrencia

En el capítulo 4 , veremos cómo resolver las recurrencias comunes de esta forma.

Análisis de clasificación de combinación

Aunque el pseudocódigo para MERGE-SORT funciona correctamente cuando el número de elementos es ni siquiera, nuestro análisis basado en la recurrencia se simplifica si asumimos que el problema original el tamaño es una potencia de 2. Cada paso de división produce dos subsecuencias de tamaño exactamente $n/2$. En En el capítulo 4 , veremos que este supuesto no afecta el orden de crecimiento de la solución. a la recurrencia.

Razonamos de la siguiente manera para configurar la recurrencia para $T(n)$, el peor tiempo de ejecución de fusión ordenar en n números. Combinar la ordenación en un solo elemento lleva un tiempo constante. Cuando tenemos $n > 1$ elementos, desglosamos el tiempo de ejecución de la siguiente manera.

- **Dividir:** el paso de división solo calcula la mitad del subarreglo, que toma tiempo constante. Por lo tanto, $D(n) = \Theta(1)$.
- **Conquista: Resolvemos de forma** recursiva dos subproblemas, cada uno de tamaño $n/2$, que contribuye $2T(n/2)$ al tiempo de ejecución.
- **combinar:** Ya hemos señalado que el procedimiento de combinación en una n -elemento subarreglo toma tiempo $\Theta(n)$, entonces $C(n) = \Theta(n)$.

Cuando sumamos las funciones $D(n)$ y $C(n)$ para el análisis de ordenación por fusión, estamos agregando un función que es $\Theta(n)$ y una función que es $\Theta(1)$. Esta suma es una función lineal de n, es decir, $\Theta(n)$. Sumarlo al término $2T(n/2)$ del paso "conquistar" da la recurrencia para el tiempo de ejecución en el peor de los casos $T(n)$ del tipo de combinación:

(2,1)

En el capítulo 4 , veremos el "teorema maestro", que podemos usar para demostrar que $T(n)$ es $\Theta(n \lg n)$, donde $\lg n$ representa $\log_2 n$. Debido a que la función logaritmo crece más lentamente que cualquier

Página 34

función lineal, para entradas lo suficientemente grandes, combinación de clasificación, con su tiempo de ejecución $\Theta(n \lg n)$, supera a la ordenación por inserción, cuyo tiempo de ejecución es $\Theta(n^2)$, en el peor de los casos.

No necesitamos el teorema maestro para comprender intuitivamente por qué la solución al la recurrencia (2.1) es $T(n) = \Theta(n \lg n)$. Reescribamos la recurrencia (2.1) como

(2,2)

donde la constante c representa el tiempo requerido para resolver problemas de tamaño 1 así como el tiempo por elemento de matriz de los pasos de dividir y combinar. [8]

La figura 2.5 muestra cómo podemos resolver la recurrencia (2.2). Por conveniencia, asumimos que n es una potencia exacta de 2. La parte (a) de la figura muestra $T(n)$, que en la parte (b) se ha ampliado en un árbol equivalente que representa la recurrencia. El término cn es la raíz (el costo en la parte superior nivel de recursividad), y los dos subárboles de la raíz son las dos recurrencias más pequeñas $T(n/2)$. La parte (c) muestra que este proceso avanzó un paso más al expandir $T(n/2)$. El costo de cada uno de los dos subnodos en el segundo nivel de recursividad es $cn/2$. Seguimos ampliando cada nodo

en el árbol rompiéndolo en sus partes constituyentes según lo determinado por la recurrencia, hasta que el los tamaños de los problemas se reducen a 1, cada uno con un costo de c . La parte (d) muestra el árbol resultante.

Figura 2.5: La construcción de un árbol de recursividad para la recurrencia $T(n) = 2T(n/2) + cn$. Parte *(a)* muestra $T(n)$, que se expande progresivamente en *(b)* - *(d)* para formar el árbol de recursividad. los

Página 35

El árbol completamente expandido en la parte (d) tiene $\lg n + 1$ niveles (es decir, tiene una altura $\lg n$, como se indica), y cada nivel aporta un costo total de cn . El costo total, por lo tanto, es $cn \lg n + cn$, que es $\Theta(n \lg n)$.

A continuación, sumamos los costos en cada nivel del árbol. El nivel superior tiene un costo total cn , el siguiente el nivel hacia abajo tiene el costo total $c(n/2) + c(n/2) = cn$, el nivel posterior tiene el costo total $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, y así sucesivamente. En general, el nivel i debajo de la parte superior tiene 2^i nodos, cada uno contribuyendo con un costo de $c(n/2^i)$, de modo que el i- ésimo nivel por debajo de la parte superior tiene un costo total $2^i c(n/2^i) = cn$. En el nivel inferior, hay n nodos, cada uno contribuyendo con un costo de c , para un costo total de cn .

El número total de niveles del "árbol de recursividad" en la Figura 2.5 es $\lg n + 1$. Este hecho es fácilmente visto por un argumento inductivo informal. El caso base ocurre cuando $n = 1$, en cuyo caso hay es solo un nivel. Dado que $\lg 1 = 0$, tenemos que $\lg n + 1$ da el número correcto de niveles. Ahora suponga como hipótesis inductiva que el número de niveles de un árbol de recursividad para 2^i nodos es $\lg 2^i + 1 = i + 1$ (ya que para cualquier valor de i , tenemos que $\lg 2^i = i$). Porque estamos asumiendo que el tamaño de entrada original es una potencia de 2, el siguiente tamaño de entrada a considerar es 2^{i+1} . UNA árbol con 2^{i+1} nodos tiene un nivel más que un árbol de 2^i nodos, por lo que el número total de niveles es $(i + 1) + 1 = \lg 2^{i+1} + 1$.

Para calcular el costo total representado por la recurrencia (2.2), simplemente sumamos los costos de todos los niveles. Hay niveles de $\lg n + 1$, cada uno con un costo cn , para un costo total de $cn(\lg n + 1) = cn \lg n + cn$. Ignorar el término de orden inferior y la constante c da el resultado deseado de $\Theta(n \lg n)$.

Ejercicios 2.3-1

Utilizando la Figura 2.4 como modelo, ilustre la operación de ordenación por combinación en la matriz $A = 3, 41, 52, 26, 38, 57, 9, 49$.

Ejercicios 2.3-2

Vuelva a escribir el procedimiento MERGE para que no utilice centinelas, sino que se detenga una vez. Array L o R ha tenido todos sus elementos copiados de nuevo a A y luego copiando el resto de la otra de vuelta matriz en A .

Ejercicios 2.3-3

Utilice la inducción matemática para demostrar que cuando n es una potencia exacta de 2, la solución del reaparición

Página 36

Ejercicios 2.3-4

La ordenación por inserción se puede expresar como un procedimiento recursivo de la siguiente manera. Para clasificar A [1 n], ordenamos recursivamente A [1 n -1] y luego insertamos A [n] en la matriz ordenada A [1 n - 1]. Escribe un recurrencia durante el tiempo de ejecución de esta versión recursiva de ordenación por inserción.

Ejercicios 2.3-5

Volviendo al problema de búsqueda (vea el ejercicio 2.1-3), observe que si la secuencia A es ordenados, podemos comparar el punto medio de la secuencia con vy eliminar la mitad de la secuencia de consideración adicional. La búsqueda binaria es un algoritmo que repite esto procedimiento, reduciendo a la mitad el tamaño de la parte restante de la secuencia cada vez. Escribir pseudocódigo, iterativo o recursivo, para búsqueda binaria. Argumenta que el peor caso de ejecución el tiempo de la búsqueda binaria es Θ (lg n).

Ejercicios 2.3-6

Observe que el **tiempo** de bucle de las líneas 5 - 7 del procedimiento de inserción-ORDENAR en la Sección 2.1 utiliza una búsqueda lineal para escanear (hacia atrás) a través del subarreglo ordenado A [1 j - 1]. ¿Podemos usar un búsqueda binaria (vea el ejercicio 2.3-5) en su lugar para mejorar el tiempo de ejecución general del peor caso de orden de inserción a Θ (n lg n)?

Ejercicios 2.3-7:

Describe un algoritmo de tiempo Θ (n lg n) que, dado un conjunto S de n enteros y otro entero x , determina si existen o no dos elementos en S cuya suma es exactamente x .

Problemas 2-1: ordenación por inserción en matrices pequeñas en ordenación por combinación

Aunque la ordenación por combinación se ejecuta en Θ (n lg n) en el peor de los casos y la ordenación por inserción se ejecuta en Θ (n^2) caso de tiempo, los factores constantes en la ordenación por inserción lo hacen más rápido para n pequeños . Por tanto, tiene sentido para utilizar la ordenación por inserción dentro de la ordenación combinada cuando los subproblemas se vuelven lo suficientemente pequeños. Conside una modificación para fusionar ordenación en la que n / k sublistas de longitud k se ordenan mediante ordenación por inserción

y luego se fusionó utilizando el mecanismo de fusión estándar, donde k es un valor por determinar.

Página 37

a. Muestre que las n / k sublistas, cada una de longitud k , se pueden ordenar por ordenación por inserción en $\Theta (nk)$ momento del peor de los casos.
si. Muestre que las sublistas pueden fusionarse en $\Theta (n \lg (n / k)$ en el peor de los casos.
C. Dado que el algoritmo modificado se ejecuta en $\Theta (nk + n \lg (n / k))$ en el peor de los casos, ¿cuál es el mayor valor asintótico (notación Θ) de k en función de n para el cual el El algoritmo tiene el mismo tiempo de ejecución asintótico que el tipo de combinación estándar?
re. ¿Cómo debería elegirse k en la práctica?

Problemas 2-2: Corrección del tipo de burbujas

Bubblesort es un algoritmo de clasificación popular. Funciona intercambiando repetidamente elementos adyacentes que están fuera de servicio.

```
BUBBLESORT ( A )
1 para i ← 1 a la longitud [ A ]
2 hacer para j ← longitud [ A ] hacia abajo i + 1
3                   hacer si A [ j ] < A [ j - 1]
4                               luego intercambia A [ j ] ↔ A [ j - 1]
```

a. Sea A' la salida de BUBBLESORT (A). Para demostrar que BUBBLESORT es correcto, tenemos que demostrar que termina y que

(2,3)

si. donde $n = longitud [A]$. ¿Qué más debe probarse para demostrar que BUBBLESORT realmente ¿ordena?

Las siguientes dos partes probarán la desigualdad (2.3).

si. Indique con precisión un ciclo invariante para el ciclo **for** en las líneas 2-4, y demuestre que este ciclo invariante sostiene. Su prueba debe usar la estructura de la prueba invariante de bucle presentado en este capítulo.
C. Utilizando la condición de terminación del invariante de bucle demostrado en el inciso b), establezca un bucle invariante para el ciclo **for** en las líneas 1-4 que le permitirá probar la desigualdad (2.3). Su prueba debe usar la estructura de la prueba invariante de bucle presentada en este capítulo.
re. ¿Cuál es el peor tiempo de ejecución de bubblesort? ¿Cómo se compara con el tiempo de ejecución del tipo de inserción?

Problemas 2-3: Corrección de la regla de Horner

El siguiente fragmento de código implementa la regla de Horner para evaluar un polinomio

Página 38

dados los coeficientes a_0 , a_1 ,. . . , una_{my} un valor para x :

```
1 y ← 0
2 i ← n
3 mientras yo ≥ 0
4 hacer y ← a_i + x · y
5                yo ← yo - 1
```

a. ¿Cuál es el tiempo de ejecución asintótico de este fragmento de código para la regla de Horner?

si. Escriba un pseudocódigo para implementar el algoritmo ingenuo de evaluación de polinomios que calcula cada término del polinomio desde cero. ¿Cuál es el tiempo de ejecución de este ¿algoritmo? ¿Cómo se compara con la regla de Horner?

C. Demostrar que la siguiente es una invariante de bucle para el **tiempo** de bucle en las líneas 3 -5.

Al comienzo de cada iteración del **tiempo** de bucle de las líneas 3-5,

Interprete una suma sin términos como igual a 0. Su demostración debe seguir el estructura de la prueba invariante de bucle presentada en este capítulo y debe mostrar que, en terminación, .

re. Concluya argumentando que el fragmento de código dado evalúa correctamente un polinomio caracterizado por los coeficientes a_0 , a_1 ,. . . , un_n .

Problemas 2-4: Inversiones

Sea A [1 n] una matriz de n números distintos. Si $i < j$ y A [i]> A [j], entonces el par (i , j) es llama una inversión de A .

a. Enumere las cinco inversiones de la matriz 2, 3, 8, 6, 1.

si. ¿Qué matriz con elementos del conjunto {1, 2,. . . , n } tiene la mayor cantidad de inversiones? Cómo muchos tiene?

C. ¿Cuál es la relación entre el tiempo de ejecución de la ordenación por inserción y el número de inversiones en la matriz de entrada? Justifica tu respuesta.

re. Dar un algoritmo que determine el número de inversiones en cualquier permutación en n elementos en Θ (n lg n) en el peor de los casos. (*Sugerencia:* modifique el orden de combinación).

[6] Veremos en el Capítulo 3 cómo interpretar formalmente ecuaciones que contienen notación Θ.

Página 39

[7] La expresión ⌈ x ⌉ denota el menor número entero mayor o igual que x , y ⌊ x ⌋ denota el mayor entero menor o igual ax . Estas notaciones se definen en el Capítulo 3 . Lo más fácil forma de verificar que el ajuste de q a ⌊ (p + r) / 2⌋ produce subarreglos A [pq] y A [q + 1 r] de tamaños ⌈ n / 2⌉ y ⌊ n / 2⌋, respectivamente, es examinar los cuatro casos que surgen dependiendo de si cada uno de p y r es par o impar.

[8] Es poco probable que la misma constante represente exactamente tanto el tiempo para resolver problemas de tamaño 1 y el tiempo por elemento de matriz de los pasos de división y combinación. Podemos evitar esto problema al dejar que c sea el mayor de estos tiempos y comprender que nuestra recurrencia da un límite superior en el tiempo de ejecución, o dejando que c sea el menor de estos tiempos y entendiendo que nuestra recurrencia da un límite inferior en el tiempo de ejecución. Ambos límites serán estar en el orden de n lg n y, en conjunto, dar un tiempo de ejecución Θ (n lg n).

Notas del capítulo

En 1968, Knuth publicó el primero de tres volúmenes con el título general *The Art of Computer Programación* [182 , 183 , 185]. El primer volumen marcó el comienzo del estudio moderno de la computadora algoritmos con un enfoque en el análisis del tiempo de ejecución, y la serie completa sigue siendo un referencia interesante y valiosa para muchos de los temas presentados aquí. De acuerdo a Knuth, la palabra "algoritmo" se deriva del nombre "al-Khowârizmî", un Matemático persa.

Aho, Hopcroft y Ullman [5] defendieron el análisis asintótico de algoritmos como un medio de comparar el desempeño relativo. También popularizaron el uso de relaciones de recurrencia para describir los tiempos de ejecución de los algoritmos recursivos.

Knuth [185] proporciona un tratamiento enciclopédico de muchos algoritmos de clasificación. Su comparación de algoritmos de clasificación (página 381) incluye análisis exactos de conteo de pasos, como el que realizado aquí para la ordenación por inserción. La discusión de Knuth sobre el tipo de inserción abarca varios variaciones del algoritmo. El más importante de ellos es el tipo de Shell, introducido por DL Shell, que utiliza la ordenación por inserción en subsecuencias periódicas de la entrada para producir una algoritmo de clasificación.

Knuth también describe la ordenación por combinación. Menciona que un compactador mecánico capaz de fusionar dos mazos de cartas perforadas en una sola pasada se inventó en 1938. J. von Neumann, uno de los pioneros de la informática, aparentemente escribió un programa para la clasificación de Computadora EDVAC en 1945.

Gries [133] describe la historia temprana de demostrar que los programas son correctos , y atribuye a P. Naur con el primer artículo en este campo. Gries atribuye invariantes de bucle a RW Floyd. los El libro de texto de Mitchell [222] describe el progreso más reciente en la demostración de que los programas son correctos.

Capítulo 3: Crecimiento de funciones

Visión general

Página 40

El orden de crecimiento del tiempo de ejecución de un algoritmo, definido en el Capítulo 2 , da una simple caracterización de la eficiencia del algoritmo y también nos permite comparar la relativa ejecución de algoritmos alternativos. Una vez que el tamaño de entrada n sea lo suficientemente grande, combine sort, con su tiempo de ejecución worst ($n \lg n$) en el peor de los casos, supera a la ordenación por inserción, cuyo el tiempo es $\Theta(n^2)$. Aunque a veces podemos determinar el tiempo de ejecución exacto de un algoritmo, como hicimos para la ordenación por inserción en el Capítulo 2 , la precisión adicional no suele merecer el esfuerzo de calcularlo. Para entradas suficientemente grandes, las constantes multiplicativas y los términos de orden inferior de un tiempo de ejecución exacto está dominado por los efectos del tamaño de entrada en sí.

Cuando miramos los tamaños de entrada lo suficientemente grandes como para hacer solo el orden de crecimiento de la ejecución relevante en el tiempo, estamos estudiando la eficiencia **asintótica** de los algoritmos. Es decir, somos preocupado por cómo el tiempo de ejecución de un algoritmo aumenta con el tamaño de la entrada *en el límite* , ya que el tamaño de la entrada aumenta sin límite. Por lo general, un algoritmo que es asintóticamente más eficiente será la mejor opción para todas las entradas excepto para muy pequeñas.

Este capítulo ofrece varios métodos estándar para simplificar el análisis asintótico de algoritmos. La siguiente sección comienza definiendo varios tipos de "notación asintótica", de que ya hemos visto en un ejemplo en notación Θ. Varias convenciones de notación utilizadas A lo largo de este libro se presentan a continuación, y finalmente revisamos el comportamiento de las funciones que surgen comúnmente en el análisis de algoritmos.

3.1 Notación asintótica

Las notaciones que usamos para describir el tiempo de ejecución asintótico de un algoritmo se definen en términos de funciones cuyos dominios son el conjunto de números naturales $\mathbf{N} = \{0, 1, 2, \ldots\}$. Tal Las notaciones son convenientes para describir la función de tiempo de ejecución del peor caso $T(n)$, que es generalmente se define solo en tamaños de entrada enteros. Sin embargo, a veces es conveniente *abusar* notación asintótica de diversas formas. Por ejemplo, la notación se extiende fácilmente al dominio de los números reales o, alternativamente, restringido a un subconjunto de los números naturales. Es Sin embargo, es importante comprender el significado preciso de la notación de modo que cuando sea

abusado, no se *usa indebidamente* . Esta sección define las notaciones asintóticas básicas y también introduce algunos abusos comunes.

Notación Θ

En el Capítulo 2 , encontramos que el peor tiempo de ejecución de ordenación por inserción es $T(n) = \Theta(n^2)$. Dejar definamos lo que significa esta notación. Para una función dada $g(n)$, denotamos por $\Theta(g(n))$ el *conjunto de funciones*

$\Theta(g(n)) = \{f(n)$: existen constantes positivas c_1, c_2 y n_0 tales que $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ para todo $n \ge n_0\}$. [1]

Una función $f(n)$ pertenece al conjunto $\Theta(g(n))$ si existen constantes positivas c_1 y c_2 tales que se puede "emparedar" entre $c_1 g(n)$ y $c_2 g(n)$, para n suficientemente grande . Porque $\Theta(g(n))$ es un conjunto, podríamos escribir "$f(n) \Theta(g(n))$" para indicar que $f(n)$ es un miembro de $\Theta(g(n))$. En cambio, nosotros normalmente escribirá "$f(n) = \Theta(g(n))$" para expresar la misma noción. Este abuso de igualdad para denotar La pertenencia a un conjunto puede parecer confusa al principio, pero veremos más adelante en esta sección que tiene ventajas.

Página 41

La figura 3.1 (a) da una imagen intuitiva de las funciones $f(n)$ y $g(n)$, donde tenemos que $f(n) = \Theta(g(n))$. Para todos los valores de n a la derecha de n_0, el valor de $f(n)$ se encuentra en o por encima de $c_1 g(n)$ y en o por debajo de $c_2 g(n)$. En otras palabras, para todo $n \ge n_0$, la función $f(n)$ es igual a $g(n)$ dentro de un factor constante. Decimos que $g(n)$ es una cota asintóticamente ajustada para $f(n)$.

Figura 3.1: Ejemplos gráficos de las notaciones Θ, *O* y Ω. En cada parte, el valor de n_0 se muestra el valor mínimo posible; cualquier valor mayor también funcionaría. (*a*) notación Θ limita una función dentro de factores constantes. Escribimos $f(n) = \Theta(g(n))$ si existen positivos constantes n_0, c_1 y c_2 tales que a la derecha de n_0, el valor de $f(n)$ siempre se encuentra entre $c_1 g(n)$ y $c_2 g(n)$ inclusive. (*b*) La notación *O* da un límite superior para una función dentro de una constante factor. Escribimos $f(n) = O(g(n))$ si no son constantes positivas n_0 y c de tal manera que a la derecha de n_0, el valor de $f(n)$ siempre se encuentra en o por debajo de $cg(n)$. (*c*) La notación Ω da un límite inferior para un función dentro de un factor constante. Escribimos $f(n) = \Omega(g(n))$ si hay constantes positivas n_0 y c de tal manera que a la derecha del n_0, el valor de $f(n)$ siempre se encuentra sobre o por encima de $cg(n)$.

La definición de $\Theta(g(n))$ requiere que cada miembro $f(n) \Theta(g(n))$ sea asintóticamente no negativo , es decir, que $f(n)$ sea no negativo siempre que n sea suficientemente grande. (Un La función asintóticamente positiva es aquella que es positiva para todos los n suficientemente grandes). En consecuencia, la función $g(n)$ en sí debe ser asintóticamente no negativa, o de lo contrario el conjunto $\Theta(g(n))$ está vacío. Por lo tanto, asumiremos que cada función utilizada dentro de la notación Θ es asintóticamente no negativo. Este supuesto es válido para las otras notaciones asintóticas definidas en este capítulo también.

En el capítulo 2 , presentamos una noción informal de notación Θ que equivalía a desechar términos de orden inferior e ignorando el coeficiente principal del término de orden superior. Nos deja Justifique brevemente esta intuición utilizando la definición formal para demostrar que $1/2\, n^2 - 3n = \Theta(n^2)$. A Para hacerlo, debemos determinar constantes positivas c_1, c_2 y n_0 tales que

$c_1 n^2 \le 1/2\, n^2 - 3n \le c_2 n^2$

para todo $n \ge n_0$. Dividiendo por n^2 se obtiene

$c_1 \le 1/2 - 3/n \le c_2$.

Se puede hacer que la desigualdad de la derecha sea válida para cualquier valor de $n \ge 1$ eligiendo $c_2 \ge 1/2$. Asimismo, se puede hacer que la desigualdad de la izquierda sea válida para cualquier valor de $n \ge 7$ eligiendo $c_1 \le 1/14$. Por lo tanto, al elegir $c_1 = 1/14$, $c_2 = 1/2$ y $n_0 = 7$, podemos verificar que $1/2\, n^2 - 3n = \Theta(n^2)$. Ciertamente, existen otras opciones para las constantes, pero lo importante es que *alguna* opción existe. Tenga en cuenta que estas constantes dependen de la función $1/2\, n^2 - 3n$; una función diferente

pertenecer a $\Theta(n^2)$ normalmente requeriría diferentes constantes.

También podemos usar la definición formal para verificar que $6n^3 \neq \Theta(n^2)$. Supongamos con el propósito de contradicción de que c_2 y n_0 existen tal que $6n^3 \leq c_2 n^2$ para todo $n \geq n_0$. Pero entonces $n \leq c_2/6$, que no puede ser válido para n arbitrariamente grande , ya que c_2 es constante.

Página 42

Intuitivamente, los términos de orden inferior de una función asintóticamente positiva pueden ignorarse en determinar límites asintóticamente estrechos porque son insignificantes para n grandes . Una pequeña fracción del término de orden superior es suficiente para dominar los términos de orden inferior. Por lo tanto, estableciendo c_1 a un valor que es ligeramente más pequeño que el coeficiente del término de orden más alto y ajuste c_2 a un valor ligeramente mayor permite que las desigualdades en la definición de notación Θ sean satisfecho. El coeficiente del término de orden más alto puede igualmente ignorarse, ya que sólo cambia c_1 y c_2 por un factor constante igual al coeficiente.

Como ejemplo, considere cualquier función cuadrática $f(n) = an^2 + bn + c$, donde a , b y c son constantes y $a > 0$. Desechando los términos de orden inferior e ignorando los rendimientos constantes $f(norte) = \Theta(norte^2)$. Formalmente, para mostrar lo mismo, tomamos las constantes $c_1 = a/4$, $c_2 = 7a/4$ y . El lector puede verificar que $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ para todo $n \geq n_0$. En general, para cualquier polinomio , donde a_i son constantes y $a_d > 0$, tenemos $p(n) = \Theta(n^d)$ (vea el problema 3-1).

Dado que cualquier constante es un polinomio de grado 0, podemos expresar cualquier función constante como $\Theta(n^0)$, o $\Theta(1)$. Esta última notación es un abuso menor, sin embargo, porque no está claro qué variable es tendiendo al infinito. [2] A menudo usaremos la notación $\Theta(1)$ para significar una constante o una función constante con respecto a alguna variable.

Notación O

La notación Θ delimita asintóticamente una función desde arriba y desde abajo. Cuando solo tenemos un límite superior asintótico , usamos la notación *O.* Para una función dada $g(n)$, denotamos por $O(g(n))$ (pronunciado "gran-oh de g de n " oa veces simplemente "oh de g de n ") el conjunto de funciones

$O(g(n)) = \{f(n)$: existen constantes positivas c y n_0 tales que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0\}$.

Usamos la notación O para dar un límite superior en una función, dentro de un factor constante. Figura 3.1 (b) muestra la intuición detrás de la notación *O.* Para todos los valores n a la derecha de n_0, el valor de la función $f(n)$ está en o por debajo de $g(n)$.

Escribimos $f(n) = O(g(n))$ para indicar que una función $f(n)$ es miembro del conjunto $O(g(n))$. Nota que $f(n) = \Theta(g(n))$ implica $f(n) = O(g(n))$, ya que la notación Θ es una noción más fuerte que O - notación. Escrito en teoría de conjuntos, tenemos $\Theta(g(n)) \subseteq O(g(n))$. Por tanto, nuestra prueba de que cualquier función cuadrática $an^2 + bn + c$, donde $a > 0$, está en $\Theta(n^2)$ también muestra que cualquier función cuadrática la función está en $O(n^2)$. Lo que puede ser más sorprendente es que cualquier función *lineal* $an + b$ está en $O(n^2)$, que se verifica fácilmente tomando $c = a + |b|$ y $n_0 = 1$.

Algunos lectores que han visto la notación O antes pueden encontrar extraño que debamos escribir, por ejemplo, $n = O(n^2)$. En la literatura, la notación O *a* veces se usa informalmente para describir Límites asintóticamente estrechos, es decir, lo que hemos definido usando la notación Θ. En este libro, sin embargo, cuando escribimos $f(n) = O(g(n))$, simplemente estamos afirmando que algún múltiplo constante de $g(n)$ es un límite superior asintótico en $f(n)$, sin ninguna afirmación sobre qué tan estrecho es el límite superior es. Distinguir los límites superiores asintóticos de los límites asintóticamente estrechos ahora convertirse en estándar en la literatura sobre algoritmos.

Usando la notación O , a menudo podemos describir el tiempo de ejecución de un algoritmo simplemente por inspeccionar la estructura general del algoritmo. Por ejemplo, la estructura de bucle doblemente anidada de

Página 43

el algoritmo de ordenación por inserción del Capítulo 2 produce inmediatamente un límite superior $O(n^2)$ en el tiempo de ejecución en el peor de los casos: el costo de cada iteración del bucle interno está limitado desde arriba por $O(1)$ (constante), los índices i y j son ambos como máximo n, y el ciclo interno se ejecuta como máximo una vez para cada uno de los n^2 pares de valores para i y j.

Dado que la notación O describe un límite superior, cuando lo usamos para delimitar el peor caso de ejecución tiempo de un algoritmo, tenemos un límite en el tiempo de ejecución del algoritmo en cada entrada. Por lo tanto, el límite $O(n^2)$ en el tiempo de ejecución del peor caso de la ordenación por inserción también se aplica a su ejecución tiempo en cada entrada. El límite $\Theta(n^2)$ en el peor tiempo de ejecución de la ordenación por inserción, sin embargo, no implica un límite $\Theta(n^2)$ en el tiempo de ejecución de la ordenación por inserción en *cada* entrada. Por ejemplo, vimos en el Capítulo 2 que cuando la entrada ya está ordenada, la ordenación por inserción se ejecuta en $\Theta(n)$ tiempo.

Técnicamente, es un abuso decir que el tiempo de ejecución de la ordenación por inserción es $O(n^2)$, ya que para un dado n, el tiempo de ejecución real varía, dependiendo de la entrada particular de tamaño n. Cuando nosotros decir "el tiempo de ejecución es $O(n^2)$", queremos decir que hay una función $f(n)$ que es $O(n^2)$ tal que para cualquier valor de n, no importa qué entrada particular de tamaño n se elija, el tiempo de ejecución en ese la entrada está acotada desde arriba por el valor $f(n)$. De manera equivalente, queremos decir que el peor de los casos el tiempo de ejecución es $O(n^2)$.

Notación Ω

Así como la notación O proporciona un límite *superior* asintótico en una función, la notación Ω proporciona una límite inferior asintótico. Para una función dada $g(n)$, denotamos por $\Omega(g(n))$ (pronunciado "grande-omega de g de n "oa veces simplemente" omega de g de n ") el conjunto de funciones

$\Omega(g(n)) = \{f(n)$: existen constantes positivas c y n_0 tal que $0 \le cg(n) \le f(n)$ para todo $n \ge n_0\}$.

La intuición detrás de la notación Ω se muestra en la Figura 3.1 (c). Para todos los valores n a la derecha de n_0, el valor de $f(n)$ es *igual* o superior a $cg(n)$.

De las definiciones de las notaciones asintóticas que hemos visto hasta ahora, es fácil probar la siguiendo un importante teorema (vea el ejercicio 3.1-5).

Teorema 3.1

Para dos funciones cualesquiera $f(n)$ y $g(n)$, tenemos $f(n) = \Theta(g(n))$ si y solo si $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$.

Como ejemplo de la aplicación de este teorema, nuestra prueba de que $an^2 + bn + c = \Theta(n^2)$ para cualquier constantes de un, b, y c, donde $una > 0$, implica inmediatamente que $un^2 + bn + c = \Omega(n^2)$ y $un^2 + bn + c = O(n^2)$. En la práctica, en lugar de usar el teorema 3.1 para obtener asintóticos superior e inferior límites de límites asintóticamente estrechos, como hicimos para este ejemplo, generalmente lo usamos para demostrar límites asintóticamente estrechos desde límites superiores e inferiores asintóticos.

Página 44

Dado que la notación Ω describe un límite inferior, cuando lo usamos para acotar el mejor caso de ejecución tiempo de un algoritmo, por implicación también limitamos el tiempo de ejecución del algoritmo en entradas arbitrarias también. Por ejemplo, el mejor tiempo de ejecución de la ordenación por inserción es $\Omega(n)$, lo que implica que el tiempo de ejecución de la ordenación por inserción es $\Omega(n)$.

Por tanto, el tiempo de ejecución de la ordenación por inserción cae entre $\Omega(n)$ y $O(n^2)$, ya que cae en cualquier lugar entre una función lineal de n y una función cuadrática de n. Además, estos los límites son asintóticamente lo más ajustados posible: por ejemplo, el tiempo de ejecución de la ordenación de inserción no es $\Omega(n^2)$, ya que existe una entrada para la cual la ordenación de inserción se ejecuta en $\Theta(n)$ tiempo (por ejemplo, cuando la entrada ya está ordenada). Sin embargo, no es contradictorio,, decir que el *peor de los casos* corriendo el tiempo de la ordenación de inserción es $\Omega(n^2)$, ya que existe una entrada que hace que el algoritmo tome $\Omega(n^2)$ tiempo. Cuando decimos que el *tiempo de ejecución* (sin modificador) de un algoritmo es $\Omega(g(n))$, significa que *no importa qué entrada particular de tamaño n se elija para cada valor de n* , la ejecución el tiempo en esa entrada es al menos una constante multiplicada por $g(n)$, para n suficientemente grande .

Notación asintótica en ecuaciones y desigualdades

Ya hemos visto cómo se puede utilizar la notación asintótica dentro de fórmulas matemáticas. Por ejemplo, al introducir la notación O, escribimos "$n = O(n^2)$". También podríamos escribir $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. ¿Cómo interpretamos tales fórmulas?

Cuando la notación asintótica está sola en el lado derecho de una ecuación (o desigualdad), como en $n = O(n^2)$, ya hemos definido el signo igual para significar la pertenencia al conjunto: $n \in O(n^2)$. Sin embargo, en general, cuando aparece notación asintótica en una fórmula, interpretamos que representa una función anónima que no nos importa nombrar. Por ejemplo, el fórmula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ significa que $2n^2 + 3n + 1 = 2n^2 + f(n)$, donde $f(n)$ es algo función en el conjunto $\Theta(n)$. En este caso, $f(n) = 3n + 1$, que de hecho está en $\Theta(n)$.

El uso de la notación asintótica de esta manera puede ayudar a eliminar los detalles no esenciales y el desorden en un ecuación. Por ejemplo, en el Capítulo 2 expresamos el peor tiempo de ejecución del tipo de combinación como la recurrencia

$$T(n) = 2T(n/2) + \Theta(n).$$

Si sólo nos interesa el comportamiento asintótico de $T(n)$, no tiene sentido especificar todos los términos de orden inferior exactamente; todos se entienden incluidos en el anónimo función denotada por el término $\Theta(n)$.

Se entiende que el número de funciones anónimas en una expresión es igual al número muchas veces aparece la notación asintótica. Por ejemplo, en la expresión

solo hay una función anónima (una función de i). Por tanto, esta expresión *no es* la lo mismo que $O(1) + O(2) + \ldots + O(n)$, que realmente no tiene una interpretación limpia.

En algunos casos, la notación asintótica aparece en el lado izquierdo de una ecuación, como en

$$2n^2 + \Theta(n) = \Theta(n^2).$$

Página 45

Interpretamos tales ecuaciones usando la siguiente regla: *no importa cómo funcionan las funciones anónimas se eligen a la izquierda del signo igual, hay una manera de elegir las funciones anónimas en a la derecha del signo igual para que la ecuación sea válida*. Por tanto, el significado de nuestro ejemplo es que para *cualquier* función $f(n) \in \Theta(n)$, hay *alguna* función $g(n) \in \Theta(n^2)$ tal que $2n^2 + f(n) = g(n)$ para todos los n. En otras palabras, el lado derecho de una ecuación proporciona un nivel más grueso de detalle que el lado izquierdo.

Varias de estas relaciones se pueden encadenar juntas, como en

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2). \end{aligned}$$

Podemos interpretar cada ecuación por separado mediante la regla anterior. La primera ecuación dice que hay es *alguna* función $f(n) \in \Theta(n)$ tal que $2n^2 + 3n + 1 = 2n^2 + f(n)$ para todo n. La segunda ecuación dice que para *cualquier* función $g(n) \in \Theta(n)$ (como la $f(n)$ que acabo de mencionar), hay *alguna* función $h(n) \in \Theta(n^2)$ tal que $2n^2 + g(n) = h(n)$ para todo n. Tenga en cuenta que esta interpretación implica que $2n^2 + 3n + 1 = \Theta(n^2)$, que es lo que nos da intuitivamente el encadenamiento de ecuaciones.

o- notación

El límite superior asintótico proporcionado por la notación O puede ser asintóticamente apretado o no. El límite $2n^2 = O(n^2)$ es asintóticamente apretado, pero el límite $2n = O(n^2)$ no lo es. Usamos o - notación para denotar un límite superior que no es asintóticamente apretado. Definimos formalmente $o(g(n))$ ("pequeño-oh de g de n") como el conjunto

$o(g(n)) = \{f(n)$: para cualquier constante positiva $c > 0$, existe una constante $n_0 > 0$ tal que $0 \le f(n) < cg(n)$ para todo $n \ge n_0\}$.

Por ejemplo, $2n = o(n^2)$, pero $2n^2 \neq o(n^2)$.

Las definiciones de notación O y notación O son similares. La principal diferencia es que en $f(n) = O(g(n))$, el límite $0 \leq f(n) \leq cg(n)$ se cumple para *alguna* constante $c > 0$, pero en $f(n) = o(g(n))$, el límite $0 \leq f(n) < cg(n)$ se cumple para *todas las* constantes $c > 0$. Intuitivamente, en la notación o , la función $f(n)$ se vuelve insignificante en relación con $g(n)$ cuando n *se* acerca al infinito; es decir,

(3,1)

Algunos autores utilizan este límite como definición de la notación o ; la definición en este libro también restringe las funciones anónimas para que sean asintóticamente no negativas.

ω -notación

Por analogía, la notación ω es para la notación Ω como la notación o es para la notación O. Usamos la notación ω para denota un límite inferior que no es asintóticamente apretado. Una forma de definirlo es por

$f(n) \omega(g(n))$ si y solo si $g(n) o(f(n))$.

Formalmente, sin embargo, definimos $\omega(g(n))$ ("omega pequeño de g de n ") como el conjunto

Página 46

$\omega(g(n)) = \{f(n)$: para cualquier constante positiva $c > 0$, existe una constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$ para todo $n \geq n_0\}$.

Por ejemplo, $n^2/2 = \omega(n)$, pero $n^2/2 \neq \omega(n^2)$. La relación $f(n) = \omega(g(n))$ implica que

si existe el límite. Es decir, $f(n)$ se vuelve arbitrariamente grande en relación con $g(n)$ cuando n *se* acerca infinito.

Comparación de funciones

Muchas de las propiedades relacionales de los números reales se aplican también a las comparaciones asintóticas. Para lo siguiente, suponga que $f(n)$ *y* $g(n)$ son asintóticamente positivos.

Transitividad:

- $f(n) = \Theta(g(n))$ y $g(n) = \Theta(h(n))$ implican $f(n) = \Theta(h(n))$,
  $f(n) = O(g(n))$ y $g(n) = O(h(n))$ implican $f(n) = O(h(n))$,
  $f(n) = \Omega(g(n))$ y $g(n) = \Omega(h(n))$ implican $f(n) = \Omega(h(n))$,
  $f(n) = o(g(n))$ y $g(n) = o(h(n))$ implican $f(n) = o(h(n))$,
  $f(n) = \omega(g(n))$ y $g(n) = \omega(h(n))$ implican $f(n) = \omega(h(n))$.

Reflexividad:

- $f(n) = \Theta(f(n))$,
  $f(n) = O(f(n))$,
  $f(n) = \Omega(f(n))$.

Simetría:

$f(n) = \Theta(g(n))$ si y solo si $g(n) = \Theta(f(n))$.

Transponer simetría:

-

$f(n) = O(g(n))$ si y solo si $g(n) = \Omega(f(n))$,
$f(n) = o(g(n))$ si y solo si $g(n) = \omega(f(n))$.

Dado que estas propiedades son válidas para las notaciones asintóticas, se puede establecer una analogía entre las comparación asintótica de dos funciones f y gy la comparación de dos números reales a y b :

Página 47

$f(n) = O(g(n)) \approx a \leq b$,
$f(n) = \Omega(g(n)) \approx a \geq b$,
$f(n) = \Theta(g(n)) \approx a = segundo$,
$f(n) = o(g(n)) \approx a < b$,
$f(norte) = \omega(sol(n)) \approx a > segundo$.

Decimos que $f(n)$ es asintóticamente menor que $g(n)$ si $f(n) = o(g(n))$, $yf(n)$ es asintóticamente mayor que $g(n)$ si $f(n) = \omega(g(n))$.

Sin embargo, una propiedad de los números reales no se traslada a la notación asintótica:

- **Tricotomía:** Para cualquier par de números reales a y b , exactamente uno de los siguientes obligada mantener: $a < b$, $a = b$, o $a > b$.

Aunque se pueden comparar dos números reales, no todas las funciones son asintóticamente comparable. Es decir, para dos funciones $f(n)$ y $g(n)$, puede darse el caso de que ni $f(n) = O(g(n))$ ni $f(n) = \Omega(g(n))$ se cumple. Por ejemplo, las funciones n y $n^{1+\sin n}$ no se pueden comparar utilizando notación asintótica, ya que el valor del exponente en $n^{1+\sin n}$ oscila entre 0 y 2, asumiendo todos los valores intermedios.

Ejercicios 3.1-1

Sean $f(n)$ $yg(n)$ funciones asintóticamente no negativas. Usando la definición básica de Θ-notación, demuestre que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Ejercicios 3.1-2

Demostrar que para cualquier constantes reales *de una* y b , donde $b > 0$,

(3,2)

Ejercicios 3.1-3

Explique por qué el enunciado "El tiempo de ejecución del algoritmo A es al menos $O(n^2)$" es sin sentido.

Ejercicios 3.1-4

¿Es $2^{n+1} = O(2^n)$? ¿Es $2^{2n} = O(2^n)$?

Página 48

Ejercicios 3.1-5

Demuestre el teorema 3.1 .

Ejercicios 3.1-6

Demuestre que el tiempo de ejecución de un algoritmo es $\Theta(g(n))$ si y solo si se ejecuta en el peor de los casos el tiempo es $O(g(n))$ y su mejor tiempo de ejecución es $\Omega(g(n))$.

Ejercicios 3.1-7

Demuestre que $o(g(n)) \cap \omega(g(n))$ es el conjunto vacío.

Ejercicios 3.1-8

Podemos extender nuestra notación para el caso de dos parámetros n y m que puede ir hasta el infinito de forma independiente a diferentes ritmos. Para una función dada $g(n, m)$, denotamos por $O(g(n, m))$ el conjunto de funciones

$O(g(n, m)) = \{f(n, m)$: existen constantes positivas c, n_0 y m_0 tales que $0 \le f(n, m) \le cg(n, m)$ para todo $n \ge n_0$ y $m \ge m_0\}$.

Proporcione las definiciones correspondientes para $\Omega(g(n, m))$ y $\Theta(g(n, m))$.

[1] Dentro de la notación de conjuntos, los dos puntos deben leerse como "tal que".

[2] El problema real es que nuestra notación ordinaria para funciones no distingue funciones de los valores. En el cálculo λ, los parámetros de una función están claramente especificados: la función n^2 podría escribirse como $\lambda n.n^2$, o incluso $\lambda r.r^2$. Adoptar una notación más rigurosa, sin embargo, complicar las manipulaciones algebraicas, por lo que optamos por tolerar el abuso.

3.2 Notaciones estándar y funciones comunes

Esta sección revisa algunas funciones y notaciones matemáticas estándar y explora las relaciones entre ellos. También ilustra el uso de las notaciones asintóticas.

Página 49

Monotonicidad

Una función $f(n)$ aumenta monótonamente si $m \le n$ implica $f(m) \le f(n)$. Del mismo modo, es monótonamente decreciente si $m \le n$ implica $f(m) \ge f(n)$. Una función $f(n)$ es estrictamente creciente si $m < n$ implica $f(m) < f(n)$ y estrictamente decreciente si $m < n$ implica $f(m) > f(n)$.

Pisos y techos

Para cualquier número real x, denotamos el mayor entero menor o igual ax por $\lfloor x \rfloor$ (lea "el piso de x ") y el menor entero mayor o igual que x por $\lceil x \rceil$ (lea" el techo de x ").
todo real x,

(3,3)

Para cualquier número entero n ,

$\lceil n / 2 \rceil + \lfloor n / 2 \rfloor = n$,

y para cualquier número real $n \geq 0$ y enteros a , $b > 0$,

(3.4)
(3,5)
(3,6)
(3,7)

La función de piso $f(x) = \lfloor x \rfloor$ aumenta monótonamente, al igual que la función de techo $f(x) = \lceil x \rceil$.

Aritmética modular

Para cualquier entero a y cualquier entero positivo n , el valor a mod n es el resto (o residuo) del cociente a / n :

(3,8)

Dada una noción bien definida del resto de un número entero cuando se divide por otro, es conveniente proporcionar una notación especial para indicar la igualdad de los residuos. Si (a mod n) = (b mod n), se escribe $un \equiv b$ (mod n) y decir que *una* es equivalente a B , módulo n . En otras palabras, $un \equiv b$ (mod n) si *una* y b tienen el mismo resto cuando se divide por n . De manera equivalente, $a \equiv b$ (mod n) si y solo si n es un divisor de $b - a$. Escribimos $un \not\equiv b$ (mod n) si *una* no es equivalente a B , módulo n .

Polinomios

Dado un número entero no negativo d , un polinomio en n de grado d es una función $p(n)$ de la forma

donde las constantes a_0 , a_1 , ..., a_d son los coeficientes del polinomio y $a_d \neq 0$. A polinomio es asintóticamente positivo si y solo si $a_d > 0$. Para un asintóticamente positivo polinomio $p(n)$ de grado d , tenemos $p(n) = \Theta(n^d)$. Para cualquier constante real $a \geq 0$, la función n^a es monótonamente creciente, y para cualquier constante real $a \geq 0$, la función n^a es monótonamente decreciente. Decimos que una función $f(n)$ está acotada polinomialmente si $f(n) = O(n^k)$ para alguna constante k .

Exponenciales

Para todos los bienes *de una* > 0, m y n , tenemos las siguientes identidades:

$a^0 = 1$,
$a^1 = a$,
$a^{-1} = 1 / a$,
$(un^m)^n = un^{mn}$,
$(una^m)^n = (una^{norte})^m$,
$una^m una^n = una^{m+n}$.

Para todos los n y $un \geq 1$, la función de un^n está aumentando monótonamente en n . Cuando sea conveniente, asumirá $0^0 = 1$.

Las tasas de crecimiento de polinomios y exponenciales se pueden relacionar mediante el siguiente hecho. por Todas las constantes reales *de una* y b tal que *una* > 1,

(3,9)

de lo cual podemos concluir que

$n^{segundo} = o(un^{norte})$.

Así, cualquier función exponencial con una base estrictamente mayor que 1 crece más rápido que cualquier función polinómica.

Usando e para denotar 2.71828 ..., la base de la función logaritmo natural, tenemos para todo x real ,

(3,10)

dónde "!" denota la función factorial definida más adelante en esta sección. Para todo x real , tenemos el desigualdad

(3,11)

donde la igualdad se mantiene solo cuando $x = 0$. Cuando $|x| \leq 1$, tenemos la aproximación

(3,12)

Página 51

Cuando $x \to 0$, la aproximación de e^x por $1 + x$ es bastante buena:

$e^x = 1 + x + \Theta(x^2)$.

(En esta ecuación, la notación asintótica se usa para describir el comportamiento limitante como $x \to 0$ en lugar de como $x \to \infty$.) Tenemos para todo x ,

(3,13)

Logaritmos

Usaremos las siguientes notaciones:

$\lg n = \log_2 n$ (logaritmo binario),
$\ln n = \log_e n$ (logaritmo natural),
$\lg^k n = (\lg n)^k$ (exponenciación),
$\lg \lg n = \lg(\lg n)$ (composición).

Una importante convención de notación que adoptaremos es que las *funciones de logaritmo se aplicarán solo al siguiente término de la fórmula* , de modo que $\lg n + k$ significará $(\lg n) + k$ y no $\lg(n + k)$. Si nosotros mantenga $b > 1$ constante, entonces para $n > 0$, la función $\log_b n$ es estrictamente creciente.

Para todo real $a > 0$, $b > 0$, $c > 0$ y n ,

(3,14)

(3,15)

donde, en cada ecuación anterior, las bases de los logaritmos no son 1.

Por la ecuación (3.14), cambiar la base de un logaritmo de una constante a otra solamente cambia el valor del logaritmo por un factor constante, por lo que a menudo usaremos la notación "lg n " cuando no nos importan los factores constantes, como en la notación O. Científicos de la computación encuentra 2 como la base más natural para los logaritmos porque hay tantos algoritmos y datos Las estructuras implican dividir un problema en dos partes.

Hay una expansión en serie simple para ln $(1 + x)$ cuando $|x| < 1$:

Página 52

También tenemos las siguientes desigualdades para $x > -1$:

(3,16)

donde la igualdad es válida solo para $x = 0$.

Decimos que una función $f(n)$ está acotada polilogarítmicamente si $f(n) = O(\lg^k n)$ para algunos constante k . Podemos relacionar el crecimiento de polinomios y polilogaritmos sustituyendo lg n para n y 2^a para a en la ecuación (3.9) , dando

A partir de este límite, podemos concluir que

$\lg^b n = o(n^a)$

para cualquier constante $a > 0$. Por lo tanto, cualquier función polinomial positiva crece más rápido que cualquier función polilogarítmica.

Factoriales

La notación $n!$ (lea " n factorial") se define para enteros $n \geq 0$ como

Por lo tanto, $n! = 1 \cdot 2 \cdot 3 \, n$.

¡Un límite superior débil en la función factorial es $n! \leq n^n$, ya que cada uno de los n términos en el el producto factorial es como máximo n . Aproximación de Stirling ,

(3,17)

donde e es la base del logaritmo natural, nos da un límite superior más ajustado y un límite inferior también. Uno puede probar (vea el ejercicio 3.2-3)

(3,18)

donde la aproximación de Stirling es útil para probar la ecuación (3.18) . La siguiente ecuación también es válido para todo $n \geq 1$:

(3,19)

Página 53

dónde

(3,20)

Iteración funcional

Usamos la notación $f^{(i)}(n)$ para denotar la función $f(n)$ aplicada iterativamente i veces a una inicial valor de n . Formalmente, sea $f(n)$ una función sobre los reales. Para enteros no negativos i , definir recursivamente

Por ejemplo, si $f(n) = 2n$, entonces $f^{(i)}(n) = 2^i n$.

La función de logaritmo iterado

Usamos la notación $\lg^* n$ (lea "log estrella de n ") para denotar el logaritmo iterado, que es definido como sigue. Sea $\lg^{(i)} n$ como se definió anteriormente, con $f(n) = \lg n$. Porque el logaritmo de un número no positivo no está definido, $\lg^{(i)} n$ se define solo si $\lg^{(i-1)} n > 0$. Asegúrese de distinguir $\lg^{(i)} n$ (la función logarítmica aplicada i veces en sucesión, comenzando con el argumento n) de $\lg^i n$ (el logaritmo de n elevado a la i- ésima potencia). La función de logaritmo iterado se define como

$\lg^* n = \min \{ i = 0: \lg^{(i)} n \leq 1 \}$.

El logaritmo iterado es una función de crecimiento *muy* lento:

$\lg^* 2 = 1$,
$\lg^* 4 = 2$,
$\lg^* 16 = 3$,
$\lg^* 65536 = 4$,
$\lg^* (2^{65536}) = 5$.

Dado que se estima que el número de átomos en el universo observable es de alrededor de 10^{80} , que es mucho menos de 2^{65536} , rara vez encontramos un tamaño de entrada n tal que $\lg^* n > 5$.

Números de Fibonacci

Los **números de Fibonacci** se definen por la siguiente recurrencia:

(3,21)

Por lo tanto, cada número de Fibonacci es la suma de los dos anteriores, lo que produce la secuencia

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Página 54

Los números de Fibonacci están relacionados con la **proporción áurea** φ y su conjugado, que están dados por las siguientes fórmulas:

(3,22)

Específicamente, tenemos

(3,23)

que puede demostrarse por inducción (ejercicio 3.2-6). Ya que , tenemos ,
de modo que el *i*- ésimo número de Fibonacci F_i es igual a redondeado al número entero más cercano. Así,
Los números de Fibonacci crecen exponencialmente.

Ejercicios 3.2-1

Demuestre que si $f(n)$ y $g(n)$ son funciones monótonamente crecientes, entonces también lo son las funciones $f(n) + g(n)$ y $f(g(n))$, y si $f(n)$ y $g(n)$ son además no negativos, entonces $f(n) \cdot g(n)$ es aumentando monótonamente.

Ejercicios 3.2-2

Demuestre la ecuación (3.15) .

Ejercicios 3.2-3

Demuestre la ecuación (3.18) . También demuestre que $n! = \omega(2^n)$ y $n! = o(n^n)$.

Ejercicios 3.2-4:

¿Es la función $\lceil \lg n \rceil!$ acotado polinomialmente? ¿Es la función $\lceil \lg \lg n \rceil!$ polinomialmente ¿encerrado?

Página 55

Ejercicios 3.2-5:

¿Cuál es asintóticamente más grande: $\lg(\lg^* n)$ o $\lg^*(\lg n)$?

Ejercicios 3.2-6

Demuestre por inducción que el *i*- ésimo número de Fibonacci satisface la igualdad

donde φ es la proporción áurea y es su conjugado.

Ejercicios 3.2-7

Demuestre que para $i \geq 0$, el $(i + 2)$ nd número de Fibonacci satisface $F_{i+2} \geq \varphi^i$.

Problemas 3-1: Comportamiento asintótico de polinomios

Dejar

donde $a_d > 0$, sea un polinomio de grado d en n , y sea k una constante. Utilice las definiciones de notaciones asintóticas para probar las siguientes propiedades.

a. Si $k \geq d$, entonces $p(n) = O(n^k)$.
si. Si $k \leq d$, entonces $p(n) = \Omega(n^k)$.
C. Si $k = d$, entonces $p(n) = \Theta(n^k)$.
re. Si $k > d$, entonces $p(n) = o(n^k)$.
mi. Si $k < d$, entonces $p(n) = \omega(n^k)$.

Problemas 3-2: crecimientos asintóticos relativos

Página 56

Indique, para cada par de expresiones (A , B) en la siguiente tabla, si A es O , o , Ω, ω o Θ de B . Suponga que $k \geq 1$,> 0 $yc > 1$ son constantes. Tu respuesta debe tener la forma de la tabla con "sí" o "no" escrito en cada casilla.

	UNA	BO	o	Ω	ω	Θ
a.	$\lg^k n$	n				
si.	n^k	c^n				
C.	n	pecado n				
re.	2^n	$2^{n/2}$				
mi.	$n^{\lg c}$	$c^{\lg n}$				
F.	$\lg(n!)$	$\lg(n^n)$				

Problemas 3-3: ordenación por tasas de crecimiento asintóticas

a. Clasifique las siguientes funciones por orden de crecimiento; es decir, encuentre un arreglo g_1 , g_2 , ..., g_{30} de las funciones que satisfacen $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, ..., $g_{29} = \Omega(g_{30})$. Particione su listar en clases de equivalencia de modo que $f(n)$ y $g(n)$ estén en la misma clase si y solo si $f(n) = \Theta(g(n))$.

si. Dé un ejemplo de una única función no negativa $f(n)$ tal que para todas las funciones $g_i(n)$ en la parte (a), $f(n)$ no es ni $O(g_i(n))$ ni $\Omega(g_i(n))$.

Problemas 3-4: propiedades de notación asintótica

Sean $f(n)$ y $g(n)$ funciones asintóticamente positivas. Demuestre o refute cada uno de los siguientes conjeturas.

a. $f(n) = O(g(n))$ implica $g(n) = O(f(n))$.
si. $f(n) + g(n) = \Theta(\text{mínimo}(f(n), g(n)))$.
C. $f(n) = O(g(n))$ implica $\lg(f(n)) = O(\lg(g(n)))$, donde $\lg(g(n)) \geq 1$ y $f(n) \geq 1$ para todas suficientemente grande n .
re. $f(n) = O(g(n))$ implica $2^{f(n)} = O(2^{g(n)})$.
mi. $f(n) = O((f(n))^2)$.
F. $f(n) = O(g(n))$ implica $g(n) = \Omega(f(n))$.

Página 57

gramo. $f(n) = \Theta(f(n/2))$.
h. $f(n) + o(f(n)) = \Theta(f(n))$.

Problemas 3-5: Variaciones en O y Ω

Algunos autores definen Ω de una manera ligeramente diferente a la nuestra; usemos (lea "omega infinito ") para esta definición alternativa. Decimos que si existe un positivo constante c tal que $f(n) \geq cg(n) \geq 0$ para infinitos números enteros n .

a. Demuestre que para dos funciones cualesquiera $f(n)$ y $g(n)$ que son asintóticamente no negativas, ya sea $f(n) = O(g(n))$ o o ambos, mientras que esto no es cierto si usamos Ω en lugar de .
si. Describa las posibles ventajas y desventajas de usar en lugar de Ω para caracterizar los tiempos de ejecución de los programas.

Algunos autores también definen O de una manera ligeramente diferente; usemos O' para la alternativa definición. Decimos que $f(n) = O'(g(n))$ si y solo si $|f(n)| = O(g(n))$.

C. ¿Qué sucede con cada dirección del "si y solo si" en el Teorema 3.1 si sustituimos O' para O pero todavía usa Ω?

Algunos autores definen $\tilde{O}$ (lea "soft-oh") como O con factores logarítmicos ignorados:

$\tilde{O}(g(n)) = \{f(n)$: existen constantes positivas c , k , y n_0 tales que $0 \leq f(n) \leq cg(n) \lg^k(n)$ para todos $n \geq n_0\}$.

re. Definir y de manera similar. Demuestre el análogo correspondiente al teorema 3.1 .

Problemas 3-6: funciones iteradas

El operador de iteración * utilizado en la función lg * se puede aplicar a cualquier función creciente $f(n)$ sobre los reales. Para una constante c **R** dada , definimos el iterado función por

que no necesitan estar bien definidos en todos los casos. En otras palabras, la cantidad es el número de aplicaciones iteradas de la función f necesarias para reducir su argumento a c o menos.

Para cada una de las siguientes funciones $f(n)$ y constantes c , dé un límite lo más ajustado posible en .

f (n) c

Página 58

	f (n) c
a.	*n* - 1 0
si.	lg *n* 1
C.	*n* / 2 1
re.	*n* / 2 2
mi.	2
F.	1
gramo.	$n^{1/2}$
h.	*n* / lg *n* 2

Notas del capítulo

Knuth [182] rastrea el origen de la notación *O* en un texto de teoría de números de P. Bachmann en 1892. La notación *o* fue inventada por E. Landau en 1909 para su análisis de la distribución de números primos. Las notaciones Ω y Θ fueron recomendadas por Knuth [186] para corregir la práctica popular, pero técnicamente descuidada, en la literatura de usar la notación *O* para ambos y límites inferiores. Mucha gente continúa usando la notación *O* donde la notación Θ es más técnicamente precisa. Discusión adicional de la historia y el desarrollo de las notaciones asintóticas. se puede encontrar en Knuth [182 , 186] y Brassard y Bratley [46] .

No todos los autores definen las notaciones asintóticas de la misma manera, aunque las diversas las definiciones coinciden en las situaciones más comunes. Algunas de las definiciones alternativas abarcan funciones que no son asintóticamente no negativas, siempre que sus valores absolutos sean apropiadamente delimitado.

La ecuación (3.19) se debe a Robbins [260] . Otras propiedades de la matemática elemental Las funciones se pueden encontrar en cualquier buena referencia matemática, como Abramowitz y Stegun. [1] o Zwillinger [320] , o en un libro de cálculo, como Apostol [18] o Thomas y Finney [296] . Knuth [182] y Graham, Knuth y Patashnik [132] contienen una gran cantidad de material sobre Matemáticas discretas como se usa en informática.

Tecnicismos

En la práctica, descuidamos ciertos detalles técnicos cuando enunciamos y resolvemos recurrencias. Un bien ejemplo de un detalle que a menudo se pasa por alto es la suposición de argumentos enteros para funciones. Normalmente, el tiempo de ejecución $T(n)$ de un algoritmo solo se define cuando n es un entero, ya que para la mayoría de los algoritmos, el tamaño de la entrada es siempre un número entero. Por ejemplo, el La recurrencia que describe el peor tiempo de ejecución de MERGE-SORT es realmente

(4,2)

Las condiciones de contorno representan otra clase de detalles que normalmente ignoramos. Desde el El tiempo de ejecución de un algoritmo en una entrada de tamaño constante es una constante, las recurrencias que surgen de los tiempos de ejecución de los algoritmos generalmente tienen $T(n) = \Theta(1)$ para n suficientemente pequeño . En consecuencia, por conveniencia, generalmente omitiremos las declaraciones de las condiciones de contorno

Página 59

de recurrencias y suponga que $T(n)$ es constante para n pequeño . Por ejemplo, normalmente decimos recurrencia (4.1) como

(4,3)

sin dar explícitamente valores para n pequeño . La razón es que aunque cambiando el valor de $T(1)$ cambia la solución a la recurrencia, la solución normalmente no cambia en más

que un factor constante, por lo que el orden de crecimiento no cambia.

Cuando declaramos y resolvemos recurrencias, a menudo omitimos pisos, techos y condiciones de contorno. Seguimos adelante sin estos detalles y luego determinamos si importan o no. Ellos normalmente no lo hacen, pero es importante saber cuándo lo hacen. La experiencia ayuda, y también algunos teoremas que establecen que estos detalles no afectan los límites asintóticos de muchas recurrencias encontrado en el análisis de algoritmos (ver Teorema 4.1). En este capítulo, sin embargo, abordará algunos de estos detalles para mostrar los detalles de los métodos de solución de recurrencia.

4.1 El método de sustitución

El método de sustitución para resolver recurrencias comprende dos pasos:

1. Adivina la forma de la solución.
2. Use la inducción matemática para encontrar las constantes y demuestre que la solución funciona.

El nombre proviene de la sustitución de la respuesta adivinada por la función cuando el La hipótesis inductiva se aplica a valores menores. Este método es poderoso, pero obviamente sólo se puede aplicar en los casos en que sea fácil adivinar la forma de la respuesta.

El método de sustitución se puede utilizar para establecer límites superiores o inferiores en un reaparición. Como ejemplo, determinemos un límite superior en la recurrencia

(4,4)

que es similar a las recurrencias (4.2) y (4.3). Suponemos que la solución es $T(n) = O(n \lg n)$. Nuestro método consiste en demostrar que $T(n) \le cn \lg n$ para una elección adecuada de la constante $c > 0$.

comience asumiendo que este límite se cumple para $\lfloor n/2 \rfloor$, es decir, que $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Sustituyendo en los rendimientos de recurrencia

$$
\begin{aligned}
T(n) &\le 2(do \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\
&\le cn \lg(n/2) + n \\
&= cn \lg n - cn \lg 2 + n \\
&= cn \lg n - cn + n \\
&\le cn \lg n ,
\end{aligned}
$$

donde el último paso se mantiene mientras $c \ge 1$.

La inducción matemática ahora requiere que demostremos que nuestra solución es válida para el límite condiciones. Por lo general, lo hacemos mostrando que las condiciones de contorno son adecuadas como base casos para la prueba inductiva. Para la recurrencia (4.4), debemos demostrar que podemos elegir el

Página 60

constante c lo suficientemente grande para que el límite $T(n) = cn \lg n$ funcione para las condiciones de contorno como bien. Este requisito a veces puede dar lugar a problemas. Asumamos, por el bien de argumento, que $T(1) = 1$ es la única condición de frontera de la recurrencia. Entonces para $n = 1$, el ligado $T(n) = cn \lg n$ produce $T(1) = c\,1 \lg 1 = 0$, que está en desacuerdo con $T(1) = 1$. En consecuencia, el caso base de nuestra prueba inductiva no se sostiene.

Esta dificultad para probar una hipótesis inductiva para una condición de contorno específica puede ser superar fácilmente. Por ejemplo, en la recurrencia (4.4), aprovechamos la asintótica notación que solo requiere que demostremos $T(n) = cn \lg n$ para $n \ge n_0$, donde n_0 es una constante de nuestro elegir. La idea es eliminar la difícil condición de contorno $T(1) = 1$ de consideración en la prueba inductiva. Observe que para $n > 3$, la recurrencia no depende directamente de $T(1)$. Por lo tanto, podemos reemplazar $T(1)$ por $T(2)$ y $T(3)$ como los casos base en la prueba inductiva, dejando $n_0 = 2$. Tenga en cuenta que hacemos una distinción entre el caso base de la recurrencia ($n = 1$) y los casos base de la prueba inductiva ($n = 2$ y $n = 3$). Derivamos de la recurrencia que $T(2) = 4$ y $T(3) = 5$. La prueba inductiva de que $T(n) \le cn \lg n$ para alguna constante $c \ge 1$ puede ahora se completa eligiendo c lo suficientemente grande para que $T(2) \le c\,2 \lg 2$ y $T(3) \le c\,3 \lg 3$. Como resulta que cualquier elección de $c \ge 2$ es suficiente para que se mantengan los casos base de $n = 2$ y $n = 3$. Para la mayoría de las recurrencias que examinaremos, es sencillo extender las condiciones de frontera a hacer que el supuesto inductivo funcione para n pequeños .

Haciendo una buena suposición

Desafortunadamente, no existe una forma general de adivinar las soluciones correctas para las recurrencias. Adivinación una solución requiere experiencia y, ocasionalmente, creatividad. Afortunadamente, sin embargo, hay algunos heurísticas que pueden ayudarlo a convertirse en un buen adivino. También puede utilizar árboles de recursividad, que veremos en la Sección 4.2 , para generar buenas suposiciones.

Si una recurrencia es similar a una que ha visto antes, entonces adivinar una solución similar es razonable. Como ejemplo, considere la recurrencia

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n,$$

lo cual parece difícil debido al "17" agregado en el argumento de T en el lado derecho.
Sin embargo, intuitivamente, este término adicional no puede afectar sustancialmente la solución al problema.
reaparición. Cuando n es grande, la diferencia entre $T(\lfloor n/2 \rfloor)$ y $T(\lfloor n/2 \rfloor + 17)$ no es tan grande: ambos cortan n casi uniformemente por la mitad. En consecuencia, suponemos que $T(n) = O(n \lg n)$, que puede verificar que es correcta mediante el método de sustitución (consulte el ejercicio 4.1-5).

Otra forma de hacer una buena conjetura es demostrar límites superiores e inferiores sueltos en el recurrencia y luego reducir el rango de incertidumbre. Por ejemplo, podríamos comenzar con una menor límite de $T(n) = \Omega(n)$ para la recurrencia (4.4), ya que tenemos el término n en la recurrencia, y podemos probar un límite superior inicial de $T(n) = O(n^2)$. Luego, podemos bajar gradualmente la parte superior límite y elevar el límite inferior hasta que convergimos en el correcto, asintóticamente apretado solución de $T(n) = \Theta(n \lg n)$.

Sutilezas

Hay momentos en los que se puede adivinar correctamente un límite asintótico en la solución de un recurrencia, pero de alguna manera las matemáticas no parecen funcionar en la inducción. Por lo general, el

Página 61

El problema es que la suposición inductiva no es lo suficientemente fuerte para probar el límite detallado.
Cuando te encuentras con tal inconveniente, revisar la suposición restando un término de orden inferior a menudo permite las matemáticas para pasar.

Considere la recurrencia

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$$

Suponemos que la solución es $O(n)$ y tratamos de demostrar que $T(n) \le cn$ para un elección de la constante c . Sustituyendo nuestra conjetura en la recurrencia, obtenemos

$$\begin{aligned} T(n) &\le do \lfloor n/2 \rfloor + do \lceil n/2 \rceil + 1 \\ &= cn + 1, \end{aligned}$$

lo que no implica $T(n) \le cn$ para cualquier elección de c . Es tentador intentar una suposición más grande, di $T(n) = O(n^2)$, que se puede hacer que funcione, pero de hecho, nuestra suposición de que la solución es $T(n) = O(n)$ es correcto. Sin embargo, para mostrar esto, debemos hacer una hipótesis inductiva más sólida.

Intuitivamente, nuestra suposición es casi correcta: solo estamos fuera de la constante 1, un término de orden inferior.
Sin embargo, la inducción matemática no funciona a menos que demostremos la forma exacta de la hipótesis inductiva. Superamos nuestra dificultad *restando* un término de orden inferior de nuestra conjetura previa. Nuestro nuevo cálculo es $T(n) \le cn - b$, donde $b \ge 0$ es constante. Ahora tenemos

$$\begin{aligned} T(n) &\le (c \lfloor n/2 \rfloor - b) + (c \lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\le cn - b, \end{aligned}$$

siempre que $b \ge 1$. Como antes, la constante c debe elegirse lo suficientemente grande para manejar la condiciones de borde.

La mayoría de las personas consideran que la idea de restar un término de orden inferior es contradictoria. Después de todo, si el las matemáticas no funcionan, ¿no deberíamos aumentar nuestra conjetura? La clave para entender esto El paso es recordar que estamos usando inducción matemática: podemos probar algo más fuerte para un valor dado asumiendo algo más fuerte para valores más pequeños.

Evitando trampas

Es fácil equivocarse en el uso de la notación asintótica. Por ejemplo, en la recurrencia (4.4) podemos "probar" falsamente $T(n) = O(n)$ adivinando $T(n) \le cn$ y luego argumentando

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + n \\ &\le cn + n \\ &= ¡¡O(n), \Leftarrow mal\ !! \end{aligned}$$

ya que c es una constante. El error es que no hemos probado la *forma exacta* del inductivo.
hipótesis, es decir, que $T(n) \le cn$.

Página 62

Cambiar variables

A veces, una pequeña manipulación algebraica puede hacer que una recurrencia desconocida sea similar a una has visto antes. Como ejemplo, considere la recurrencia

que parece difícil. Sin embargo, podemos simplificar esta recurrencia con un cambio de variables. por conveniencia, no nos preocuparemos por redondear valores, tales como, para que sean enteros.
Renombrar $m = \lg n$ produce

$T(2^m) = 2T(2^{m/2}) + m$.

Ahora podemos cambiar el nombre de $S(m) = T(2^m)$ para producir la nueva recurrencia

$S(m) = 2S(m/2) + m$,

que es muy parecido a la recurrencia (4.4). De hecho, esta nueva recurrencia tiene la misma solución: $S(m) = O(m \lg m)$. Volviendo de $S(m)$ a $T(n)$, obtenemos $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Ejercicios 4.1-1

Demuestre que la solución de $T(n) = T(\lceil n/2 \rceil) + 1$ es $O(\lg n)$.

Ejercicios 4.1-2

Vimos que la solución de $T(n) = 2T(\lfloor n/2 \rfloor) + n$ es $O(n \lg n)$. Demuestre que la solución de esto la recurrencia también es $\Omega(n \lg n)$. Concluya que la solución es $\Theta(n \lg n)$.

Ejercicios 4.1-3

Demuestre que al hacer una hipótesis inductiva diferente, podemos superar la dificultad con la condición de límite $T(1) = 1$ para la recurrencia (4.4) sin ajustar el límite condiciones para la prueba inductiva.

Ejercicios 4.1-4

Muestre que $\Theta(n \lg n)$ es la solución a la recurrencia "exacta" (4.2) para la ordenación por fusión.

Página 63

Ejercicios 4.1-5

Demuestre que la solución a $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ es $O(n \lg n)$.

Ejercicios 4.1-6

Resuelve la recurrencia haciendo un cambio de variables. Tu solución debería ser asintóticamente apretado. No se preocupe si los valores son integrales.

4.2 El método del árbol de recursividad

Aunque el método de sustitución puede proporcionar una prueba sucinta de que una solución a una recurrencia es correcto, a veces es difícil hacer una buena suposición. Dibujar una recursividad árbol, como hicimos en nuestro análisis de la recurrencia de clasificación de fusión en la Sección 2.3.2 , es un manera sencilla de idear una buena suposición. En un árbol de recursividad , cada nodo representa el costo de un solo subproblema en algún lugar del conjunto de invocaciones de funciones recursivas. Sumamos el costos dentro de cada nivel del árbol para obtener un conjunto de costos por nivel, y luego sumamos todos los costos por nivel para determinar el costo total de todos los niveles de la recursividad. Los árboles de recursividad son particularmente útil cuando la recurrencia describe el tiempo de ejecución de un divide y vencerás algoritmo.

Un árbol de recursividad se utiliza mejor para generar una buena suposición, que luego es verificada por el método de sustitución. Al usar un árbol de recursividad para generar una buena suposición, a menudo puede tolere una pequeña cantidad de "descuido", ya que estará verificando su conjetura más adelante. Si tu Tenga mucho cuidado al dibujar un árbol de recursividad y sumar los costos, sin embargo, puede utilice un árbol de recursividad como prueba directa de una solución a una recurrencia. En esta sección usaremos árboles de recursividad para generar buenas suposiciones, y en la Sección 4.4 , usaremos árboles de recursividad directamente para demostrar el teorema que forma la base del método maestro.

Por ejemplo, veamos cómo un árbol de recursividad proporcionaría una buena estimación de la recurrencia $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Comenzamos concentrándonos en encontrar un límite superior para la solución. Porque sabemos que los suelos y techos suelen ser insustanciales para resolver las recurrencias. (aquí hay un ejemplo de descuido que podemos tolerar), creamos un árbol de recursividad para el recurrencia $T(n) = 3T(n/4) + cn^2$, habiendo escrito el coeficiente constante implícito $c > 0$.

La figura 4.1 muestra la derivación del árbol de recursividad para $T(n) = 3T(n/4) + cn^2$. por conveniencia, asumimos que n es una potencia exacta de 4 (otro ejemplo de tolerancia descuido). La parte (a) de la figura muestra $T(n)$, que se expande en la parte (b) a un equivalente árbol que representa la recurrencia. El término cn^2 en la raíz representa el costo en el nivel superior de recursividad, y los tres subárboles de la raíz representan los costos incurridos por los subproblemas de tamaño $n/4$. La parte (c) muestra que este proceso avanzó un paso más al expandir cada nodo con costo $T(n/4)$ del inciso b). El costo de cada uno de los tres hijos de la raíz es $c(n/4)^2$. Continuamos

Página 64

expandir cada nodo en el árbol dividiéndolo en sus partes constituyentes según lo determinado por el reaparición.

Figura 4.1: La construcción de un árbol de recursividad para la recurrencia $T(n) = 3T(n/4) + cn^2$. Parte *(a)* muestra $T(n)$, que se expande progresivamente en *(b)* - *(d)* para formar el árbol de recursividad. los El árbol completamente expandido en la parte (d) tiene una altura $\log_4 n$ (tiene niveles $\log_4 n + 1$).

Debido a que el tamaño de los subproblemas disminuye a medida que nos alejamos de la raíz, finalmente debemos alcanzar una condición de frontera. ¿A qué distancia de la raíz llegamos a uno? El tamaño del subproblema para un nodo en la profundidad i es $n/4^i$. Por tanto, el tamaño del subproblema alcanza $n = 1$ cuando $n/4^i = 1$ o, de manera equivalente, cuando $i = \log_4 n$. Por tanto, el árbol tiene $\log_4 n + 1$ niveles (0, 1, 2, ..., $\log_4 n$).

A continuación, determinamos el costo en cada nivel del árbol. Cada nivel tiene tres veces más nodos que el nivel superior, por lo que el número de nodos en la profundidad i es 3^i. Porque los tamaños de los subproblemas Reducir en un factor de 4 para cada nivel que bajamos desde la raíz, cada nodo a la profundidad i, para $i =$ 0, 1, 2, ..., $\log_4 n - 1$, tiene un costo de $c(n/4^i)^2$. Multiplicando, vemos que el costo total en todos los nodos en la profundidad i, para $i = 0, 1, 2, ..., \log_4 n - 1$, es $3^i c(n/4^i)^2 = (3/16)^i cn^2$. El último nivel, a una profundidad $\log_4 n$, tiene nodos, cada uno de los cuales contribuye con un costo $T(1)$, por un costo total de .

Ahora sumamos los costos en todos los niveles para determinar el costo de todo el árbol:

Página 65

Esta última fórmula se ve algo desordenada hasta que nos damos cuenta de que podemos volver a aprovechar pequeñas cantidades de descuido y use una serie geométrica decreciente infinita como una Unido. Retrocediendo un paso y aplicando la ecuación (A.6) , tenemos

Por lo tanto, hemos obtenido una suposición de $T(n) = O(n^2)$ para nuestra recurrencia original $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. En este ejemplo, los coeficientes de cn^2 forman una serie geométrica decreciente y, por ecuación (A.6) , la suma de estos coeficientes está acotada desde arriba por la constante 16/13. Dado que la contribución de la raíz al costo total es cn^2, la raíz aporta una fracción constante de el costo total. En otras palabras, el costo total del árbol está dominado por el costo de la raíz.

De hecho, si $O(n^2)$ es de hecho un límite superior para la recurrencia (como verificaremos en un momento), entonces debe ser un límite apretado. ¿Por qué? La primera llamada recursiva aporta un costo de $\Theta(n^2)$, por lo que $\Omega(n^2)$ debe ser un límite inferior para la recurrencia.

Ahora podemos usar el método de sustitución para verificar que nuestra suposición fue correcta, es decir, $T(n) = O(n^2)$ es un límite superior para la recurrencia $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Queremos mostrar que $T(n) \le dn^2$ para alguna constante $d > 0$. Usando la misma constante $c > 0$ que antes, tenemos

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3días(n/4)^2 + cn^2 \\ &= 3/16\, dn^2 + cn^2 \\ &\le dn^2, \end{aligned}$$

donde el último paso se mantiene mientras $d \ge (16/13)c$.

Como otro ejemplo más complejo, la Figura 4.2 muestra el árbol de recursividad para $T(n) = T(n/3) + T(2n/3) + O(n)$.

Página 66

Figura 4.2: Un árbol de recursividad para la recurrencia $T(n) = T(n/3) + T(2n/3) + cn$.

(Nuevamente, omitimos las funciones de piso y techo por simplicidad.) Como antes, dejamos que c represente el factor constante en el término $O(n)$. Cuando sumamos los valores en los niveles de la recursividad árbol, obtenemos un valor de cn para cada nivel. El camino más largo desde la raíz hasta la hoja es $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Dado que $(2/3)^k n = 1$ cuando $k = \log_{3/2} n$, la altura del árbol es $\log_{3/2} n$.

Intuitivamente, esperamos que la solución a la recurrencia sea como máximo el número de niveles veces el costo de cada nivel, u $O(cn \log_{3/2} n) = O(n \lg n)$. El costo total se distribuye uniformemente en todos los niveles del árbol de recursividad. Aquí hay una complicación: todavía tenemos que considere el costo de las hojas. Si este árbol de recursividad fuera un árbol binario completo de altura $\log_{3/2} n$, habría hojas. Dado que el costo de cada hoja es una constante, el total el costo de todas las hojas sería entonces , que es $\omega(n \lg n)$. Este árbol de recursividad no es un árbol binario completo, sin embargo, por lo que tiene menos de hojas. Además, a medida que bajamos desde la raíz, faltan cada vez más nodos internos. En consecuencia, no todos los niveles contribuir con un costo de exactamente cn; los niveles hacia el fondo contribuyen menos. Podríamos hacer ejercicio una contabilidad precisa de todos los costos, pero recuerde que solo estamos tratando de Supongo que utilizar en el método de sustitución. Toleremos el descuido e intentemos demostrar que una suposición de $O(n \lg n)$ para el límite superior es correcta.

De hecho, podemos usar el método de sustitución para verificar que $O(n \lg n)$ es un límite superior para el solución a la recurrencia. Mostramos que $T(n) \le dn \lg n$, donde d es un positivo adecuado constante. Tenemos

$$
\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (D(n/3)\lg n - d(n/3)\mathrm{LG}\,3) + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= Dn\lg n - d((n/3)\mathrm{LG}\,3 + (2n/3)\lg(3/2)) + cn \\
&= Dn\lg n - d((n/3)\mathrm{LG}\,3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn\lg n,
\end{aligned}
$$

siempre que $d \ge c / (\lg 3 - (2/3))$. Por lo tanto, no tuvimos que realizar una contabilidad más precisa de costos en el árbol de recursividad.

Ejercicios 4.2-1

Página 67

Utilice un árbol de recursividad para determinar un buen límite superior asintótico en la recurrencia $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Utilice el método de sustitución para verificar su respuesta.

Ejercicios 4.2-2

Argumenta que la solución a la recurrencia $T(n) = T(n/3) + T(2n/3) + cn$, donde c es una constante, es $\Omega(n \lg n)$ apelando a un árbol de recursividad.

Ejercicios 4.2-3

Dibuje el árbol de recursividad para $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, donde c es una constante, y proporcione un enlazado asintótico en su solución. Verifique su límite mediante el método de sustitución.

Ejercicios 4.2-4

Utilice un árbol de recursividad para dar una solución asintóticamente ajustada a la recurrencia $T(n) = T(n - a) + T(a) + cn$, donde $a \ge 1$ y $c > 0$ son constantes.

Ejercicios 4.2-5

Utilice un árbol de recursividad para dar una solución asintóticamente ajustada a la recurrencia $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, donde α es una constante en el rango $0 < \alpha < 1$ y $c > 0$ también es una constante.

4.3 El método maestro

El método maestro proporciona un método de "libro de recetas" para resolver las recurrencias del formulario.

(4,5)

donde $a \ge 1$ y $b > 1$ son constantes y $f(n)$ es una función asintóticamente positiva. los
El método maestro requiere la memorización de tres casos, pero luego la solución de muchos
las recurrencias se pueden determinar con bastante facilidad, a menudo sin lápiz y papel.

La recurrencia (4.5) describe el tiempo de ejecución de un algoritmo que divide un problema de tamaño

n en *un* subproblemas, cada uno de tamaño *n* / *b* , donde *un* y *b* son constantes positivas. El *a*

Página 68

los subproblemas se resuelven de forma recursiva, cada uno en el tiempo *T* (*n* / *b*). El costo de dividir el problema y la combinación de los resultados de los subproblemas se describe mediante la función *f* (*n*). (Es decir, usando la notación de la Sección 2.3.2 , *f* (*n*) = *D* (*n*) + *C* (*n*)). Por ejemplo, la recurrencia que surge de el procedimiento MERGE-SORT tiene *a* = 2, *b* = 2 y *f* (*n*) = Θ (*n*).

Como cuestión de corrección técnica, la recurrencia no está realmente bien definida porque *n* / *b*

puede que no sea un número entero. Sustitución de cada uno de los *un* términos *T* (*n* / *b*), ya sea con *T* (⌊ *n* / *b* ⌋) o *T*

Sin embargo, (⌈ *n* / *b* ⌉) no afecta el comportamiento asintótico de la recurrencia. (Probaremos esto en la siguiente sección .) Normalmente nos parece conveniente, por lo tanto, omitir el piso y el techo funciona al escribir recurrencias de divide y vencerás de esta forma.

El teorema maestro

El método maestro depende del siguiente teorema.

Teorema 4.1: (Teorema maestro)

Sean $a \geq 1$ y $b > 1$ constantes, sea $f(n)$ una función, y sea $T(n)$ definida en el enteros no negativos por la recurrencia

$$T(n) = aT(n/b) + f(n),$$

donde interpretamos que n/b significa $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. Entonces $T(n)$ se puede acotar asintóticamente como sigue.

1. Si para alguna constante> 0, entonces
2. Si , luego .
3. Si para alguna constante> 0, y si $af(n/b) \leq cf(n)$ para alguna constante $c < 1$ y todo n suficientemente grande , entonces $T(n) = \Theta(f(n))$.

Antes de aplicar el teorema maestro a algunos ejemplos, dediquemos un momento a intentar entienda lo que dice. En cada uno de los tres casos, estamos comparando la función *f* (*n*) con la función . Intuitivamente, la solución a la recurrencia está determinada por el mayor de los dos funciones. Si, como en el caso 1, la función es mayor, entonces la solución es . Si, como en el caso 3, la función *f* (*n*) es mayor, entonces la solución es *T* (*n*) = Θ (*f* (*n*)). Si, como en En el caso 2, las dos funciones son del mismo tamaño, multiplicamos por un factor logarítmico y el la solución es .

Más allá de esta intuición, hay algunos tecnicismos que deben entenderse. En el primer caso, *f* (*n*) no solo debe ser menor que, debe ser *polinomialmente* menor. Es decir, *f* (*n*) debe ser asintóticamente más pequeño que por un factor de *n* para alguna constante> 0. En el tercer caso, *f* (*n*) no solo debe ser mayor que, debe ser polinomialmente mayor y además satisfacer la condición de "regularidad" que *af* (*n* / *b*) ≤ *cf* (*n*). Esta condición es satisfecha por la mayoría de los funciones acotadas polinomialmente que encontraremos.

Página 69

Es importante darse cuenta de que los tres casos no cubren todas las posibilidades para *f* (*n*). Ahi esta un espacio entre los casos 1 y 2 cuando *f* (*n*) es menor que, pero no polinomialmente menor. De manera similar, existe una brecha entre los casos 2 y 3 cuando *f* (*n*) es mayor que pero no

polinomialmente más grande. Si la función $f(n)$ cae en uno de estos huecos, o si la regularidad condición en caso de que 3 no se mantenga, el método maestro no se puede utilizar para resolver la recurrencia.

Usando el método maestro

Para usar el método maestro, simplemente determinamos qué caso (si lo hay) del teorema maestro aplica y anote la respuesta.

Como primer ejemplo, considere

$T(n) = 9T(n/3) + n$.

Para esta recurrencia, tenemos $a = 9$, $b = 3$, $f(n) = n$, y por lo tanto tenemos que . Ya que , donde = 1, podemos aplicar el caso 1 del teorema maestro y concluir que la solución es $T(n) = \Theta(n^2)$.

Ahora considera

$T(n) = T(2n/3) + 1$,

en el que $a = 1$, $b = 3/2$, $f(n) = 1$, y . Se aplica el caso 2, ya que , por lo que la solución a la recurrencia es $T(n) = \Theta(\lg n)$.

Por la recurrencia

$T(n) = 3T(n/4) + n \lg n$,

tenemos $a = 3$, $b = 4$, $f(n) = n \lg n$, y . Ya que

, donde ≈ 0.2, el caso 3 se aplica si podemos demostrar que la condición de regularidad se cumple para $f(n)$. Para n suficientemente grande , $af(n/b) = 3(n/4) \lg(n/4) \le (3/4) n \lg n = cf(n)$ para $c = 3/4$. En consecuencia, para el caso 3, la solución a la recurrencia es $T(n) = \Theta(n \lg n)$.

El método maestro no se aplica a la recurrencia

$T(n) = 2T(n/2) + n \lg n$,

aunque tiene la forma adecuada: $a = 2$, $b = 2$, $f(n) = n \lg n$, y . Puede parecer que el caso 3 debería aplicarse, ya que $f(n) = n \lg n$ es asintóticamente mayor que . El problema es que no es *polinomialmente* mayor. El radio es asintóticamente menor que n para cualquier constante positiva. En consecuencia, la recurrencia cae en la brecha entre el caso 2 y el caso 3. (Consulte el ejercicio 4.4-2 para obtener una solución).

Ejercicios 4.3-1

Utilice el método maestro para dar límites asintóticos ajustados para las siguientes recurrencias.

Página 70

a. $T(n) = 4T(n/2) + n$.
si. $T(n) = 4T(n/2) + n^2$.
C. $T(n) = 4T(n/2) + n^3$.

Ejercicios 4.3-2

La recurrencia $T(n) = 7T(n/2) + n^2$ describe el tiempo de ejecución de un algoritmo A . Una competencia el algoritmo A' tiene un tiempo de ejecución de $T'(n) = aT'(n/4) + n^2$. ¿Cuál es el valor entero más grande para un tal que A' es asintóticamente más rápido que A ?

Ejercicios 4.3-3

Utilice el método maestro para mostrar que la solución a la recurrencia de la búsqueda binaria $T(n) = T(n/2) + \Theta(1)$ es $T(n) = \Theta(\lg n)$. (Consulte el ejercicio 2.3-5 para obtener una descripción de la búsqueda binaria).

Ejercicios 4.3-4

¿Se puede aplicar el método maestro a la recurrencia $T(n) = 4T(n/2) + n^2 \lg n$? Por que o por que ¿no? Dé un límite superior asintótico para esta recurrencia.

Ejercicios 4.3-5:

Considere la condición de regularidad $af(n/b) \le cf(n)$ para alguna constante $c < 1$, que es parte del caso 3 del teorema maestro. Da un ejemplo de constantes $a \ge 1$ y $b > 1$ y una función $f(n)$ que satisface todas las condiciones del caso 3 del teorema maestro excepto la condición de regularidad.

4.4: Prueba del teorema maestro

Esta sección contiene una demostración del teorema maestro (Teorema 4.1). La prueba no necesita ser entendido para aplicar el teorema.

La prueba consta de dos partes. La primera parte analiza la recurrencia "maestra" (4.5), bajo el simplificando el supuesto de que $T(n)$ se define solo en potencias exactas de $b > 1$, es decir, para $n = 1, b, b^2, ..$ Esta parte proporciona toda la intuición necesaria para comprender por qué el teorema maestro es verdadero. La segunda parte muestra cómo el análisis se puede extender a todos los enteros positivos n y es Técnica meramente matemática aplicada al problema del manejo de suelos y techos.

En esta sección, a veces abusaremos ligeramente de nuestra notación asintótica usándola para describir el comportamiento de funciones que están definidas solo sobre potencias exactas de b . Recuerde que el Las definiciones de notaciones asintóticas requieren que los límites se demuestren para todos los números, no solo los que son poderes de b . Dado que podríamos hacer nuevas notaciones asintóticas que se aplican al conjunto $\{b^i : i = 0, 1, ...\}$, en lugar de los enteros no negativos, este abuso es menor.

Sin embargo, siempre debemos estar en guardia cuando usamos notación asintótica sobre un dominio limitado para que no saquemos conclusiones indebidas. Por ejemplo, probando que $T(n) = O(n)$ cuando n es una potencia exacta de 2 no garantiza que $T(n) = O(n)$. La función $T(n)$ podría definirse como

en cuyo caso el mejor límite superior que puede demostrarse es $T(n) = O(n^2)$. Debido a este tipo de consecuencia drástica, nunca usaremos la notación asintótica sobre un dominio limitado sin dejando absolutamente claro por el contexto que lo estamos haciendo.

4.4.1 La prueba de poderes exactos

La primera parte de la demostración del teorema maestro analiza la recurrencia (4.5)

$$T(n) = aT(n/b) + f(n),$$

para el método maestro, bajo el supuesto de que n es una potencia exacta de $b > 1$, donde b necesita no ser un número entero. El análisis se divide en tres lemas. El primero reduce el problema de resolver la recurrencia maestra al problema de evaluar una expresión que contiene un suma. El segundo determina los límites de esta suma. El tercer lema pone el primero dos juntos para probar una versión del teorema maestro para el caso en el que n es una poder de b .

Lema 4.2

Sean $a \geq 1$ y $b > 1$ constantes, y sea $f(n)$ una función no negativa definida en poderes de b . Defina $T(n)$ en potencias exactas de b por la recurrencia

donde i es un número entero positivo. Luego

(4,6)

Prueba Usamos el árbol de recursividad en la Figura 4.3 . La raíz del árbol ha costado $f(n)$ y tiene un niños, cada uno con costo $f(n / b)$. (Es conveniente pensar en a como un número entero, especialmente al visualizar el árbol de recursividad, pero las matemáticas no lo requieren.) Cada uno de estos niños tiene un niños con coste $f(n / b^2)$, y por lo tanto hay un^2 nodos que son distancia 2 de

Página 72

la raíz. En general, hay un^j nodos que son distancia j de la raíz, y cada uno tiene costo $f(n / b^j)$. El costo de cada hoja es $T(1) = \Theta(1)$, y cada hoja está a una profundidad $\log_b n$, ya que . Existen hojas en el árbol.

Figura 4.3: El árbol de recursividad generado por $T(n) = aT(n / b) + f(n)$. El árbol es un completo a - arbol ario con hojas y altura logarítmica $\log_b n$. El costo de cada nivel se muestra a la derecha y su suma se da en la ecuación (4.6).

Podemos obtener la ecuación (4.6) sumando los costos de cada nivel del árbol, como se muestra en la figura. El costo para un nivel j de nodos internos es $a^j f(n / b^j)$, por lo que el total de todos los nodos internos niveles es

En el algoritmo subyacente de divide y vencerás, esta suma representa los costos de dividir problemas en subproblemas y luego recombinando los subproblemas. El costo de todas las hojas que es el costo de resolver todos los subproblemas de tamaño 1, es .

En términos del árbol de recursividad, los tres casos del teorema maestro corresponden a casos en en el que el costo total del árbol está (1) dominado por los costos en las hojas, (2) uniformemente distribuidos a lo largo de los niveles del árbol, o (3) dominados por el costo de la raíz.

La suma de la ecuación (4.6) describe el costo de dividir y combinar pasos en el algoritmo subyacente de divide y vencerás. El siguiente lema proporciona límites asintóticos en el crecimiento de la suma.

Lema 4.3

Sean $a \geq 1$ y $b > 1$ constantes, y sea $f(n)$ una función no negativa definida en poderes de b . Una función $g(n)$ definida sobre potencias exactas de b por

Página 73

(4,7)

entonces se puede acotar asintóticamente para potencias exactas de b como sigue.

1. Si para alguna constante> 0, entonces .
2. Si , luego .
3. Si $af(n/b) \le cf(n)$ para alguna constante $c < 1$ y para todo $n \ge b$, entonces $g(n) = \Theta(f(n))$.

Prueba Para el caso 1, tenemos , lo que implica que .
Sustituyendo en la ecuación (4.7) se obtiene

(4,8)

Unimos la suma dentro de la notación O factorizando términos y simplificando, lo que deja una serie geométrica creciente:

Dado que by son constantes, podemos reescribir la última expresión como .
Sustituyendo esta expresión por la suma en la ecuación (4.8) se obtiene

y se prueba el caso 1.

Bajo el supuesto de que para el caso 2, tenemos que .
Sustituyendo en la ecuación (4.7) se obtiene

(4,9)

Limitamos la suma dentro de Θ como en el caso 1, pero esta vez no obtenemos una serie. En cambio, descubrimos que todos los términos de la suma son iguales:

Página 74

Sustituyendo esta expresión por la suma en la ecuación (4.9) se obtiene

$g(n) = ($
$= ($,

y se prueba el caso 2.

El caso 3 se prueba de manera similar. Dado que $f(n)$ aparece en la definición (4.7) de $g(n)$ y todos los términos de $g(n)$ no son negativos, podemos concluir que $g(n) = \Omega(f(n))$ para potencias exactas de b . Bajo nuestro Suponiendo que $af(n/b) \le cf(n)$ para alguna constante $c < 1$ y todo $n \ge b$, tenemos $f(n/b) \le (c/a)f(n)$. Iterando j veces, tenemos $f(n/b^j) \le (c/a)^j f(n)$ o, de manera equivalente, $a^j f(n/b^j) \le c^j f(n)$. Sustituir en la ecuación (4.7) y simplificar produce una serie geométrica, pero a diferencia de la serie en el caso 1, este tiene términos decrecientes:

ya que c es constante. Por lo tanto, podemos concluir que $g(n) = \Theta(f(n))$ para potencias exactas de b . El caso 3 es probado, que completa la demostración del lema.

Ahora podemos probar una versión del teorema maestro para el caso en el que n es una potencia exacta de b .

Lema 4.4

Sean $a \ge 1$ y $b > 1$ constantes, y sea $f(n)$ una función no negativa definida en poderes de b . Defina $T(n)$ en potencias exactas de b por la recurrencia

Página 75

donde i es un número entero positivo. Entonces $T(n)$ se puede acotar asintóticamente para potencias exactas de b como sigue.

1. Si para alguna constante> 0, entonces .
2. Si , luego .
3. Si para alguna constante> 0, y si $af(n/b) \le cf(n)$ para alguna constante $c <$ 1 y todo n suficientemente grande , entonces $T(n) = \Theta(f(n))$.

Demostración Usamos los límites en el Lema 4.3 para evaluar la suma (4.6) del Lema 4.2 .
Para el caso 1, tenemos

$T(n) =$

= ,

y para el caso 2,

$T(n) =$
= .

Para el caso 3,

$T(n) =$
$= \Theta(f(n))$,

porque .

4.4.2 Pisos y techos

Para completar la demostración del teorema maestro, ahora debemos extender nuestro análisis al situación en la que se utilizan pisos y techos en la recurrencia maestra, de modo que la recurrencia se define para todos los números enteros, no solo para las potencias exactas de b . Obtener un límite inferior en

(4.10)

y un límite superior en

(4,11)

es una rutina, ya que el límite $\lceil n / b \rceil \geq n / b$ se puede empujar a través en el primer caso para producir el resultado deseado, y el límite $\lfloor n / b \rfloor \leq n / b$ se puede impulsar en el segundo caso. Inferior delimitar la recurrencia (4.11) requiere la misma técnica que para delimitar recurrencia (4.10), por lo que presentaremos solo este último límite.

Página 76

Modificamos el árbol de recursividad de la Figura 4.3 para producir el árbol de recursividad de la Figura 4.4 . Como nosotros bajar en el árbol de recursividad, obtenemos una secuencia de invocaciones recursivas en los argumentos

Figura 4.4: El árbol de recursividad generado por $T(n) = aT(\lceil n / b \rceil) + f(n)$. El argumento recursivo n_j viene dado por la ecuación (4.12).

n ,
$\lceil n / b \rceil$,
$\lceil \lceil n / b \rceil / b \rceil$,
$\lceil \lceil \lceil n / b \rceil / b \rceil / b \rceil$,
⋮

Denotemos el j- ésimo elemento en la secuencia por n_j , donde

(4.12)

Nuestro primer objetivo es determinar la profundidad k tal que n_k sea una constante. Usando la desigualdad $\lceil x \rceil \leq x + 1$, obtenemos

En general,

Página 77

Dejando $j = \lfloor \log_b n \rfloor$, obtenemos

y así vemos que a la profundidad $\lfloor \log_b n \rfloor$, el tamaño del problema es como mucho una constante.

En la Figura 4.4 , vemos que

(4,13)

que es muy similar a la ecuación (4.6) , excepto que n es un entero arbitrario y no restringido a ser un poder exacto de b .

Ahora podemos evaluar la suma

(4,14)

de (4.13) de una manera análoga a la demostración del Lema 4.3 . Comenzando con el caso 3, si $af(\lceil n / b \rceil) \le cf(n)$ para $n > b + b/(b - 1)$, donde $c < 1$ es una constante, entonces se sigue que $a^j f(n_j) \le c^j f(n)$. Por tanto, la suma de la ecuación (4.14) se puede evaluar como en el Lema 4.3 . En caso 2, tenemos . Si podemos mostrar eso , entonces la prueba de el caso 2 del Lema 4.3 pasará. Observe que $j = \le \lfloor \log_b n \rfloor$ implica $b^j / n \le 1$. El límite implica que existe una constante $c > 0$ tal que para todo n_j suficientemente grande ,

Página 78

ya que es una constante. Por tanto, se prueba el caso 2. La prueba del caso 1 es casi idéntico. La clave es demostrar el límite , que es similar al correspondiente prueba del caso 2, aunque el álgebra es más compleja.

Ahora hemos probado los límites superiores en el teorema maestro para todos los enteros n . La prueba de los límites inferiores son similares.

Ejercicios 4.4-1:

Dé una expresión simple y exacta para n_j en la ecuación (4.12) para el caso en el que b es un entero positivo en lugar de un número real arbitrario.

Ejercicios 4.4-2:

Demuestra que si , donde $k \geq 0$, entonces la recurrencia maestra tiene solución . Para simplificar, limite su análisis a las potencias exactas de b .

Ejercicios 4.4-3:

Demuestre que el caso 3 del teorema maestro está exagerado, en el sentido de que la condición de regularidad $af(n / b) \leq cf(n)$ para alguna constante $c < 1$ implica que existe una constante> 0 tal que .

Problemas 4-1: ejemplos de recurrencia

Dé los límites superior e inferior asintóticos para $T(n)$ en cada una de las siguientes recurrencias. Suponga que $T(n)$ es constante para $n \leq 2$. Haga sus límites lo más ajustados posible y justifique su respuestas.

a. $T(n) = 2T(n / 2) + n^3$.
si. $T(n) = T(9n / 10) + n$.
C. $T(n) = 16T(n / 4) + n^2$.
re. $T(n) = 7T(n / 3) + n^2$.
mi. $T(n) = 7T(n / 2) + n^2$.
F. .
gramo. $T(n) = T(n - 1) + n$.
h. .

Página 79

Problemas 4-2: Encontrar el entero faltante

Una matriz A [1 n] contiene todos los números enteros de 0 a n excepto uno. Sería fácil determinar el entero faltante en el tiempo O (n) usando una matriz auxiliar B [0 n] para registrar qué números aparecen en *una* . En este problema, sin embargo, no podemos acceder a un entero completo en A con una sola operación. Los elementos de A se representan en binario y la única operación que podemos usar para acceder a ellos es "buscar el j- ésimo bit de A [i]", lo que lleva un tiempo constante.

Demuestre que si usamos solo esta operación, aún podemos determinar el entero que falta en O (n) hora.

Problemas 4-3: Costos de pasar parámetros

A lo largo de este libro, asumimos que el paso de parámetros durante las llamadas a procedimientos toma tiempo constante, incluso si se pasa una matriz de N elementos. Esta suposición es válida en la mayoría de sistemas porque se pasa un puntero a la matriz, no la matriz en sí. Este problema examina las implicaciones de tres estrategias de paso de parámetros:

1. Una matriz se pasa por puntero. Tiempo = Θ (1).
2. Una matriz se pasa copiando. Tiempo = Θ (N), donde N es el tamaño de la matriz.
3. Una matriz se pasa copiando solo el subrango al que podría acceder el llamado procedimiento. Tiempo = Θ (q - p + 1) si se pasa el subarreglo A [pq].

a. Considere el algoritmo de búsqueda binaria recursiva para encontrar un número en una matriz ordenada (vea el ejercicio 2.3-5). Dar recurrencias para los peores tiempos de ejecución de binario buscar cuando se pasan matrices utilizando cada uno de los tres métodos anteriores, y dar una buena límites superiores en las soluciones de las recurrencias. Sea N el tamaño del original problema yn del tamaño de un subproblema.

si. Rehaga la parte (a) para el algoritmo MERGE-SORT de la Sección 2.3.1 .

Problemas 4-4: más ejemplos de recurrencia

Dé los límites superior e inferior asintóticos para T (n) en cada una de las siguientes recurrencias. Suponga que T (n) es constante para n suficientemente pequeño . Haz tus límites lo más ajustados posible, y justifique sus respuestas.

a. T (n) = 3 T (n / 2) + n lg n .
si. T (n) = 5 T (n / 5) + n / lg n .
C.
re. T (n) = 3 T (n / 3 + 5) + n / 2.
mi. T (n) = 2 T (n / 2) + n / lg n .
F. T (n) = T (n / 2) + T (n / 4) + T (n / 8) + n .
gramo. T (n) = T (n - 1) + 1 / n .

Página 80

h. T (n) = T (n - 1) + lg n .
yo. T (n) = T (n - 2) + 2 lg n .
j. .

Problemas 4-5: números de Fibonacci

Este problema desarrolla propiedades de los números de Fibonacci, que se definen por recurrencia (3,21). Usaremos la técnica de generar funciones para resolver la recurrencia de Fibonacci. Defina la función generadora (o serie formal de potencia) F como

donde F_i es el i- ésimo número de Fibonacci.

a. Muestre que $F(z) = z + z\,F(z) + z^2 F(z)$.
si. Muestra esa

dónde

y

C. Muestra esa

re. Pruebalo para $i > 0$, redondeado al número entero más cercano. (*Sugerencia:* observe
mi. Demuestre que $F_{i+2} \geq \varphi^i$ para $i \geq 0$.

Problemas 4-6: prueba del chip VLSI

Página 81

El profesor Diogenes tiene n chips VLSI [1] supuestamente idénticos que, en principio, son capaces de probándose unos a otros. La plantilla de prueba del profesor tiene capacidad para dos chips a la vez. Cuando la plantilla es cargado, cada chip prueba el otro e informa si es bueno o malo. Un buen chip siempre informa con precisión si el otro chip es bueno o malo, pero la respuesta de un chip defectuoso no puede se Confiable. Por tanto, los cuatro posibles resultados de una prueba son los siguientes:

Chip A dice	**Chip B dice**	**Conclusión**
B es bueno	*A* es bueno	ambos son buenos o ambos son malos
B es bueno	*A* es malo	al menos uno es malo
B es malo	*A* es bueno	al menos uno es malo
B es malo	*A* es malo	al menos uno es malo

a. Demuestre que si más de $n/2$ chips son malos, el profesor no necesariamente puede determinar qué chips son buenos usando cualquier estrategia basada en este tipo de prueba por pares. Asumir que las malas fichas pueden conspirar para engañar al profesor.
si. Considere el problema de encontrar un solo chip bueno entre n chips, asumiendo que más de $n/2$ de los chips son buenos. Demuestre que $\lfloor n/2 \rfloor$ pruebas por pares son suficientes para reduzca el problema a casi la mitad del tamaño.
C. Demuestre que los chips buenos se pueden identificar con $\Theta(n)$ pruebas por pares, suponiendo que más de $n/2$ de los chips son buenos. Dar y resolver la recurrencia que describe la número de pruebas.

Problemas 4-7: matrices Monge

Una matriz A $m \times n$ de números reales es una **matriz de Monge** si para todo i , j , k y l tal que $1 \le i < k \le m$ y $1 \le j < l \le n$, tenemos

$A [i , j] + A [k , l] \le A [i , l] + A [k , j]$.

En otras palabras, siempre que elegimos dos filas y dos columnas de una matriz de Monge y consideramos los cuatro elementos en las intersecciones de las filas y las columnas, la suma de la parte superior izquierda y los elementos inferior derecha es menor o igual a la suma de los elementos inferior izquierdo y superior derecho elementos. Por ejemplo, la siguiente matriz es Monge:

10 17 13 28 23
17 22 16 29 23
24 28 22 34 24
11 13 6 17 7
45 44 32 37 23
36 33 19 21 6
75 66 51 53 34

a. Demuestre que una matriz es Monge si y solo si para todo $i = 1, 2, \ldots, m - 1$ y $j = 1, 2, \ldots, n - 1$, tenemos

$A [i , j] + A [i + 1, j + 1] \le A [i , j + 1] + A [i + 1, j]$.

Nota (Para la parte "solo si", use la inducción por separado en filas y columnas).

si. La siguiente matriz no es Monge. Cambie un elemento para convertirlo en Monge. (*Sugerencia:* utilice el inciso a).)

37 23 22 32
21 6 7 10
53 34 30 31
32 13 9 6
43 21 15 8

C. Sea $f (i)$ el índice de la columna que contiene el elemento mínimo más a la izquierda de la fila i . Demuestre que $f (1) \le f (2) \le \cdots \le f (m)$ para cualquier matriz de Monge $m \times n$.

re. Aquí hay una descripción de un algoritmo de divide y vencerás que calcula el extremo izquierdo elemento mínimo en cada fila de una matriz Monge $m \times n$ A :

- Construir una submatriz A 'de A que consiste en las filas de numeración par de A . Determina recursivamente el mínimo más a la izquierda para cada fila de $A '$. Luego calcular el mínimo más a la izquierda en las filas impares de A .

Explique cómo calcular el mínimo más a la izquierda en las filas impares de A (dado que se conoce el mínimo más a la izquierda de las filas pares) en tiempo $O (m + n)$.

mi. Escriba la recurrencia que describa el tiempo de ejecución del algoritmo descrito en la parte (d). Demuestre que su solución es $O (m + n \log m)$.

[1] VLSI significa "integración a muy gran escala", que es el chip de circuito integrado tecnología utilizada para fabricar la mayoría de los microprocesadores en la actualidad.

Notas del capítulo

Las recurrencias fueron estudiadas ya en 1202 por L. Fibonacci, para quien los números de Fibonacci son llamados. A. De Moivre introdujo el método de generar funciones (vea el problema 4-5) para resolver recurrencias. El método maestro está adaptado de Bentley, Haken y Saxe [41] ,

que proporciona el método ampliado justificado por el ejercicio 4.4-2. Knuth [182] y Liu [205] muestran cómo resolver recurrencias lineales usando el método de generación de funciones. Purdom y Brown [252] y Graham, Knuth y Patashnik [132] contienen amplias discusiones sobre resolución de recurrencias.

Varios investigadores, incluidos Akra y Bazzi [13] , Roura [262] y Verma [306] , han dados métodos para resolver recurrencias de divide y vencerás más generales que los resueltos por

el método maestro. Aquí describimos el resultado de Akra y Bazzi, que funciona para recurrencias de la forma

(4,15)

donde $k \geq 1$; todos los coeficientes a_i son positivos y suman al menos 1; todos b_i son al menos 2; $f(n)$ es acotado, positivo y no decreciente; y para todas las constantes $c > 1$, existen constantes n_0, $d > 0$ tal que $f(n/c) \geq df(n)$ para todo $n \geq n_0$. Este método funcionaría en una recurrencia como $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$, para lo cual no se aplica el método maestro. Para resolver el recurrencia (4.15), primero encontramos el valor de p tal que . (Tal p siempre existe, y es única y positiva.) La solución a la recurrencia es entonces

para n' una constante suficientemente grande. El método Akra-Bazzi puede ser algo difícil de usar, pero sirve para resolver las recurrencias que modelan la división del problema en subproblemas de tamaño desigual. El método maestro es más simple de usar, pero se aplica solo cuando los tamaños de los subproblemas son iguales.

Capítulo 5: Análisis probabilístico y Algoritmos aleatorios

Este capítulo presenta el análisis probabilístico y los algoritmos aleatorios. Si usted es si no está familiarizado con los conceptos básicos de la teoría de la probabilidad, debe leer el Apéndice C , que revisa este material. El análisis probabilístico y los algoritmos aleatorios se revisarán varias veces. a lo largo de este libro.

5.1 El problema de la contratación

Suponga que necesita contratar un nuevo asistente de oficina. Sus intentos anteriores de contratación han no ha tenido éxito y decide utilizar una agencia de empleo. La agencia de empleo le enviará un candidato cada día. Entrevistará a esa persona y luego decidirá si contrate a esa persona o no. Debe pagar a la agencia de empleo una pequeña tarifa para entrevistar a un solicitante. Sin embargo, contratar a un solicitante es más costoso, ya que debe despedir a su asistente de oficina actual y pagar una gran tarifa de contratación a la agencia de empleo. Usted está comprometidos con tener, en todo momento, la mejor persona posible para el puesto. Por eso tu decides que, después de entrevistar a cada solicitante, si ese solicitante está mejor calificado que el actual asistente de oficina, despedirá al asistente de oficina actual y contratará al nuevo solicitante. Usted está dispuesto a pagar el precio resultante de esta estrategia, pero desea estimar cuál será el precio ser.

El procedimiento CONTRATANTE-ASISTENTE, que se detalla a continuación, expresa esta estrategia de contratación en pseudocódigo. Se asume que los candidatos para el puesto de asistente de oficina están numerados del 1 al n . El procedimiento asume que usted puede, después de entrevistar al candidato i , determinar si

El candidato *i* es el mejor candidato que ha visto hasta ahora. Para inicializar, el procedimiento crea un candidato ficticio, numerado 0, que está menos calificado que cada uno de los demás candidatos.

```
AYUDANTE DE CONTRATACIÓN ( n )
1 mejor ← 0 → candidato 0 es un candidato ficticio menos calificado
2 para i ← 1 a n
3 do entrevista candidato i
4                   si el candidato i es mejor que el candidato mejor
5                       entonces mejor ← i
6                                   contratar candidato i
```

El modelo de costos para este problema difiere del modelo descrito en el Capítulo 2 . No somos preocupado por el tiempo de ejecución de HIRE-ASSISTANT, pero en cambio con el costo incurrido por entrevista y contratación. En la superficie, analizar el costo de este algoritmo puede parecer muy diferente de analizar el tiempo de ejecución de, digamos, combinación de clasificación. Las técnicas analíticas utilizadas, sin embargo, son idénticos tanto si analizamos el coste como el tiempo de ejecución. En cualquier caso, estamos contando el número de veces que se ejecutan determinadas operaciones básicas.

Entrevistar tiene un costo bajo, digamos c_i , mientras que contratar es costoso, cuesta c_h . Sea m el número de personas contratadas. Entonces, el costo total asociado con este algoritmo es $O(nc_i + mc_h)$. No importa cuántas personas contratemos, siempre entrevistamos n candidatos y, por lo tanto, siempre incurrimos en el costar nc_i asociado con la entrevista. Por tanto, nos concentramos en analizar mc_h , la contratación costo. Esta cantidad varía con cada ejecución del algoritmo.

Este escenario sirve como modelo para un paradigma computacional común. A menudo es el caso que necesitamos encontrar el valor máximo o mínimo en una secuencia examinando cada elemento de la secuencia y manteniendo un "ganador" actual. El problema de la contratación modela cómo a menudo actualizamos nuestra noción de qué elemento está ganando actualmente.

Análisis del peor de los casos

En el peor de los casos, contratamos a todos los candidatos que entrevistamos. Esta situación ocurre si los candidatos vienen en orden creciente de calidad, en cuyo caso contratamos n veces, para un total coste de contratación de $O(nc_h)$.

Sin embargo, podría ser razonable esperar que los candidatos no siempre entren orden creciente de calidad. De hecho, no tenemos idea del orden en que llegan, ni ¿Tenemos algún control sobre este pedido? Por lo tanto, es natural preguntar qué esperamos suceden en un caso típico o promedio.

Análisis probabilístico

El análisis probabilístico es el uso de la probabilidad en el análisis de problemas. Más comúnmente, utilizamos el análisis probabilístico para analizar el tiempo de ejecución de un algoritmo. A veces, usamos para analizar otras cantidades, como el coste de contratación en el procedimiento CONTRATACIÓN-ASISTENTE. En Para realizar un análisis probabilístico, debemos utilizar el conocimiento o hacer suposiciones. sobre, la distribución de los insumos. Luego analizamos nuestro algoritmo, calculando un esperado tiempo de ejecución. Se asume la expectativa sobre la distribución de los posibles insumos. Así nosotros están, en efecto, promediando el tiempo de ejecución sobre todas las entradas posibles.

Página 85

Debemos tener mucho cuidado al decidir la distribución de insumos. Para algunos problemas, es razonable asumir algo sobre el conjunto de todas las entradas posibles, y podemos usar análisis probabilístico como técnica para diseñar un algoritmo eficiente y como medio para obtener información sobre un problema. Para otros problemas, no podemos describir una entrada razonable distribución, y en estos casos no podemos utilizar el análisis probabilístico.

Para el problema de contratación, podemos asumir que los solicitantes vienen en un orden aleatorio. Que hace que significa para este problema? Suponemos que podemos comparar dos candidatos cualesquiera y decidir cuál está mejor calificado; es decir, hay un orden total sobre los candidatos. (Ver Apéndice B para la definición de un orden total). Por lo tanto, podemos clasificar a cada candidato con un número del 1 al n , usando el *rango (i)* para denotar el rango del solicitante i , y adopte el convención de que un rango más alto corresponde a un solicitante mejor calificado. La lista ordenada

< *rango* (1), *rango* (2), ..., *rango* (*n*)> es una permutación de la lista <1, 2, ..., *n* >. Diciendo que el los solicitantes vienen en un orden aleatorio es equivalente a decir que esta lista de rangos es igualmente probablemente sea cualquiera de los *n* ! permutaciones de los números del 1 al *n* . Alternativamente, decimos que los rangos forman una permutación aleatoria uniforme ; es decir, cada uno de los posibles *n* ! las permutaciones aparecen con igual probabilidad.

La sección 5.2 contiene un análisis probabilístico del problema de contratación.

Algoritmos aleatorios

Para utilizar el análisis probabilístico, necesitamos saber algo sobre la distribución en el entradas. En muchos casos, sabemos muy poco sobre la distribución de entrada. Incluso si lo sabemos algo sobre la distribución, es posible que no podamos modelar este conocimiento computacionalmente. Sin embargo, a menudo podemos usar la probabilidad y la aleatoriedad como una herramienta para el algoritmo. diseño y análisis, haciendo aleatorio el comportamiento de parte del algoritmo.

En el problema de la contratación, puede parecer que los candidatos se nos presentan al azar. orden, pero no tenemos forma de saber si realmente lo son o no. Por lo tanto, para Desarrollar un algoritmo aleatorio para el problema de contratación, debemos tener un mayor control sobre el orden en el que entrevistamos a los candidatos. Por tanto, cambiaremos ligeramente el modelo. Nosotros dirán que la agencia de empleo tiene *n* candidatos, y nos envían una lista de los candidatos por adelantado. Cada día, elegimos, al azar, a qué candidato entrevistar. Aunque no sabemos nada sobre los candidatos (además de sus nombres), hemos realizado una cambio significativo. En lugar de confiar en la suposición de que los candidatos vendrán a nosotros en un orden aleatorio, en su lugar hemos ganado el control del proceso y hemos aplicado un orden aleatorio.

De manera más general, llamamos aleatorio a un algoritmo si su comportamiento está determinado no solo por su entrada, sino también por valores producidos por un generador de números aleatorios . Asumiremos que nosotros tenemos a nuestra disposición un generador de números aleatorios RANDOM. Una llamada a RANDOM (*a* , *b*) devuelve un número entero comprendido entre *un* y *b* , inclusive, con ser igualmente probable cada uno de tales entero. por ejemplo, RANDOM (0, 1) produce 0 con probabilidad 1/2, y produce 1 con probabilidad 1/2. Una llamada a RANDOM (3, 7) devuelve 3, 4, 5, 6 o 7, cada uno con una probabilidad de 1/5. Cada El entero devuelto por RANDOM es independiente de los enteros devueltos en llamadas anteriores. Tú Puede imaginarse ALEATORIO lanzando un dado de lados (*b* - *a* + 1) para obtener su salida. (En la práctica, la mayoría de los entornos de programación ofrecen un generador de números pseudoaleatorios : un determinista algoritmo que devuelve números que "parecen" estadísticamente aleatorios).

Ejercicios 5.1-1

Página 86

Demuestre que la suposición de que siempre podemos determinar qué candidato es el mejor en la línea 4 del procedimiento HIRE-ASSISTANT implica que conocemos un orden total en las filas del candidatos.

Ejercicios 5.1-2:

Describir una implementación del procedimiento RANDOM (*a* , *b*) que solo realiza llamadas a ALEATORIO (0, 1). ¿Cuál es el tiempo de ejecución esperado de su procedimiento, en función de *un* y *b* ?

Ejercicios 5.1-3:

Suponga que desea generar 0 con probabilidad 1/2 y 1 con probabilidad 1/2. En tu la eliminación es un procedimiento BIASED-RANDOM, que genera 0 o 1. Genera 1 con algunos probabilidad *py* 0 con probabilidad 1 - *p* , donde 0 < *p* <1, pero no sabe qué es *p* . Proporcione un algoritmo que utilice BIASED-RANDOM como subrutina y devuelva un respuesta, devolviendo 0 con probabilidad 1/2 y 1 con probabilidad 1/2. Que es lo esperado tiempo de ejecución de su algoritmo en función de *p* ?

5.2 Variables aleatorias del indicador

Para analizar muchos algoritmos, incluido el problema de contratación, usaremos indicador
variables aleatorias. Las variables aleatorias del indicador proporcionan un método conveniente para convertir
entre probabilidades y expectativas. Supongamos que se nos da un espacio muestral S y un evento
Una . Entonces, la variable aleatoria del indicador $I\{A\}$ asociada con el evento A se define como

(5,1)

Como ejemplo simple, determinemos el número esperado de caras que obtenemos cuando
lanzar una moneda justa. Nuestro espacio muestral es $S = \{H, T\}$, y definimos una variable aleatoria Y que
toma los valores H y T , cada uno con probabilidad 1/2. Entonces podemos definir un indicador
variable aleatoria X_H , asociada con la moneda que sale cara, que podemos expresar como
evento $Y = H$. Esta variable cuenta el número de caras obtenidas en este lanzamiento, y es 1 si el
la moneda sale cara y 0 en caso contrario. Nosotros escribimos

El número esperado de caras obtenidas en un lanzamiento de la moneda es simplemente el valor esperado de
nuestra variable indicadora X_H :

Página 87

$$\begin{aligned} E[X_H] &= E[I\{Y = H\}] \\ &= 1 \cdot \Pr\{Y = H\} + 0 \cdot \Pr\{Y = T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2. \end{aligned}$$

Por lo tanto, el número esperado de caras obtenidas con un lanzamiento de una moneda justa es 1/2. Como el seguiente
El lema muestra, el valor esperado de una variable aleatoria indicadora asociada con un evento A
es igual a la probabilidad de que ocurra A.

Lema 5.1

Dado un espacio muestral S y un evento A en el espacio muestral S , sea $X_A = I\{A\}$. Entonces $E[X_A] =$
$\Pr\{A\}$.

Prueba Por la definición de una variable aleatoria indicadora de la ecuación 1) y la definición de
valor esperado, tenemos

$$\begin{aligned} E[X_A] &= E[I\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\bar{A}\} \\ &= \Pr\{A\}, \end{aligned}$$

donde A representa S - A , el complemento de A .

Aunque las variables aleatorias del indicador pueden parecer engorrosas para una aplicación como
contar el número esperado de caras en un lanzamiento de una sola moneda, son útiles para analizar
situaciones en las que realizamos ensayos aleatorios repetidos. Por ejemplo, indicador aleatorio
Las variables nos dan una forma sencilla de llegar al resultado de la ecuación (C.36) . En esta ecuación, nosotros
Calcule el número de caras en n lanzamientos de moneda considerando por separado la probabilidad de
obteniendo 0 caras, 1 caras, 2 caras, etc. Sin embargo, el método más simple propuesto en la ecuación
(C.37) en realidad usa implícitamente variables aleatorias de indicador. Haciendo este argumento más
explícita, podemos dejar que X_i sea la variable aleatoria indicadora asociada con el evento en el que el
i th flip sale cara. Dejando que Y_i sea la variable aleatoria que denota el resultado del i- ésimo giro,
tenemos que $X_i = I\{Y_i = H\}$. Sea X la variable aleatoria que denota el número total de caras
en los n lanzamientos de monedas, de modo que

Deseamos calcular el número esperado de caras, por lo que tomamos la expectativa de ambos lados de la ecuación anterior para obtener

Página 88

El lado izquierdo de la ecuación anterior es la expectativa de la suma de n variables aleatorias. Por Lema 5.1 , podemos calcular fácilmente la expectativa de cada una de las variables aleatorias. Por ecuación (C.20) -linealidad de la expectativa-es fácil calcular la expectativa de la suma: es igual a la suma de las expectativas de las n variables aleatorias. La linealidad de la expectativa hace el uso de indicadores de variables aleatorias, una poderosa técnica analítica; se aplica incluso cuando hay dependencia entre las variables aleatorias. Ahora podemos calcular fácilmente el esperado número de cabezas:

Por lo tanto, en comparación con el método utilizado en la ecuación (C.36) , las variables aleatorias indicadoras simplificar el cálculo. Usaremos variables aleatorias indicadoras a lo largo de este libro.

Análisis del problema de contratación mediante indicadores de variables aleatorias

Volviendo al problema de la contratación, ahora deseamos calcular el número esperado de veces que contratamos un nuevo asistente de oficina. Para utilizar un análisis probabilístico, asumimos que el los candidatos llegan en un orden aleatorio, como se discutió en la sección anterior . (Veremos en Sección 5.3 cómo eliminar esta suposición.) Sea X la variable aleatoria cuyo valor es igual a la cantidad de veces que contratamos a un nuevo asistente de oficina. Entonces podríamos aplicar la definición de valor esperado de la ecuación (C.19) para obtener

pero este cálculo sería engorroso. En su lugar, utilizaremos variables aleatorias de indicador para simplifica enormemente el cálculo.

Para usar indicadores de variables aleatorias, en lugar de calcular E [X] definiendo una variable asociado con la cantidad de veces que contratamos a un nuevo asistente de oficina, definimos n variables relacionado con la contratación o no de cada candidato en particular. En particular, dejamos que X_i sea el Indicador variable aleatoria asociada con el evento en el que la i es contratado º candidato. Así,

(5,2)

y

(5,3)

Por el Lema 5.1 , tenemos que

Página 89

E [X_i] = Pr {candidato i es contratado},

y, por lo tanto, debemos calcular la probabilidad de que las líneas 5-6 de HIRE-ASSISTANT sean ejecutado.

El candidato i es contratado, en la línea 5, exactamente cuando el candidato i es mejor que cada uno de los candidatos 1 hasta i - 1. Debido a que hemos asumido que los candidatos llegan en un orden aleatorio, la primera i *los* candidatos han aparecido en un orden aleatorio. Cualquiera de estos primeros i candidatos es igualmente probablemente sea el mejor calificado hasta ahora. El candidato i tiene una probabilidad de 1 / i de ser mejor calificado que los candidatos 1 a i - 1 y, por lo tanto, una probabilidad de 1 / i de ser contratado. Por Lema 5.1 , concluimos que

(5,4)

Ahora podemos calcular E [X]:

(5,5)

(5,6)

A pesar de que entrevistamos a n personas, en realidad solo contratamos aproximadamente a n de ellas, en promedio. Resumimos este resultado en el siguiente lema.

Lema 5.2

Suponiendo que los candidatos se presentan en un orden aleatorio, el algoritmo HIRE-ASSISTANT tiene un costo total de contratación de O ($c_h \ln n$).

Prueba El límite se deriva inmediatamente de nuestra definición del costo de contratación y la ecuación (5,6) .

El costo esperado de la entrevista es una mejora significativa sobre el costo de contratación del peor caso O (nc_h).

Ejercicios 5.2-1

En HIRE-ASSISTANT, asumiendo que los candidatos se presentan en un orden aleatorio, lo que es la probabilidad de que contrate exactamente una vez? ¿Cuál es la probabilidad de que contrate exactamente n veces?

Página 90

Ejercicios 5.2-2

En HIRE-ASSISTANT, asumiendo que los candidatos se presentan en un orden aleatorio, lo que es la probabilidad de que contrate exactamente dos veces?

Ejercicios 5.2-3

Utilice variables aleatorias de indicador para calcular el valor esperado de la suma de n dados.

Ejercicios 5.2-4

Variables aleatorias uso de indicadores para resolver el siguiente problema, que se conoce como el sombrero-comprobar problema . Cada uno de n clientes regala un sombrero a una persona a cuadros de sombrero en un restaurante. El sombrero-La persona de control devuelve los sombreros a los clientes en un orden aleatorio. Que es lo esperado número de clientes que recuperan su propio sombrero?

Ejercicios 5.2-5

Sea A [1 .. n] una matriz de n números distintos. Si $i < j$ y A [i]> A [j], entonces el par (i , j) es llama una inversión de A . (Consulte el problema 2-4 para obtener más información sobre las inversiones). Suponga que cada El elemento de A se elige aleatoria, independiente y uniformemente del rango de 1 a n . Utilice variables aleatorias de indicador para calcular el número esperado de inversiones.

5.3 Algoritmos aleatorios

En la sección anterior , mostramos cómo conocer una distribución de las entradas puede ayudarnos a Analizar el comportamiento de caso promedio de un algoritmo. Muchas veces, no tenemos tales conocimiento y no es posible un análisis de casos promedio. Como se menciona en la Sección 5.1 , podemos ser capaz de utilizar un algoritmo aleatorio.

Para un problema como el de la contratación, en el que es útil asumir que todos permutaciones de la entrada son igualmente probables, un análisis probabilístico guiará la desarrollo de un algoritmo aleatorizado. En lugar de asumir una distribución de insumos, imponer una distribución. En particular, antes de ejecutar el algoritmo, permutamos aleatoriamente el candidatos para hacer cumplir la propiedad de que cada permutación es igualmente probable. Esta La modificación no cambia nuestra expectativa de contratar un nuevo asistente de oficina aproximadamente en n n veces. Sin embargo, significa que para *cualquier* entrada esperamos que este sea el caso, en lugar de insumos extraídos de una distribución particular.

Página 91

Ahora exploramos la distinción entre análisis probabilístico y algoritmos aleatorios. más lejos. En la Sección 5.2 , afirmamos que, asumiendo que los candidatos se presentan en un orden aleatorio, el número esperado de veces que contratamos a un nuevo asistente de oficina es aproximadamente ln n . Nota que el algoritmo aquí es determinista; para cualquier entrada en particular, el número de veces que un nuevo El asistente de oficina que se contrate será siempre el mismo. Además, la cantidad de veces que contratamos a un nuevo asistente de oficina difiere para diferentes entradas, y depende de los rangos de los diversos candidatos. Dado que este número depende solo de las filas de los candidatos, podemos representar una entrada particular enumerando, en orden, los rangos de los candidatos, es decir, < *rango* (1), *rango* (2), ..., *rango* (n)>. Dada la lista de rango A_1 =<1, 2, 3, 4, 5, 6, 7, 8, 9, 10>, un nuevo asistente de oficina ser contratado siempre 10 veces, ya que cada candidato sucesivo es mejor que el anterior, y Las líneas 5-6 se ejecutarán en cada iteración del algoritmo. Dada la lista de rangos A_2 = <10, 9, 8, 7, 6, 5, 4, 3, 2, 1>, se contratará un nuevo asistente de oficina solo una vez, en la primera iteración. Dada una lista de rangos A_3 = <5, 2, 1, 8, 4, 7, 10, 9, 3, 6>, se contratará un nuevo asistente de oficina tres veces, al entrevistar a los candidatos con los rangos 5, 8 y 10. Recordando que el costo de nuestro algoritmo depende de cuántas veces contratemos a un nuevo asistente de oficina, vemos que hay entradas caras, como A_1 , entradas baratas, como A_2 , y moderadamente insumos costosos, como A_3 .

Considere, por otro lado, el algoritmo aleatorio que primero permuta los candidatos y luego determina el mejor candidato. En este caso, la aleatorización está en el algoritmo, no en la distribución de entrada. Dada una entrada en particular, digamos A_3 arriba, no podemos decir cuántas veces el máximo se actualizará, porque esta cantidad difiere con cada ejecución del algoritmo. La primera vez que ejecutamos el algoritmo en A_3 , puede producir la permutación A_1 y realizar 10 actualizaciones, mientras que la segunda vez que ejecutamos el algoritmo, podemos producir la permutación A_2 y

realice solo una actualización. La tercera vez que lo ejecutamos, podamos realizar algún otro número de actualizaciones. Cada vez que ejecutamos el algoritmo, la ejecución depende de las elecciones aleatorias realizadas. y es probable que difiera de la ejecución anterior del algoritmo. Para este algoritmo y muchos otros algoritmos aleatorios, *ninguna entrada en particular provoca su comportamiento en el peor de los casos* . Incluso tu peor enemigo no puede producir una mala matriz de entrada, ya que la permutación aleatoria hace que el orden de entrada irrelevante. El algoritmo aleatorio funciona mal solo si el número aleatorio generador produce una permutación "desafortunada".

Para el problema de contratación, el único cambio necesario en el código es permutar aleatoriamente la matriz.

```
ASISTENTE DE CONTRATACIÓN ALEATORIA ( n )
1 permutar aleatoriamente la lista de candidatos
2 mejor ← 0 → candidato 0 es un candidato ficticio menos calificado
3 para i ← 1 a n
4 do entrevista candidato i
5                   si el candidato i es mejor que el candidato mejor
6                       entonces mejor ← i
7                                   contratar candidato i
```

Con este simple cambio, hemos creado un algoritmo aleatorio cuyo rendimiento coincide con el obtenido asumiendo que los candidatos se presentaron en un orden aleatorio.

Lema 5.3

El costo de contratación esperado del procedimiento ALEATORIA-CONTRATACIÓN-ASISTENTE es $O (c_h \ln n)$.

Página 92

Prueba Después de permutar la matriz de entrada, hemos logrado una situación idéntica a la del análisis probabilístico de HIRE-ASSISTANT.

La comparación entre los Lemas 5.2 y 5.3 captura la diferencia entre pruebas probabilísticas análisis y algoritmos aleatorizados. En el Lema 5.2 , hacemos una suposición sobre la entrada. En el Lema 5.3 , no hacemos tal suposición, aunque aleatorizar la entrada requiere tiempo adicional. En el resto de esta sección, discutimos algunos problemas relacionados con permutando entradas.

Matrices de permutación aleatoria

Muchos algoritmos aleatorios aleatorizan la entrada permutando la matriz de entrada dada. (Ahí Hay otras formas de usar la aleatorización.) Aquí, discutiremos dos métodos para hacerlo. Nosotros Supongamos que se nos da una matriz A que, sin pérdida de generalidad, contiene los elementos 1 hasta n . Nuestro objetivo es producir una permutación aleatoria de la matriz.

Un método común es asignar a cada elemento $A [i]$ de la matriz una prioridad aleatoria $P [i]$, y luego clasifique los elementos de A *de* acuerdo con estas prioridades. Por ejemplo, si nuestra matriz inicial es $A = \langle 1, 2, 3, 4 \rangle$ y elegimos prioridades aleatorias $P = \langle 36, 3, 97, 19 \rangle$, produciríamos una matriz $B = \langle 2, 4, 1, 3 \rangle$, ya que la segunda prioridad es la más pequeña, seguida de la cuarta, luego la primera, y finalmente el tercero. A este procedimiento lo llamamos PERMUTO POR CLASIFICACIÓN:

```
PERMUTO POR CLASIFICACIÓN ( A )
1 n ← longitud [ A ]
2 para i ← 1 an
3 hacer P [ i ] = ALEATORIO (1, n 3 )
4 ordenar A , usando P como teclas de ordenación
5 devuelve A
```

La línea 3 elige un número aleatorio entre 1 y n^3 . Usamos un rango de 1 an^3 para que sea probable que todas las prioridades en P son únicas. (El ejercicio 5.3-5 le pide que pruebe que la probabilidad que todas las entradas son únicas es al menos 1 - 1 / n , y el ejercicio 5.3-6 pregunta cómo implementar el algoritmo incluso si dos o más prioridades son idénticas). Supongamos que todas las prioridades son único.

El paso que lleva mucho tiempo en este procedimiento es la clasificación en la línea 4. Como veremos en el capítulo 8 , si usamos una clasificación de comparación, la clasificación toma $\Omega (n \lg n)$ tiempo. Podemos lograr este límite inferior,

ya que hemos visto que la ordenación por fusión toma $\Theta(n \lg n)$ tiempo. (Veremos otros tipos de comparación que toman $\Theta(n \lg n)$ tiempo en la Parte II.) Después de ordenar, si $P[i]$ es la j-ésima prioridad más pequeña, entonces $A[i]$ estará en la posición j de la salida. De esta manera obtenemos una permutación. Queda por probar que el procedimiento produce una **permutación aleatoria uniforme**, es decir, que toda permutación de es igualmente probable que se produzcan los números del 1 al n.

Lema 5.4

El procedimiento PERMUTE-BY-SORTING produce una permutación aleatoria uniforme de la entrada, asumiendo que todas las prioridades son distintas.

Página 93

Prueba Comenzamos por considerar la permutación particular en la que cada elemento $A[i]$ recibe la i ésima prioridad más pequeña. Demostraremos que esta permutación ocurre con probabilidad exactamente $1/n!$. Para $i = 1, 2, ..., n$, dejar que X_i ser el caso de que el elemento $A[i]$ recibe el i ésimo más pequeño prioridad. Entonces deseamos calcular la probabilidad de que para todo i, ocurra el evento X_i, que es

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\}.$$

Usando el ejercicio C.2-6, esta probabilidad es igual a

$$\Pr\{X_1\} \cdot \Pr\{X_2 \mid X_1\} \cdot \Pr\{X_3 \mid X_2 \cap X_1\} \cdot \Pr\{X_4 \mid X_3 \cap X_2 \cap X_1\}$$

$$\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \Pr\{X_n \mid X_{n-1} \cap \cdots \cap X_1\}.$$

Tenemos que $\Pr\{X_1\} = 1/n$ porque es la probabilidad de que una prioridad se elija al azar de un conjunto de n es el más pequeño. A continuación, observamos que $\Pr\{X_2 \mid X_1\} = 1/(n-1)$ porque dado que elemento $A[1]$ tiene la menor prioridad, cada uno de los $n-1$ elementos restantes tiene una igual posibilidad de tener la segunda prioridad más pequeña. En general, para $i = 2, 3, ..., n$, tenemos que $\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} = 1/(n-i+1)$, ya que, dado que los elementos $A[1]$ a $A[i-1]$ tienen las $i-1$ prioridades más pequeñas (en orden), cada uno de los $n-(i-1)$ elementos restantes tiene un igual posibilidad de tener la i-ésima prioridad más pequeña. Por lo tanto, tenemos

y hemos demostrado que la probabilidad de obtener la permutación de identidad es $1/n!$.

Podemos extender esta prueba para que funcione para cualquier permutación de prioridades. Considere cualquier fijo permutación $\sigma = \langle \sigma(1), \sigma(2), ..., \sigma(n) \rangle$ del conjunto $\{1, 2, ..., n\}$. Denotemos por r_i el rango de la prioridad asignada al elemento $A[i]$, donde el elemento con la j-ésima prioridad más pequeña tiene rango j. Si definimos X_i como el evento en el que el elemento $A[i]$ recibe la $\sigma(i)$ ésima prioridad más pequeña, o $r_i = \sigma(i)$, se sigue aplicando la misma prueba. Por tanto, si calculamos la probabilidad de obtener alguna permutación particular, el cálculo es idéntico al anterior, por lo que la probabilidad de obtener esta permutación también es $1/n!$.

Se podría pensar que para demostrar que una permutación es una permutación aleatoria uniforme es suficiente para mostrar que, para cada elemento $A[i]$, la probabilidad de que termine en la posición j es $1/n$. El ejercicio 5.3-4 muestra que esta condición más débil es, de hecho, insuficiente.

Un mejor método para generar una permutación aleatoria es permutar la matriz dada en su lugar. El procedimiento RANDOMIZE-IN-PLACE lo hace en tiempo $O(n)$. En la iteración i, el elemento $A[i]$ se elige al azar entre los elementos $A[i]$ a $A[n]$. Posterior a la iteración i, $A[i]$ es nunca alterado.

```
ALEATORIZAR EN SU LUGAR ( A )
1 n ← longitud [ A ]
2 para i ← a n
```

Página 94

3 **hacen** de intercambio $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$

Usaremos un ciclo invariante para mostrar que el procedimiento RANDOMIZE-IN-PLACE produce un permutación aleatoria uniforme. Dado un conjunto de n elementos, una k -permutación es una secuencia que contiene k de los n elementos. (Consulte el Apéndice B). ¡Hay $n! / (N - k)!$ tan posible k - permutaciones.

Lema 5.5

El procedimiento RANDOMIZE-IN-PLACE calcula una permutación aleatoria uniforme.

Prueba Usamos el siguiente ciclo invariante:

- Justo antes de la i- ésima iteración del bucle **for** de las líneas 2-3, para cada posible $(i - 1)$ - permutación, el subarreglo $A[1 .. i - 1]$ contiene esta $(i - 1)$ -permutación con probabilidad $(n - i + 1)! / n!$.

Necesitamos mostrar que este invariante es verdadero antes de la primera iteración del ciclo, que cada iteración de el bucle mantiene el invariante, y que el invariante proporciona una propiedad útil para mostrar corrección cuando el bucle termina.

- **Inicialización:** considere la situación justo antes de la primera iteración del ciclo, de modo que $i = 1$. El invariante de bucle dice que para cada posible permutación 0, el subarreglo $A[1 .. 0]$ contiene esta permutación 0 con probabilidad $(n - i + 1)! / n! = n! / n! = 1$. El subarreglo $Un[1 .. 0]$ es un subarreglo vacío y una permutación 0 no tiene elementos. Por tanto, $A[1 .. 0]$ contiene cualquier permutación 0 con probabilidad 1, y el invariante de bucle se mantiene antes de la primera iteración.
- **Mantenimiento:** Suponemos que justo antes de la $(i - 1)$ a iteración, cada posible $(i - 1)$ - La permutación aparece en el subarreglo $A[1 .. i - 1]$ con probabilidad $(n - i + 1)! / n!$, y mostrará que después de la i- ésima iteración, cada posible i -permutación aparece en la subarreglo $A[1 .. i]$ con probabilidad $(n - i)! / n!$. Incrementar i para la siguiente iteración luego mantenga el ciclo invariante.

  Examinemos la i- ésima iteración. Considere una permutación i particular , y denote el elementos en él por $< x_1, x_2, ..., x_i >$. Esta permutación consiste en una $(i - 1)$ -permutación $< x_1, ..., x_{i-1} >$ seguido del valor x_i que el algoritmo coloca en $A[i]$. Dejar que E_1 denotan el evento en el que las primeras $i - 1$ iteraciones han creado el particular $(i - 1)$ - permutación $< x_1, ..., x_{i-1} >$ en $A[1 .. i - 1]$. Por el ciclo invariante, $\Pr\{E_1\} = (n - i + 1)! / N!$. Sea E_2 el evento de que i- ésima iteración ponga x_i en la posición $A[i]$. La i -permutación $< x_1, ..., x_i >$ se forma en $A[1 .. i]$ precisamente cuando ocurren tanto E_1 como E_2 , por lo que deseamos calcular $\Pr\{E_2 \cap E_1\}$. Usando la ecuación (C.14) , tenemos

  $$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\}.$$

  La probabilidad $\Pr\{E_2 \mid E_1\}$ es igual a $1 / (n - i + 1)$ porque en la línea 3 el algoritmo elige x_i aleatoriamente de los valores $n - i + 1$ en las posiciones $A[i .. n]$. Por lo tanto, tenemos

Página 95

- **Terminación:** En la terminación, $i = n + 1$, y tenemos que el subarreglo $A[1 .. n]$ es un dada n -permutación con probabilidad $(n - n)! / n! = 1 / n!$.

Por tanto, RANDOMIZE-IN-PLACE produce una permutación aleatoria uniforme.

Un algoritmo aleatorio suele ser la forma más sencilla y eficaz de resolver un problema. Nosotros utilizará algoritmos aleatorios ocasionalmente a lo largo de este libro.

Ejercicios 5.3-1

El profesor Marceau se opone al invariante de bucle utilizado en la demostración del Lema 5.5 . El cuestiona si es cierto antes de la primera iteración. Su razonamiento es que uno podría fácilmente declare que un subarreglo vacío no contiene permutaciones 0. Por tanto, la probabilidad de que un el subarreglo vacío contiene una permutación 0 que debe ser 0, invalidando así el invariante del ciclo antes de la primera iteración. Reescriba el procedimiento RANDOMIZE-IN-PLACE para que su el invariante de bucle asociado se aplica a un subarreglo no vacío antes de la primera iteración, y modifique la prueba de Lemma 5.5 para su procedimiento.

Ejercicios 5.3-2

El profesor Kelp decide escribir un procedimiento que producirá al azar cualquier permutación además de la permutación de identidad. Propone el siguiente procedimiento:

```
PERMUTO-SIN-IDENTIDAD ( A )
1 n ← longitud [ A ]
2 para i ← 1 a n
3 hacen de intercambio A [ i ] ↔ A [RANDOM ( i + 1, n )]
```

¿Este código hace lo que pretende el profesor Kelp?

Ejercicios 5.3-3

Suponga que en lugar de intercambiar el elemento $A[i]$ con un elemento aleatorio del subarreglo $A[i..n]$, lo intercambiamos con un elemento aleatorio desde cualquier lugar de la matriz:

Página 96

```
PERMUTE-CON-TODOS ( A )
1 n ← longitud [ A ]
2 para i ← 1 a n
3 hacen de intercambio A [ i ] ↔ A [RANDOM (1, n )]
```

¿Este código produce una permutación aleatoria uniforme? ¿Por qué o por qué no?

Ejercicios 5.3-4

El profesor Armstrong sugiere el siguiente procedimiento para generar un uniforme aleatorio permutación:

```
PERMUTO POR CÍCLICO ( A )
1 n ← longitud [ A ]
2 desplazamiento ← RANDOM (1, n )
3 para i ← 1 a n
4 do dest ← i + desplazamiento
5                 si dest > n
6                         luego dest ← dest - n
7                 B [ dest ] ← A [ i ]
8 vuelve B
```

Demuestre que cada elemento $A[i]$ tiene una probabilidad de $1/n$ de terminar en cualquier posición particular en B. Luego demuestre que el profesor Armstrong se equivoca al mostrar que la permutación resultante no es uniformemente aleatorio.

Ejercicios 5.3-5:

Demuestre que en la matriz P en el procedimiento PERMUTE-POR-CLASIFICACIÓN, la probabilidad de que todos los elementos son únicos es al menos $1 - 1/n$.

Ejercicios 5.3-6

Explique cómo implementar el algoritmo PERMUTE-BY-SORTING para manejar el caso en cuáles dos o más prioridades son idénticas. Es decir, su algoritmo debe producir un uniforme permutación aleatoria, incluso si dos o más prioridades son idénticas.

5.4 Análisis probabilístico y usos posteriores del indicador variables aleatorias

Esta sección avanzada ilustra aún más el análisis probabilístico a través de cuatro ejemplos. los primero determina la probabilidad de que en una habitación de k personas, alguna pareja comparta el mismo cumpleaños.

El segundo ejemplo examina el lanzamiento aleatorio de bolas en contenedores. El tercero investiga "Rayas" de caras consecutivas al lanzar una moneda. El ejemplo final analiza una variante del problema de contratación en el que hay que tomar decisiones sin entrevistar a todos los candidatos.

5.4.1 La paradoja del cumpleaños

Nuestro primer ejemplo es la *paradoja del cumpleaños*. ¿Cuántas personas debe haber en una habitación antes ¿Existe un 50% de probabilidad de que dos de ellos hayan nacido el mismo día del año? La respuesta es sorprendentemente pocos. La paradoja es que, de hecho, es mucho menor que el número de días en un año, o incluso la mitad del número de días en un año, como veremos.

Para responder a esta pregunta, indexamos a las personas en la habitación con los números enteros 1, 2, ..., k, donde k es el número de personas en la habitación. Ignoramos el tema de los años bisiestos y asumimos que todos los años tienen $n = 365$ días. Para $i = 1, 2, ..., k$, y mucho b_i ser el día del año en el que la persona i 's cumpleaños cae, donde $1 \le b_i \le n$. También asumimos que los cumpleaños se distribuyen uniformemente a lo largo de los n días del año, de modo que $\Pr\{b_i = r\} = 1/n$ para $i = 1, 2, ..., k$ y $r = 1, 2, ..., n$.

La probabilidad de que dos personas dadas, digamos i y j, tengan cumpleaños iguales depende de si la selección aleatoria de cumpleaños es independiente. Asumimos a partir de ahora que los cumpleaños son independientes, de modo que la probabilidad de que el cumpleaños de i y el cumpleaños de j caigan en el día r es

$$\begin{aligned}\Pr\{b_i = r \text{ y } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= 1/n^2 .\end{aligned}$$

Por tanto, la probabilidad de que ambos caigan el mismo día es

(5,7)

De manera más intuitiva, una vez que se elige b_i, la probabilidad de que b_j se elija para el mismo día es $1/n$. Por tanto, la probabilidad de que i y j tengan el mismo cumpleaños es la misma que la probabilidad de que

el cumpleaños de uno de ellos cae en un día determinado. Note, sin embargo, que esta coincidencia depende en el supuesto de que los cumpleaños sean independientes.

Podemos analizar la probabilidad de que al menos 2 de cada k personas tengan cumpleaños coincidentes mediante mirando el evento complementario. La probabilidad de que al menos dos de los cumpleaños coincidan es 1 menos la probabilidad de que todos los cumpleaños sean diferentes. El evento que tienen k personas distintos cumpleaños es

Página 98

donde A_i es el evento de que el cumpleaños de la persona i es diferente del de la persona j para todo $j < i$. Desde que nosotros podemos escribir $B_k = A_k \cap B_{k-1}$, obtenemos de la ecuación (C.16) la recurrencia

(5,8)

donde tomamos Pr $\{ B_1 \} =$ Pr $\{ A_1 \} = 1$ como condición inicial. En otras palabras, la probabilidad de que $b_1 , b_2 , ..., b_k$ son cumpleaños distintos es la probabilidad de que $b_1 , b_2 , ..., b_{k-1}$ sean cumpleaños distintos multiplicada por la probabilidad de que $b_k \neq b_i$ para $i = 1, 2, ..., k - 1$, dado que $b_1 , b_2 , ..., b_{k-1}$ son distintos.

Si $b_1 , b_2 , ..., b_{k-1}$ son distintos, la probabilidad condicional de que $b_k \neq b_i$ para $i = 1, 2, ..., k - 1$ es Pr $\{ A_k \mid B_{k-1} \} = (n - k + 1) / n$, ya que de los n días, hay $n - (k - 1)$ que no se toman. Nosotros aplicar iterativamente la recurrencia (5.8) para obtener

Desigualdad (3.11), $1 + x \leq e^x$, nos da

cuando $- k (k - 1) / 2 n \leq \ln (1/2)$. La probabilidad de que todos los k cumpleaños sean distintos es como máximo 1/2 cuando $k (k - 1) = 2 n \ln 2$ o, resolviendo la ecuación cuadrática, cuando . Para $n = 365$, debemos tener $k \geq 23$. Por lo tanto, si al menos 23 personas están en una habitación, la probabilidad es al menos 1/2 que al menos dos personas tengan el mismo cumpleaños. En Marte, un año tiene 669 días marcianos; eso por lo tanto, se necesitan 31 marcianos para obtener el mismo efecto.

Un análisis que utiliza variables aleatorias de indicador

Podemos utilizar variables aleatorias del indicador para proporcionar un análisis más simple pero aproximado de la paradoja del cumpleaños. Para cada par (i , j) de las k personas en la sala, definimos el indicador variable aleatoria X_{ij} , para $1 \leq i < j \leq k$, por

Por la ecuación (5.7) , la probabilidad de que dos personas tengan cumpleaños iguales es $1 / n$, y por lo tanto por Lema 5.1 , tenemos

Página 99

$E[X_{ij}] = \Pr\{\text{la persona } i \text{ y la persona } j \text{ tienen el mismo cumpleaños}\}$
$= 1/n .$

Sea X la variable aleatoria que cuenta el número de pares de individuos que tienen la mismo cumpleaños, tenemos

Tomando las expectativas de ambos lados y aplicando la linealidad de la expectativa, obtenemos

Cuando $k(k-1) \geq 2n$, por lo tanto, el número esperado de parejas de personas con el mismo cumpleaños es al menos 1. Por lo tanto, si tenemos al menos individuos en una habitación, podemos esperar al menos dos tener el mismo cumpleaños. Para $n = 365$, si $k = 28$, el número esperado de pares con el mismo cumpleaños es $(28 \cdot 27)/(2 \cdot 365) \approx 1.0356$.

Por lo tanto, con al menos 28 personas, esperamos encontrar al menos un par de días de nacimiento coincidentes. En Marte, donde un año tiene 669 días marcianos, necesitamos al menos 38 marcianos.

El primer análisis, que utilizó solo probabilidades, determinó el número de personas necesarias para que la probabilidad exceda la mitad de que exista un par de cumpleaños coincidentes, y el segundo El análisis, que utilizó variables aleatorias del indicador, determinó el número el número esperado de cumpleaños coincidentes es 1. Aunque el número exacto de personas difiere para las dos situaciones, son iguales asintóticamente: .

5.4.2 Balones y cubos

Considere el proceso de lanzar al azar bolas idénticas en b contenedores, numerados 1, 2, ..., b. los Los lanzamientos son independientes, y en cada lanzamiento es igualmente probable que la pelota termine en cualquier recipiente. los La probabilidad de que una bola lanzada caiga en cualquier recipiente es $1/b$. Por tanto, el proceso de lanzar la pelota es un secuencia de ensayos de Bernoulli (ver Apéndice C.4) con una probabilidad de éxito de $1/b$, donde El éxito significa que la bola cae en el contenedor dado. Este modelo es particularmente útil para analizando hash (ver Capítulo 11), y podemos responder una variedad de preguntas interesantes sobre el proceso de lanzamiento de la pelota. (El problema C-1 hace preguntas adicionales sobre pelotas y cubos).

¿Cuántas bolas caen en un recipiente determinado? El número de bolas que caen en un contenedor determinado sigue el distribución binomial $b(k; n, 1/b)$. Si se lanzan n bolas, la ecuación (C.36) nos dice que el El número esperado de bolas que caen en el contenedor dado es n/b.

Página 100

¿Cuántas bolas se deben lanzar, en promedio, hasta que un recipiente determinado contenga una bola? El número de lanzamientos hasta que el recipiente dado recibe una pelota sigue la distribución geométrica con probabilidad $1/b$, según la ecuación (C.31) , el número esperado de lanzamientos hasta el éxito es $1/(1/b) = b$.

¿Cuántas bolas debe lanzar una hasta que cada recipiente contenga al menos una bola? Llamemos un lanzamiento

que una bola caiga en un recipiente vacío un "golpe". Queremos saber el número esperado n de lanzamientos requerido para obtener b hits.

Los golpes se pueden utilizar para dividir los n lanzamientos en etapas. La i ésima etapa consiste en los lanzamientos después del (i - 1) st hit hasta el i th hit. La primera etapa consiste en el primer lanzamiento, ya que estamos garantizado para tener un éxito cuando todos los contenedores están vacíos. Para cada lanzamiento durante la i- ésima etapa, hay i - 1 bins que contienen bolas y b - i + 1 bins vacíos. Por lo tanto, para cada lanzamiento en la i- ésima etapa, el la probabilidad de obtener un acierto es (b - i +1) / b .

Sea n_i el número de lanzamientos en la i- ésima etapa. Por lo tanto, el número de lanzamientos necesarios para obtener b hits es . Cada variable aleatoria n_i tiene una distribución geométrica con probabilidad de éxito (b - i + 1) / b y, por la ecuación (C.31) ,

Por linealidad de expectativa,

La última línea se sigue del límite (A.7) de la serie armónica. Por lo tanto, se necesita aproximadamente b ln b lanzamientos antes de que podamos esperar que cada recipiente tenga una bola. Este problema es también conocido como el problema del cobrador de cupones , y dice que una persona que intenta cobrar cada uno de b cupones diferentes deben adquirir aproximadamente b ln b cupones obtenidos al azar para poder tener éxito.

5.4.3 Rayas

Suponga que lanza una moneda justa n veces. ¿Cuál es la racha más larga de cabezas consecutivas que esperar ver? La respuesta es $\Theta(\lg n)$, como muestra el siguiente análisis.

Primero probamos que la longitud esperada de la racha más larga de cabezas es $O(\lg n)$. los La probabilidad de que cada lanzamiento de moneda sea una cara es 1/2. Sea A_{ik} el evento de que una racha de cabezas de longitud al menos k comienza con el i- ésimo lanzamiento de moneda o, más precisamente, el evento de que el k

Página 101

lanzamientos consecutivos de monedas i , i + 1, ..., i + k - 1 dan solo cara, donde $1 \le k \le n$ y $1 \le i \le n$ - k +1. Dado que los lanzamientos de moneda son mutuamente independientes, para cualquier evento dado A_{ik} , la probabilidad de que k volteretas son cabezas es

(5,9)

y así la probabilidad de que una racha de cabezas de al menos $2\lceil \lg n \rceil$ comience en la posición i es bastante pequeño. Hay como máximo n - $2\lceil \lg n \rceil$ + 1 posiciones donde puede comenzar tal racha. los Por tanto, la probabilidad de que una racha de cabezas de al menos $2\lceil \lg n \rceil$ comience en cualquier lugar es

(5,10)

ya que por la desigualdad de Boole (C.18), la probabilidad de una unión de eventos es como máximo la suma de las probabilidades de los eventos individuales. (Tenga en cuenta que la desigualdad de Boole es válida incluso para eventos como estos que no son independientes.)

Ahora usamos la desigualdad (5.10) para acotar la longitud de la racha más larga. Para $j = 0, 1, 2, ..., n$, sea L_j sea el evento de que la racha más larga de cabezas tenga una longitud exactamente j , y sea L la longitud de la racha más larga. Según la definición de valor esperado,

(5,11)

Podríamos intentar evaluar esta suma usando límites superiores en cada Pr $\{ L_j \}$ similares a los calculado en desigualdad (5.10). Desafortunadamente, este método arrojaría límites débiles. Podemos Sin embargo, use algo de intuición obtenida por el análisis anterior para obtener un buen límite. Informalmente observamos que para ningún término individual en la suma de la ecuación (5.11) son ambos los factores j y Pr $\{ L_j \}$ grandes. ¿Por qué? Cuando $j \geq 2 \lceil \lg n \rceil$, entonces Pr $\{ L_j \}$ es muy pequeño, y cuando $j <$ $2 \lceil \lg n \rceil$, entonces j es bastante pequeño. Más formalmente, observamos que los eventos L_j para $j = 0, 1, ..., n$ son disjuntos, por lo que la probabilidad de que una racha de cabezas de al menos $2 \lceil \lg n \rceil$ comience dondequiera es . Por desigualdad (5.10), tenemos . Además, notando que , tenemos eso . Así obtenemos

Página 102

Las posibilidades de que una racha de caras supere $r \lceil \lg n \rceil$ flips disminuyen rápidamente con r . Para $r \geq 1$, la probabilidad de que una racha de $r \lceil \lg n \rceil$ caras comience en la posición i es

$$\Pr \{ A_{i, r \lceil \lg n \rceil} \} = 1/2^{r \lceil \lg n \rceil} \leq 1 / n^r .$$

Por lo tanto, la probabilidad es como máximo $n / n^r = 1 / n^{r-1}$ de que la racha más larga sea al menos $r \lceil \lg n \rceil$, o de manera equivalente, la probabilidad es al menos $1 - 1 / n^{r-1}$ de que la racha más larga tenga una longitud menor que $r \lceil \text{Lg } n \rceil$.

Como ejemplo, para $n = 1000$ monedas lanzadas, la probabilidad de tener una racha de al menos $2 \lceil \lg n \rceil =$ 20 cabezas es como máximo $1 / n = 1/1000$. Las posibilidades de tener una racha superior a $3 \lceil \lg n \rceil = 30$ cabezas es como máximo $1 / n^2 = 1 / 1.000.000$.

Ahora demostramos un límite inferior complementario: la longitud esperada de la racha más larga de caras en n lanzamientos de moneda es $\Omega (\lg n)$. Para probar este límite, buscamos rayas de longitud s por dividir los n voltea en aproximadamente n / s grupos de s cada uno. Si elegimos $s = \lfloor (\lg n) / 2 \rfloor$, podemos demostrar que es probable que al menos uno de estos grupos aparezca en todas las cabezas, y

por tanto, es probable que la racha más larga tenga una longitud de al menos $s = \Omega(\lg n)$. Entonces mostraremos que la racha más larga tiene una longitud esperada $\Omega(\lg n)$.

Dividimos los n lanzamientos de monedas en al menos $\lfloor n / \lfloor (\lg n)/2 \rfloor \rfloor$ grupos de $\lfloor (\lg n)/2 \rfloor$ consecutivos voltea, y enlazamos la probabilidad de que ningún grupo salga con todas las caras. Por la ecuación (5.9) , el La probabilidad de que el grupo que comienza en la posición i salga todas cabezas es

$$\Pr\{A_{i,\lfloor (\lg n)/2 \rfloor}\} = 1/2^{\lfloor (\lg n)/2 \rfloor} \geq .$$

La probabilidad de que una racha de cabezas de al menos $\lfloor (\lg n)/2 \rfloor$ no comience en la posición i es por tanto como mucho . Dado que los grupos $\lfloor n / \lfloor (\lg n)/2 \rfloor \rfloor$ se forman a partir de

Página 103

lanzamientos de moneda exclusivos e independientes, la probabilidad de que cada uno de estos grupos *no* sea un racha de longitud $\lfloor (\lg n)/2 \rfloor$ es como máximo

Para este argumento, usamos la desigualdad (3.11), $1 + x \leq e^x$, y el hecho, que es posible que desee para verificar, que para n suficientemente grande .

Por tanto, la probabilidad de que la racha más larga supere $\lfloor (\lg n)/2 \rfloor$ es

(5,12)

Ahora podemos calcular un límite inferior en la longitud esperada de la racha más larga, comenzando con la ecuación (5.11) y procediendo de una manera similar a nuestro análisis del límite superior:

Al igual que con la paradoja del cumpleaños, podemos obtener un análisis más simple pero aproximado usando Indicador de variables aleatorias. Dejamos que $X_{ik} = I\{A_{ik}\}$ sea la variable aleatoria indicadora asociada con una racha de caras de al menos k *que* comienzan con el i- ésimo lanzamiento de moneda. Para contar el total número de tales rayas, definimos

Tomando expectativas y usando la linealidad de la expectativa, tenemos

Página 104

Al introducir varios valores para k , podemos calcular el número esperado de rayas de longitud k . Si este número es grande (mucho mayor que 1), entonces se espera que muchas rayas de longitud k ocurrir y la probabilidad de que ocurra uno es alta. Si este número es pequeño (mucho menos de 1), entonces se espera que ocurran muy pocas rayas de longitud k y la probabilidad de que ocurra una es bajo. Si $k = c \lg n$, para alguna constante positiva c , obtenemos

Si c es grande, el número esperado de rayas de longitud $c \lg n$ es muy pequeño, y concluimos que es poco probable que ocurran. Por otro lado, si $c < 1/2$, obtenemos $E[X] = \Theta(1 / n^{1/2-1}) = \Theta(n^{1/2})$, y esperamos que haya una gran cantidad de rayas de longitud $(1/2) \lg n$. Por lo tanto, es muy probable que ocurra una racha de tal longitud. De estas estimaciones aproximadas por sí solos, podemos concluir que la longitud de la racha más larga es $\Theta(\lg n)$.

5.4.4 El problema de la contratación en línea

Como ejemplo final, consideramos una variante del problema de contratación. Supongamos ahora que no Deseo entrevistar a todos los candidatos para encontrar el mejor. Tampoco deseamos contratar y disparar a medida que encontremos mejores y mejores candidatos. En cambio, estamos dispuestos a conformarnos con un candidato que está cerca de los mejores, a cambio de contratar exactamente una vez. Debemos obedecer a una empresa requisito: después de cada entrevista debemos ofrecer inmediatamente el puesto al solicitante o debe decirles que no recibirán el trabajo. ¿Cuál es la compensación entre minimizando la cantidad de entrevistas y maximizando la calidad del candidato contratado?

Podemos modelar este problema de la siguiente manera. Después de conocer a un solicitante, podemos dar a cada uno una puntuación; deje que la *puntuación* (i) denote la puntuación otorgada al i- ésimo solicitante, y suponga que no hay dos solicitantes que reciban la misma puntuación. Después de haber visto j solicitantes, sabemos cuál de la j tiene la puntuación más alta, pero no sabemos si alguno de los $n - j$ solicitantes restantes tener una puntuación más alta. Decidimos adoptar la estrategia de seleccionar un entero positivo $k < n$, entrevistar y luego rechazar a los primeros k solicitantes, y contratar al primer solicitante a partir de entonces que tiene una puntuación más alta que todos los solicitantes anteriores. Si resulta que el mejor calificado

Página 105

solicitante fue de los primeros k entrevistado, a continuación, vamos a contratar el n º solicitante. Esta estrategia se formaliza en el procedimiento ON-LINE-MAXIMUM (k , n), que aparece a continuación. Procedimiento ON-LINE-MAXIMUM devuelve el índice del candidato que deseamos contratar.

```
ON-LINE-MAXIMO ( k , n )
1 mejor puntuación ← -∞
2 para i ← a k
3 hacer si puntuación ( i )> mejor puntuación
4                     luego mejor puntuación ← puntuación ( i )
5 para i ← k + 1 a n
6 hacer si puntuación ( i )> mejor puntuación
7                     luego regrese yo
8 volver n
```

Deseamos determinar, para cada valor posible de k , la probabilidad de que contratemos más solicitante calificado. Luego elegiremos la mejor k posible e implementaremos la estrategia con ese valor. Por el momento, suponga que k es fijo. Sea $M (j) = \max_{\leq i \leq j} \{ score (i)\}$ denote el puntuación máxima entre los solicitantes del 1 al j . Sea S el evento que logramos elegir el solicitante mejor calificado, y dejemos que S_i sea el evento en el que tengamos éxito cuando el mejor calificado el solicitante es el i ésimo entrevistado. Dado que los diversos S_i son inconexos, tenemos que . Observando que nunca tenemos éxito cuando el solicitante mejor calificado es uno de los primero k , tenemos que $\Pr \{ S_i \} = 0$ para $i = 1, 2, ..., k$. Así obtenemos

(5,13)

Ahora calculamos $\Pr \{ S_i \}$. Para tener éxito cuando el solicitante mejor calificado es el i- ésimo, deben suceder dos cosas. Primero, el solicitante mejor calificado debe estar en la posición i , un evento que denotamos por B_i . En segundo lugar, el algoritmo no debe seleccionar a ninguno de los solicitantes en posiciones $k + 1$ a través de $i - 1$, lo que ocurre solo si, para cada j tal que $k + 1 \leq j \leq i - 1$, encontrar esa *puntuación* (j) <mejor *puntuación* en la línea 6. (Debido a que las puntuaciones son únicas, podemos ignorar la posibilidad de *puntuación* (j) = mejor *puntuación* .) En otras palabras, debe darse el caso de que todos los valores *la puntuación* ($k + 1$) hasta la *puntuación* ($i - 1$) son menores que $M (k)$; si alguno es mayor que $M (k)$ lo haremos en su lugar devuelve el índice del primero que sea mayor. Usamos O_i para denotar el evento de que ninguno de los solicitantes en las posiciones $k + 1$ a $i - 1$ son elegidos. Afortunadamente, los dos eventos B_i y *Oh,* $_{soy}$ independiente. El evento O_i depende solo del orden relativo de los valores en posiciones 1 a $i - 1$, mientras que B_i depende solo de si el valor en la posición i es mayor que todos los valores 1 a $i - 1$. El orden de las posiciones 1 a $i - 1$ no afecta si i es mayor que todos ellos, y el valor de i no afecta el orden de posiciones 1 a $i - 1$. Por tanto, podemos aplicar la ecuación (C.15) para obtener

$$\Pr \{ S_i \} = \Pr \{ B_i \cap O_i \} = \Pr \{ B_i \} \Pr \{ O_i \}.$$

La probabilidad $\Pr \{ B_i \}$ es claramente $1 / n$, ya que es igualmente probable que el máximo esté en cualquiera de las n posiciones. Para que ocurra el evento O_i , el valor máximo en las posiciones 1 a $i - 1$ debe ser en una de las primeras k posiciones, y es igualmente probable que esté en cualquiera de estas $i - 1$ posiciones. En consecuencia, $\Pr \{ O_i \} = k / (i - 1)$ y $\Pr \{ S_i \} = k / (n (i - 1))$. Usando la ecuación (5.13) , tenemos

Página 106

Aproximamos por integrales para unir esta suma desde arriba y desde abajo. Por el desigualdades (A.12), tenemos

Evaluar estas integrales definidas nos da los límites

que proporcionan un límite bastante estrecho para Pr { S }. Porque deseamos maximizar nuestra probabilidad del éxito, centrémonos en elegir el valor de k que maximice el límite inferior de Pr { S }. (Además, la expresión de límite inferior es más fácil de maximizar que la expresión de límite superior). Diferenciando la expresión (k / n) (ln n - ln k) con respecto a k , obtenemos

Al establecer esta derivada igual a 0, vemos que el límite inferior de la probabilidad se maximiza cuando ln k = ln n - 1 = ln (n / e) o, de manera equivalente, cuando $k = n / e$. Por lo tanto, si implementamos nuestra con $k = n / e$, lograremos contratar a nuestro candidato mejor calificado con probabilidad de al menos 1 / e .

Ejercicios 5.4-1

¿Cuántas personas debe haber en una habitación antes de que la probabilidad de que alguien tenga la misma cumpleaños como lo haces es al menos 1/2? ¿Cuántas personas debe haber antes de la probabilidad de que al menos dos personas cumplen años el 4 de julio es mayor que 1/2?

Ejercicios 5.4-2

Suponga que las bolas se lanzan a contenedores b . Cada lanzamiento es independiente y cada bola es igualmente es probable que termine en cualquier contenedor. ¿Cuál es el número esperado de lanzamientos de pelota antes de al menos uno de los contenedores contienen dos bolas?

Página 107

Ejercicios 5.4-3:

Para el análisis de la paradoja del cumpleaños, ¿es importante que los cumpleaños sean mutuamente independiente, o es suficiente la independencia por pares? Justifica tu respuesta.

Ejercicios 5.4-4:

¿Cuántas personas deben ser invitadas a una fiesta para que sea probable que haya *tres* personas con el mismo cumpleaños?

Ejercicios 5.4-5:

¿Cuál es la probabilidad de que una k- cadena sobre un conjunto de tamaño n sea en realidad una k -permutación? Cómo ¿Esta pregunta se relaciona con la paradoja del cumpleaños?

Ejercicios 5.4-6:

Suponga que se lanzan n bolas en n contenedores, donde cada lanzamiento es independiente y la bola es igualmente probable que terminen en cualquier contenedor. ¿Cuál es el número esperado de contenedores vacíos? Cuál es el número esperado de contenedores con exactamente una bola?

Ejercicios 5.4-7:

Afile el límite inferior de la longitud de la racha mostrando que en n lanzamientos de una moneda justa, el
La probabilidad es menor que $1 / n$ *de* que no se produzca una racha superior a $\lg n - 2 \lg \lg n$ caras consecutivas.

Problemas 5-1: conteo probabilístico

Con un contador de b bits, normalmente solo podemos contar hasta $2^b - 1$. Con el método probabilístico de R. Morris contando , podemos contar hasta un valor mucho mayor a expensas de cierta pérdida de precisión.

Página 108

Dejamos que un valor de contador de i represente una cuenta de n_i para $i = 0, 1, ..., 2^b - 1$, donde n_i forma una secuencia creciente de valores no negativos. Suponemos que el valor inicial del contador es 0, que representa un recuento de $n_0 = 0$. La operación INCREMENT funciona en un contador que contiene el valor i de forma probabilística. Si $i = 2^b - 1$, se informa de un error de desbordamiento. De lo contrario, el contador aumenta en 1 con probabilidad $1 / (n_{i+1} - n_i)$, y permanece sin cambios con probabilidad $1 - 1 / (n_{i+1} - n_i)$.

Si seleccionamos $n_i = i$ para todo $i \geq 0$, entonces el contador es ordinario. Situaciones más interesantes surgen si seleccionamos, digamos, $n_i = 2^{i-1}$ para $i > 0$ o $n_i = F_i$ (el i- ésimo número de Fibonacci; consulte la Sección 3.2).

Para este problema, suponga que es lo suficientemente grande como para que la probabilidad de un error de desbordamiento sea despreciable.

a. Muestre que el valor esperado representado por el contador después de n INCREMENTO operaciones realizadas es exactamente n .
si. El análisis de la varianza del recuento representado por el contador depende de la secuencia del n_i . Consideremos un caso simple: $n_i = 100\,i$ para todo $i \geq 0$. Estime el varianza en el valor representado por el registro después de que n operaciones INCREMENT hayan realizado.

Problemas 5-2: búsqueda de una matriz sin clasificar

Por lo tanto, el problema examina tres algoritmos para buscar un valor x en una matriz no ordenada A que consta de n elementos.

Considere la siguiente estrategia aleatorio: escoger al azar un índice i en *una* . Si $A[i] = x$, entonces Terminar; de lo contrario, continuamos la búsqueda seleccionando un nuevo índice de azar en *una* . Nosotros Continuar seleccionando índices aleatorios en A hasta que encontremos un índice j tal que $A[j] = xo$ hasta que sometidos al control de todos los elementos de A . Tenga en cuenta que elegimos de todo el conjunto de índices cada vez, para que podamos examinar un elemento dado más de una vez.

a. Escribir pseudocódigo para un procedimiento RANDOM-SEARCH para implementar la estrategia encima. Asegúrese de que su algoritmo finalice cuando todos los índices de A hayan sido escogido.
si. Suponga que hay exactamente un índice i tal que $A[i] = x$. Que es lo esperado número de índices en A que deben seleccionarse antes de que se encuentre x y RANDOM- ¿TERMINA LA BÚSQUEDA?
C. Generalizando su solución al inciso b), suponga que hay $k \geq 1$ índices i tales que $A[i] = x$. ¿Cuál es el número esperado de índices en A que deben seleccionarse antes de que x sea encontrado y RANDOM-SEARCH termina? Tu respuesta debe ser una función de n y k .
re. Suponga que no hay índices i tales que $A[i] = x$. ¿Cuál es el número esperado de índices en A que deben seleccionarse antes de que todos los elementos de A se hayan verificado y ¿TERMINA RANDOM-SEARCH?

Ahora considere un algoritmo de búsqueda lineal determinista, al que nos referimos como BÚSQUEDA DETERMINISTICA. Específicamente, el algoritmo de búsqueda A para x en orden,

Página 109

considerando A [1], A [2], A [3], ..., A [n] hasta que A [i] = x se encuentre o el final de la matriz sea alcanzado. Suponga que todas las posibles permutaciones de la matriz de entrada son igualmente probables.

mi. Suponga que hay exactamente un índice i tal que A [i] = x . Que es lo esperado tiempo de ejecución de DETERMINISTIC-SEARCH? ¿Cuál es el peor tiempo de ejecución de ¿BÚSQUEDA DETERMINISTICA?

F. Generalizando su solución al inciso e), suponga que hay $k \geq 1$ índices i tales que A [i] = x . ¿Cuál es el tiempo de ejecución esperado de DETERMINISTIC-SEARCH? Que es ¿Cuál es el peor tiempo de ejecución de DETERMINISTIC-SEARCH? Tu respuesta debería ser una función de n y k .

gramo. Suponga que no hay índices i tales que A [i] = x . Cual es la carrera esperada hora de BÚSQUEDA DETERMINISTICA ¿Cuál es el peor tiempo de ejecución de ¿BÚSQUEDA DETERMINISTICA?

Finalmente, considere un algoritmo aleatorio SCRAMBLE-SEARCH que funciona primero permutar aleatoriamente la matriz de entrada y luego ejecutar la búsqueda lineal determinista dada arriba en la matriz permutada resultante.

h. Si k es el número de índices i tales que A [i] = x , da el peor de los casos y tiempos de ejecución esperados de SCRAMBLE-SEARCH para los casos en los que $k = 0$ y $k = 1$. Generalice su solución para manejar el caso en el que $k \geq 1$.

yo. ¿Cuál de los tres algoritmos de búsqueda utilizaría? Explica tu respuesta.

Notas del capítulo

Bollobás [44] , Hofri [151] y Spencer [283] contienen una gran cantidad de pruebas probabilísticas avanzadas Técnicas Karp analiza y analiza las ventajas de los algoritmos aleatorios [174] y Rabin [253] . El libro de texto de Motwani y Raghavan [228] ofrece una extensa tratamiento de algoritmos aleatorizados.

Se han estudiado ampliamente varias variantes del problema de la contratación. Estos problemas son más comúnmente conocido como "problemas de secretaria". Un ejemplo de trabajo en esta área es el papel por Ajtai, Meggido y Waarts [12] .

Parte II: Clasificación y estadísticas de orden

Lista de capítulos

Capítulo 6: Heapsort
Capítulo 7: Clasificación rápida
Capítulo 8: Ordenación en tiempo lineal
Capítulo 9: Medianas y estadísticas de orden

Introducción

Esta parte presenta varios algoritmos que resuelven el siguiente problema de clasificación :

• **Entrada:** una secuencia de n números $a_1, a_2, ..., a_n$.

Página 110

• **Salida:** una permutación (reordenamiento) de la secuencia de entrada de modo que

La secuencia de entrada suele ser una matriz de n elementos, aunque puede estar representada en algunos de otra manera, como una lista enlazada.

La estructura de los datos

En la práctica, los números que se van a ordenar rara vez son valores aislados. Cada uno es generalmente parte de un recopilación de datos denominada **registro** . Cada registro contiene una **clave** , que es el valor a ordenar, y el resto del registro consiste en **datos satelitales** , que generalmente se transportan con la llave. En la práctica, cuando un algoritmo de clasificación permuta las claves, debe permutar el también datos de satélite. Si cada registro incluye una gran cantidad de datos de satélite, a menudo permutamos una matriz de punteros a los registros en lugar de los registros mismos para minimizar los datos movimiento.

En cierto sentido, son estos detalles de implementación los que distinguen un algoritmo de un programa. Si ordenamos números individuales o registros grandes que contienen números es irrelevante para el *método* por el cual un procedimiento de clasificación determina el orden clasificado. Así, cuando centrándonos en el problema de la clasificación, normalmente asumimos que la entrada consiste solo en números. La traducción de un algoritmo para clasificar números en un programa para clasificar registros es conceptualmente sencillo, aunque en una situación de ingeniería dada Hay otras sutilezas que hacen que la tarea de programación sea un desafío.

¿Por qué clasificar?

Muchos informáticos consideran que la clasificación es el problema más fundamental en el estudio de algoritmos. Hay varias razones:

- A veces, la necesidad de ordenar la información es inherente a una aplicación. Por ejemplo, en Para preparar los extractos de los clientes, los bancos deben clasificar los cheques por número de cheque.
- Los algoritmos suelen utilizar la clasificación como una subrutina clave. Por ejemplo, un programa que renderiza Los objetos gráficos que están superpuestos pueden tener que ordenar los objetos. de acuerdo con una relación "arriba" para que pueda dibujar estos objetos de abajo hacia arriba. Veremos numerosos algoritmos en este texto que utilizan la clasificación como una subrutina.
- Existe una amplia variedad de algoritmos de clasificación y utilizan un amplio conjunto de técnicas. En De hecho, muchas técnicas importantes utilizadas en el diseño de algoritmos están representadas en el cuerpo de algoritmos de clasificación que se han desarrollado a lo largo de los años. De este modo, la clasificación también es un problema de interés histórico.
- La clasificación es un problema para el que podemos probar un límite inferior no trivial (como haremos en el Capítulo 8). Nuestros mejores límites superiores coinciden con el límite inferior asintóticamente, por lo que sabemos que nuestros algoritmos de clasificación son asintóticamente óptimos. Además, podemos utilizar el límite inferior para la clasificación para demostrar límites inferiores para ciertos otros problemas.
- Muchos problemas de ingeniería pasan a primer plano al implementar algoritmos de clasificación. los El programa de clasificación más rápido para una situación particular puede depender de muchos factores, como conocimiento previo sobre las claves y los datos satelitales, la jerarquía de la memoria (cachés y memoria virtual) del equipo host y el entorno del software. Muchos de estos Los problemas se tratan mejor a nivel algorítmico, en lugar de "retocar" el código.

Ordenar algoritmos

Página 111

Introdujimos dos algoritmos que clasifican n números reales en el Capítulo 2 . La ordenación por inserción toma $\Theta(n^2)$ tiempo en el peor de los casos. Debido a que sus bucles internos son estrechos, sin embargo, es una clasificación rápida en el lugar algoritmo para pequeños tamaños de entrada. (Recuerde que un algoritmo de clasificación ordena **en su lugar** si solo un número constante de elementos de la matriz de entrada se almacenan fuera de la matriz). tiene un mejor tiempo de ejecución asintótico, $\Theta(n \lg n)$, pero el procedimiento MERGE que utiliza no operar en su lugar.

En esta parte, presentaremos dos algoritmos más que clasifican números reales arbitrarios. Heapsort presentado en el Capítulo 6 , ordena n números en su lugar en $O(n \lg n)$ tiempo. Utiliza datos importantes estructura, llamada heap, con la que también podemos implementar una cola de prioridad.

Quicksort, en el Capítulo 7 , también ordena n números en su lugar, pero su peor tiempo de ejecución es $\Theta(n^2)$. Sin embargo, su tiempo de ejecución promedio de caso es $\Theta(n \lg n)$, y generalmente supera heapsort en la práctica. Al igual que la ordenación por inserción, la ordenación rápida tiene un código ajustado, por lo que el factor constante oculto en su tiempo de ejecución es pequeño. Es un algoritmo popular para clasificar matrices de entrada grandes.

La ordenación por inserción, la ordenación por combinación, la ordenación en pila y la ordenación rápida son todas clases de comparación: determinan orden ordenado de una matriz de entrada comparando elementos. El capítulo 8 comienza presentando el modelo de árbol de decisiones para estudiar las limitaciones de rendimiento de los tipos de comparación. Utilizando este modelo, demostramos un límite inferior de $\Omega (n \lg n)$ en el peor tiempo de ejecución de cualquier ordenación por comparación en n entradas, lo que muestra que la ordenación en pila y la ordenación por combinación son asintóticamente tipos de comparación óptimos.

El capítulo 8 continúa mostrando que podemos superar este límite inferior de $\Omega (n \lg n)$ si podemos recopilar información sobre el orden de clasificación de la entrada por medios distintos a la comparación elementos. El algoritmo de clasificación de conteo, por ejemplo, asume que los números de entrada están en el establecer $\{1, 2, ..., k \}$. Mediante el uso de la indexación de matrices como herramienta para determinar el orden relativo, la clasificación de conteo puede ordenar n números en $\Theta (k + n)$ tiempo. Por lo tanto, cuando $k = O (n)$, la ordenación de conteo se ejecuta en un tiempo que es lineal en el tamaño de la matriz de entrada. Se puede utilizar un algoritmo relacionado, la ordenación por base, para ampliar la rango de tipo de conteo. Si hay n enteros para ordenar, cada entero tiene d dígitos y cada dígito está en el conjunto $\{1, 2, ..., k \}$, luego radix sort puede ordenar los números en $\Theta (d (n + k))$ tiempo. Cuando d es un constante y k es $O (n)$, la ordenación por radix se ejecuta en tiempo lineal. Un tercer algoritmo, la clasificación de cubos, requiere conocimiento de la distribución probabilística de números en la matriz de entrada. Puede ordenar n real números distribuidos uniformemente en el intervalo semiabierto [0, 1) en el tiempo $O (n)$ del caso medio .

Estadísticas de pedidos

La estadística de i- ésimo orden de un conjunto de n números es el i- ésimo número más pequeño del conjunto. Uno puede, de Por supuesto, seleccione la estadística de i- ésimo orden ordenando la entrada e indexando el elemento i- ésimo de la salida. Sin suposiciones sobre la distribución de entrada, este método se ejecuta en un tiempo $\Omega (n \lg n)$, como muestra el límite inferior en el capítulo 8 .

En el Capítulo 9 , mostramos que podemos encontrar el i- ésimo elemento más pequeño en $O (n)$ tiempo, incluso cuando el los elementos son números reales arbitrarios. Presentamos un algoritmo con pseudocódigo ajustado que se ejecuta en tiempo $\Theta (n^2)$ en el peor de los casos, pero tiempo lineal en promedio. También damos una más complicada algoritmo que se ejecuta en el peor de los casos $O (n)$.

Antecedentes

Página 112

Aunque la mayor parte de esta parte no se basa en matemáticas difíciles, algunas secciones requieren sofisticación matemática. En particular, los análisis de casos promedio de clasificación rápida, clasificación de cubos, y el algoritmo estadístico de orden usa probabilidad, que se revisa en el Apéndice C , y el material sobre análisis probabilístico y algoritmos aleatorios en el Capítulo 5 . El análisis de la El algoritmo de tiempo lineal del peor de los casos para las estadísticas de pedidos implica algo más sofisticado matemáticas que los otros análisis del peor de los casos en esta parte.

Capítulo 6: Heapsort

Visión general

En este capítulo, presentamos otro algoritmo de clasificación. Como combinación de ordenación, pero a diferencia de la inserción sort, el tiempo de ejecución de heapsort es $O (n \lg n)$. Como la ordenación por inserción, pero a diferencia de la ordenación por combinación, heapsort ordena en su lugar: solo un número constante de elementos de la matriz se almacenan fuera de la matriz de entrada en en cualquier momento. Por lo tanto, heapsort combina los mejores atributos de los dos algoritmos de clasificación que tenemos ya discutido.

Heapsort también introduce otra técnica de diseño de algoritmos: el uso de una estructura de datos, en a este caso lo llamamos "montón", para gestionar la información durante la ejecución del algoritmo. La estructura de datos del montón no solo es útil para heapsort, sino que también constituye una prioridad eficiente cola. La estructura de datos del montón reaparecerá en algoritmos en capítulos posteriores.

Observamos que el término "montón" se acuñó originalmente en el contexto de heapsort, pero desde entonces vienen a referirse a "almacenamiento recolectado de basura", como los lenguajes de programación Lisp y Proporcionar Java. Nuestra estructura de datos de pila *no* es un almacenamiento de basura recolectado, y cada vez que hacemos referencia A montones en este libro, nos referiremos a la estructura definida en este capítulo.

.1 montones

La estructura de datos del montón (binario) es un objeto de matriz que puede verse como un árbol binario (consulte la Sección B.5.3), como se muestra en la Figura 6.1 . Cada nodo del árbol corresponde a un elemento de la matriz que almacena el valor en el nodo. El árbol está completamente lleno en todos niveles excepto posiblemente el más bajo, que se llena desde la izquierda hasta un punto. Una matriz A que representa un montón es un objeto con dos atributos: *longitud* [A], que es el número de elementos en la matriz, y *el montón de tamaño* [A], el número de elementos en el montón almacenado dentro de la matriz A . Ese es decir, aunque A [1 *longitud* [A]] puede contener números válidos, ningún elemento después de A [*tamaño de pila* [A]], donde *tamaño de montón* [A] ≤ *longitud* [A], es un elemento del montón. La raíz del árbol es A [1], y dado el índice i de un nodo, los índices de su padre padre (i), hijo izquierdo IZQUIERDO (i) y derecho child RIGHT (i) se puede calcular simplemente:

```
PADRE ( i )
    volver ⌊ i / 2⌋

IZQUIERDA ( i )
    volver 2 i

DERECHA ( i )
    devuelve 2 i + 1
```

Página 113

Figura 6.1: Un montón máximo visto como *(a)* un árbol binario y *(b)* una matriz. El número dentro del el círculo en cada nodo del árbol es el valor almacenado en ese nodo. El número sobre un nodo es el índice correspondiente en la matriz. Arriba y abajo de la matriz hay líneas que muestran padre-hijo relaciones; los padres siempre están a la izquierda de sus hijos. El árbol tiene una altura de tres; el el nodo en el índice 4 (con valor 8) tiene una altura.

En la mayoría de las computadoras, el procedimiento IZQUIERDO puede calcular 2 i en una instrucción simplemente cambiando la representación binaria de i dejó una posición de bit. Del mismo modo, el procedimiento DERECHO puede Calcule rápidamente 2 i + 1 cambiando la representación binaria de i *a la* izquierda una posición de bit y agregando un 1 como el bit de orden inferior. El procedimiento PADRE puede calcular ⌊ i / 2⌋ desplazando i posición de un bit a la derecha. En una buena implementación de heapsort, estos tres procedimientos son a menudo implementado como procedimientos "macros" o "en línea".

Hay dos tipos de montones binarios: max-montones y min-montones. En ambos tipos, los valores en los nodos satisfacen una propiedad del montón , cuyos detalles dependen del tipo de montón. en un max-heap , la propiedad max-heap es que para cada nodo i *que* no sea la raíz,

$$A[\text{PADRE}(i)] \geq A[i],$$

es decir, el valor de un nodo es como máximo el valor de su padre. Así, el elemento más grande en un max-heap se almacena en la raíz, y el subárbol enraizado en un nodo contiene valores no mayores que el contenido en el propio nodo. Un min-heap se organiza de forma opuesta; el min-montón La propiedad es que para cada nodo i *que* no sea la raíz,

$$A[\text{PADRE}(i)] \leq A[i].$$

El elemento más pequeño de un min-heap está en la raíz.

Para el algoritmo heapsort, usamos max-heaps. Los min-montones se usan comúnmente en prioridad colas, que discutimos en la Sección 6.5 . Seremos precisos al especificar si necesitamos un max-heap o min-heap para cualquier aplicación en particular, y cuando las propiedades se aplican a max-montones o min-montones, simplemente usamos el término "montón".

Al ver un montón como un árbol, definimos la altura de un nodo en un montón como el número de bordes en el camino descendente simple más largo desde el nodo hasta una hoja, y definimos la altura del montón para ser la altura de su raíz. Dado que un montón de n elementos se basa en un árbol binario completo, su altura es Θ (lg n) (vea el ejercicio 6.1-2). Veremos que las operaciones básicas sobre montones se ejecutan en el tiempo como máximo proporcional a la altura del árbol y, por lo tanto, toma O (lg n) tiempo. El resto de este capítulo presenta cinco procedimientos básicos y muestra cómo se utilizan en una clasificación algoritmo y una estructura de datos de cola de prioridad.

- El procedimiento MAX-HEAPIFY, que se ejecuta en tiempo $O(\lg n)$, es la clave para mantener la propiedad max-heap.
- El procedimiento BUILD-MAX-HEAP, que se ejecuta en tiempo lineal, produce un montón máximo de una matriz de entrada desordenada.

Página 114

- El procedimiento HEAPSORT, que se ejecuta en tiempo $O(n \lg n)$, ordena una matriz en su lugar.
- El MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY y
  Los procedimientos HEAP-MAXIMUM, que se ejecutan en tiempo $O(\lg n)$, permiten que los datos del montón estructura que se utilizará como cola de prioridad.

Ejercicios 6.1-1

¿Cuáles son los números mínimo y máximo de elementos en un montón de altura h ?

Ejercicios 6.1-2

Demuestre que un montón de n elementos tiene una altura $\lfloor \lg n \rfloor$.

Ejercicios 6.1-3

Muestre que en cualquier subárbol de un montón máximo, la raíz del subárbol contiene el valor más grande ocurriendo en cualquier lugar de ese subárbol.

Ejercicios 6.1-4

Dónde en un montón máximo podría residir el elemento más pequeño, asumiendo que todos los elementos son ¿distinto?

Ejercicios 6.1-5

¿Es una matriz que está en orden ordenado un min-heap?

Ejercicios 6.1-6

¿Es la secuencia 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 un montón máximo?

Ejercicios 6.1-7

Página 115

Muestre que, con la representación de matriz para almacenar un montón de n elementos, las hojas son las nodos indexados por $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Mantenimiento de la propiedad del montón

MAX-HEAPIFY es una subrutina importante para manipular max-montones. Sus entradas son un matriz A y un índice i en la matriz. Cuando se llama a MAX-HEAPIFY, se supone que el Los árboles binarios enraizados en IZQUIERDA (i) y DERECHA (i) son montones máximos, pero que $A[i]$ puede ser más pequeño que sus hijos, violando así la propiedad max-heap. La función de MAX-HEAPIFY es dejar el valor en $A[i]$ "flota hacia abajo" en el montón máximo de modo que el subárbol enraizado en el índice i *se* convierte en un max-heap.

```
MAX-HEAPIFY ( A , i )
 1 l ← IZQUIERDA ( i )
 2 r ← DERECHA ( i )
 3 si l ≤ tamaño de pila [ A ] y A [ l ]> A [ i ]
 4 luego más grande ← l
 5 más grande ← i
 6 si r ≤ tamaño de pila [ A ] y A [ r ]> A [ mayor ]
 7 luego más grande ← r
 8 si es más grande ≠ i
 9 luego intercambie A [ i ] ↔ A [ más grande ]
10                MAX-HEAPIFY ( A , más grande )
```

La figura 6.2 ilustra la acción de MAX-HEAPIFY. En cada paso, el mayor de los elementos *Se determina* $A[i]$, A [IZQUIERDA (i)] y A [DERECHA (i)], y su índice se almacena en el *mayor* . Si $A[i]$ es más grande, entonces el subárbol enraizado en el nodo i es un montón máximo y el procedimiento termina. De lo contrario, uno de los dos hijos tiene el elemento más grande y $A[i]$ se intercambia con A [*más grande*], que hace que el nodo i *y* sus hijos satisfagan la propiedad max-heap. El nodo indexado por *mayor* , sin embargo, ahora tiene el valor original $A[i]$, y por lo tanto el subárbol tiene su raíz en *más grande* puede violar la propiedad max-heap. En consecuencia, MAX-HEAPIFY debe llamarse recursivamente en ese subárbol.

Figura 6.2: La acción de MAX-HEAPIFY (A , 2), donde *heap-size* $[A] = 10$. *(a)* La inicial configuración, con A [2] en el nodo $i = 2$ violando la propiedad max-heap ya que no es más grande que ambos niños. La propiedad max-heap se restaura para el nodo 2 en *(b)* intercambiando A [2]

Página 116

con A [4], que destruye la propiedad max-heap para el nodo 4. La llamada recursiva MAX-HEAPIFY (A , 4) ahora tiene $i = 4$. Después de intercambiar A [4] con A [9], como se muestra en *(c)* , el nodo 4 es arreglado, y la llamada recursiva MAX-HEAPIFY (A , 9) no produce más cambios en los datos estructura.

El tiempo de ejecución de MAX-HEAPIFY en un subárbol de tamaño n arraigado en el nodo i dado es el $\Theta(1)$ tiempo para arreglar las relaciones entre los elementos $A[i]$, A [IZQUIERDA (i)] y A [DERECHA (i)], más el tiempo para ejecutar MAX-HEAPIFY en un subárbol enraizado en uno de los hijos del nodo i . los los subárboles de los niños tienen un tamaño máximo de $2n/3$; el peor de los casos ocurre cuando la última fila del El árbol está exactamente medio lleno y, por lo tanto, se puede describir el tiempo de ejecución de MAX-HEAPIFY por la recurrencia

$$T(n) \le T(2n/3) + \Theta(1).$$

La solución a esta recurrencia, por el caso 2 del teorema maestro (Teorema 4.1), es $T(n) = O(\lg n)$. Alternativamente, podemos caracterizar el tiempo de ejecución de MAX-HEAPIFY en un nodo de altura h como $O(h)$.

Ejercicios 6.2-1

Usando la Figura 6.2 como modelo, ilustre la operación de MAX-HEAPIFY (A , 3) en el arreglo A = 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0.

Ejercicios 6.2-2

Comenzando con el procedimiento MAX-HEAPIFY, escriba el pseudocódigo para el procedimiento MIN-HEAPIFY (A , i), que realiza la manipulación correspondiente en un min-heap. Cómo el tiempo de ejecución de MIN-HEAPIFY en comparación con el de MAX-HEAPIFY?

Ejercicios 6.2-3

¿Cuál es el efecto de llamar a MAX-HEAPIFY (A , i) cuando el elemento $A[i]$ es mayor que su ¿niños?

Ejercicios 6.2-4

¿Cuál es el efecto de llamar a MAX-HEAPIFY (A , i) para $i > heap\text{-}size[A]/2$?

Página 117

Ejercicios 6.2-5

El código para MAX-HEAPIFY es bastante eficiente en términos de factores constantes, excepto posiblemente para la llamada recursiva en la línea 10, lo que puede hacer que algunos compiladores produzcan código. Escriba un MAX-HEAPIFY eficiente que utilice una construcción de control iterativa (un bucle) en lugar de recursividad.

Ejercicios 6.2-6

Muestre que el peor tiempo de ejecución de MAX-HEAPIFY en un montón de tamaño n es $\Omega(\lg n)$. (*Sugerencia:* para un montón con n nodos, proporcione valores de nodo que hagan que se llame a MAX-HEAPIFY recursivamente en cada nodo en un camino desde la raíz hasta la hoja).

6.3 Construyendo un montón

Podemos usar el procedimiento MAX-HEAPIFY de abajo hacia arriba para convertir una matriz $A[1 \ldots n]$, donde $n = longitud[A]$, en un montón máximo. Mediante el ejercicio 6.1-7 , los elementos del subarreglo $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ son todas las hojas del árbol, por lo que cada una es un montón de 1 elemento para empezar. los El procedimiento BUILD-MAX-HEAP pasa por los nodos restantes del árbol y ejecuta MAX-HEAPIFY en cada uno.

```
CONSTRUIR-MAX-HEAP ( A )
1 tamaño de pila [ A ] ← longitud [ A ]
2 para i ← ⌊ longitud [ A ] / 2⌋ hacia abajo a 1
```

3 **hacer** MAX-HEAPIFY (*A* , *i*)

La figura 6.3 muestra un ejemplo de la acción de BUILD-MAX-HEAP.

Página 118

Figura 6.3: El funcionamiento de BUILD-MAX-HEAP, que muestra la estructura de datos antes de la llamada a MAX-HEAPIFY en la línea 3 de BUILD-MAX-HEAP. *(a)* A 10-elemento de array entrada *A* y la árbol binario que representa. La figura muestra que el índice de bucle *i se* refiere al nodo 5 antes de la llamada MAX-HEAPIFY (*A, i*). *(b)* La estructura de datos resultante. El índice de bucle *i* para el siguiente iteración se refiere al nodo 4. *(c)* - *(e)* iteraciones posteriores del bucle *for* en BUILD-MAX-MONTÓN. Observe que siempre que se llama a MAX-HEAPIFY en un nodo, los dos subárboles de ese node son max-montones. *(f)* El montón máximo después de que finaliza BUILD-MAX-HEAP.

Para mostrar por qué BUILD-MAX-HEAP funciona correctamente, usamos el siguiente ciclo invariante:

- Al comienzo de cada iteración del bucle **for** de las líneas 2-3, cada nodo $i + 1, i + 2, \ldots, n$ es la raíz de un montón máximo.

Necesitamos mostrar que este invariante es verdadero antes de la primera iteración del ciclo, que cada iteración de el bucle mantiene el invariante, y que el invariante proporciona una propiedad útil para mostrar corrección cuando el bucle termina.

- **Inicialización:** antes de la primera iteración del ciclo, $i = \lfloor n / 2 \rfloor$. Cada nodo $\lfloor n / 2 \rfloor + 1, \lfloor n / 2 \rfloor + 2, \ldots, n$ es una hoja y, por lo tanto, es la raíz de un max-heap trivial.
- **Mantenimiento:** Para ver que cada iteración mantiene invariante el ciclo, observe que el los hijos del nodo *i* se numeran más alto que *i* . Por el ciclo invariante, por lo tanto,

son ambas raíces de max-montones. Esta es precisamente la condición requerida para la llamada MAX-HEAPIFY (A , i) para convertir el nodo i *en* una raíz de montón máximo. Además, la llamada MAX-HEAPIFY conserva la propiedad de que los nodos $i + 1$, $i + 2$,. . . , n son todas raíces de max-montones. Decrementar i en el **de** actualización del lazo restablece el invariante de bucle para la próxima iteración.

- **Terminación:** En la terminación, $i = 0$. Por el ciclo invariante, cada nodo 1, 2,. . . , n es la raíz de un montón máximo. En particular, el nodo 1 es.

Página 119

Podemos calcular un límite superior simple en el tiempo de ejecución de BUILD-MAX-HEAP como sigue. Cada llamada a MAX-HEAPIFY cuesta O (lg n) tiempo, y hay O (n) de este tipo. Así, el tiempo de ejecución es O (n lg n). Este límite superior, aunque correcto, no es asintóticamente estrecho.

Podemos derivar un límite más estricto al observar que el tiempo para que MAX-HEAPIFY se ejecute en un nodo varía con la altura del nodo en el árbol y las alturas de la mayoría de los nodos son pequeñas. Nuestra

Un análisis más estricto se basa en las propiedades que tiene un montón de n elementos de altura $\lfloor \lg n \rfloor$ (ver Ejercicio 6.1-2) y como máximo $\lceil n / 2^{h+1} \rceil$ nodos de cualquier altura h (vea el ejercicio 6.3-3).

El tiempo requerido por MAX-HEAPIFY cuando se llama a un nodo de altura h es O (h), por lo que podemos expresar el costo total de BUILD-MAX-HEAP como

La última suma se puede evaluar sustituyendo $x = 1/2$ en la fórmula (A.8), que rendimientos

Por lo tanto, el tiempo de ejecución de BUILD-MAX-HEAP se puede limitar como

Por lo tanto, podemos construir un montón máximo a partir de una matriz desordenada en tiempo lineal.

Podemos construir un min-heap mediante el procedimiento BUILD-MIN-HEAP, que es lo mismo que BUILD-MAX-HEAP pero con la llamada a MAX-HEAPIFY en la línea 3 reemplazada por una llamada a MIN-HEAPIFY (vea el ejercicio 6.2-2). BUILD-MIN-HEAP produce un min-montón a partir de un matriz lineal desordenada en tiempo lineal.

Ejercicios 6.3-1

Utilizando la Figura 6.3 como modelo, ilustre la operación de BUILD-MAX-HEAP en el arreglo A = 5, 3, 17, 10, 84, 19, 6, 22, 9.

Ejercicios 6.3-2

¿Por qué queremos que el índice de bucle i en la línea 2 de BUILD-MAX-HEAP disminuya de $\lfloor$ *longitud* [A] / 2$\rfloor$ a 1 en lugar de aumentar de 1 a $\lfloor$ *longitud* [A] / 2$\rfloor$?

Página 120

Ejercicios 6.3-3

Demuestre que hay como máximo $\lceil n / 2^{h+1} \rceil$ nodos de altura h en cualquier montón de n elementos.

6.4 El algoritmo heapsort

El algoritmo de heapsort comienza usando BUILD-MAX-HEAP para construir un montón máximo en la entrada matriz A [1 n], donde n = *longitud* [A]. Dado que el elemento máximo de la matriz se almacena en el raíz A [1], se puede poner en su posición final correcta intercambiándola con A [n]. Si ahora "descartar" el nodo n del montón (al disminuir *el tamaño del montón* [A]), observamos que A [1 (n - 1)] se puede convertir fácilmente en un montón máximo. Los hijos de la raíz siguen siendo max-montones, pero el el nuevo elemento raíz puede violar la propiedad max-heap. Todo lo que se necesita para restaurar el máximo La propiedad del montón, sin embargo, es una llamada a MAX-HEAPIFY (A , 1), que deja un montón máximo en A [1 (n - 1)]. El algoritmo de clasificación de pilas repite este proceso para el montón máximo de tamaño n - 1 hasta un montón de tamaño 2. (Consulte el ejercicio 6.4-2 para obtener un invariante de bucle preciso).

```
HEAPSORT ( A )
1 BUILD-MAX-HEAP ( A )
2 para i ← longitud [ A ] hacia abajo a 2
3 hacer intercambio A [1] ↔ A [ i ]
4 tamaño de pila [ A ] ← tamaño de pila [ A ] - 1
5 MAX-HEAPIFY ( A , 1)
```

La figura 6.4 muestra un ejemplo de la operación de heapsort después de que se construye inicialmente max-heap. Cada max-heap se muestra al comienzo de una iteración del ciclo **for** de las líneas 2-5.

Figura 6.4: El funcionamiento de HEAPSORT. *(a)* La estructura de datos max-heap justo después de ha sido construido por BUILD-MAX-HEAP. *(b)* - *(j)* Max-heap justo después de cada llamada de MAX-HEAPIFY en la línea 5. Se muestra el valor de i en ese momento. Solo los nodos ligeramente sombreados permanecen en el montón. *(k)* La resultante Ordenada matriz A .

Página 121

El procedimiento HEAPSORT toma tiempo O (n lg n), ya que la llamada a BUILD-MAX-HEAP toma tiempo O (n) y cada una de las n - 1 llamadas a MAX-HEAPIFY toma el tiempo O (lg n).

Ejercicios 6.4-1

Utilizando la Figura 6.4 como modelo, ilustre la operación de HEAPSORT en el arreglo A = 5, 13, 2, 25, 7, 17, 20, 8, 4.

Ejercicios 6.4-2

Argumentar la corrección de HEAPSORT utilizando el siguiente invariante de bucle:

- Al comienzo de cada iteración del ciclo **for** de las líneas 2-5, el subarreglo A [1 i] es un max-heap que contiene los i elementos más pequeños de A [1 n], y el subarreglo A [i + 1 n] contiene los n - i elementos más grandes de A [1 n], ordenados.

Ejercicios 6.4-3

¿Cuál es el tiempo de ejecución de heapsort en una matriz A de longitud n que ya está ordenada en orden creciente? ¿Y el orden decreciente?

Ejercicios 6.4-4

Muestre que el peor tiempo de ejecución de heapsort es Ω (n lg n).

Ejercicios 6.4-5:

Muestre que cuando todos los elementos son distintos, el mejor tiempo de ejecución de heapsort es Ω (n lg n).

6.5 Colas de prioridad

Heapsort es un algoritmo excelente, pero una buena implementación de quicksort, presentado en El capítulo 7 suele superarlo en la práctica. Sin embargo, la estructura de datos del montón en sí enorme utilidad. En esta sección, presentamos una de las aplicaciones más populares de un montón: su

Página 122

utilizar como una cola de prioridad eficiente. Al igual que con los montones, hay dos tipos de colas de prioridad: máx. colas de prioridad y colas de prioridad mínima. Nos centraremos aquí en cómo implementar el máximo colas de prioridad, que a su vez se basan en max-montones; El ejercicio 6.5-3 le pide que escriba procedimientos para colas de prioridad mínima.

Una *cola de prioridad* es una estructura de datos para mantener un conjunto S de elementos, cada uno con un valor asociado llamado *clave* . Una *cola de prioridad máxima* admite las siguientes operaciones.

- INSERT (S , x) inserta el elemento de x en el conjunto S . Esta operación podría escribirse como S $\leftarrow S \{ x \}$.
- MAXIMUM (S) devuelve el elemento de S con la clave más grande.
- EXTRACT-MAX (S) elimina y devuelve el elemento de S con la clave más grande.
- INCREASE-KEY (S , x , k) aumenta el valor de la clave del elemento x al nuevo valor k , que se supone que es al menos tan grande como el valor de clave actual de x .

Una aplicación de las colas de máxima prioridad es programar trabajos en una computadora compartida. El máximo- La cola de prioridad realiza un seguimiento de los trabajos a realizar y sus prioridades relativas. Cuando un trabajo finaliza o se interrumpe, se selecciona el trabajo de mayor prioridad entre los que están pendientes EXTRACTO-MAX. Se puede agregar un nuevo trabajo a la cola en cualquier momento usando INSERT.

Alternativamente, una *cola de prioridad mínima* admite las operaciones INSERT, MINIMUM, EXTRACT-MIN y DISMINUIR-TECLA. Se puede utilizar una cola de prioridad mínima en un evento simulador impulsado. Los elementos de la cola son eventos que se van a simular, cada uno con un tiempo de ocurrencia que le sirve como clave. Los eventos deben ser simulados en orden de su tiempo. de ocurrencia, porque la simulación de un evento puede causar que otros eventos sean simulados en el futuro. El programa de simulación utiliza EXTRACT-MIN en cada paso para elegir el siguiente evento para simular. A medida que se producen nuevos eventos, se insertan en la cola de prioridad mínima

utilizando INSERT. Veremos otros usos para las colas de prioridad mínima, destacando las
Operación DISMINUCIÓN-LLAVE, en los Capítulos 23 y 24 .

No es sorprendente que podamos usar un montón para implementar una cola de prioridad. En una aplicación dada, como la programación de trabajos o la simulación basada en eventos, los elementos de una cola de prioridad corresponden a objetos en la aplicación. A menudo es necesario determinar qué objeto de aplicación corresponde a un elemento de cola de prioridad dado, y viceversa. Cuando un montón se usa para implementar una cola de prioridad, por lo tanto, a menudo necesitamos almacenar un identificador en el correspondiente objeto de aplicación en cada elemento del montón. La composición exacta del mango (es decir, un puntero, un entero, etc.) depende de la aplicación. Del mismo modo, necesitamos almacenar un identificador en el elemento de montón correspondiente en cada objeto de aplicación. Aquí, el asa normalmente sería un índice de matriz. Debido a que los elementos del montón cambian de ubicación dentro de la matriz durante las operaciones del montón, una implementación real, al reubicar un elemento de montón, también tendría que actualizar el índice de matriz en el objeto de aplicación correspondiente. Porque los detalles de acceder los objetos de la aplicación dependen en gran medida de la aplicación y su implementación, no seguirlos aquí, además de señalar que, en la práctica, estos identificadores deben estar correctamente mantenido.

Ahora discutimos cómo implementar las operaciones de una cola de máxima prioridad. El procedimiento HEAP-MAXIMUM implementa la operación MAXIMUM en $\Theta(1)$ tiempo.

```
HEAP-MAXIMUM ( A )
1 devuelve A [1]
```

Página 123

El procedimiento HEAP-EXTRACT-MAX implementa la operación EXTRACT-MAX. Es similar al cuerpo del bucle **for** (líneas 3-5) del procedimiento HEAPSORT.

```
HEAP-EXTRACT-MAX ( A )
1 si el tamaño del montón [ A ] <1
2 luego error "subdesbordamiento del montón"
3 máx. ← A [1]
4 A [1] ← A [ tamaño de pila [ A ]]
5 tamaño de pila [ A ] ← tamaño de pila [ A ] - 1
6 MAX-HEAPIFY ( A , 1)
7 retorno máximo
```

El tiempo de ejecución de HEAP-EXTRACT-MAX es $O(\lg n)$, ya que solo realiza una constante cantidad de trabajo además del tiempo $O(\lg n)$ para MAX-HEAPIFY.

El procedimiento HEAP-INCREASE-KEY implementa la operación INCREASE-KEY. los El elemento de cola de prioridad cuya clave se va a aumentar se identifica mediante un índice i en la matriz. El procedimiento primero actualiza la clave del elemento $A[i]$ a su nuevo valor. Porque aumentando el clave de $A[i]$ puede violar la propiedad max-heap, el procedimiento entonces, de una manera que recuerda a el bucle de inserción (líneas 5-7) de INSERTION-SORT de la Sección 2.1 , atraviesa una ruta desde este nodo hacia la raíz para encontrar un lugar adecuado para la clave recién aumentada. Durante esto transversal, compara repetidamente un elemento con su padre, intercambiando sus claves y continúa si la clave del elemento es más grande y termina si la clave del elemento es más pequeña, ya que la propiedad max-heap ahora se mantiene. (Consulte el ejercicio 6.5-5 para obtener un invariante de bucle preciso).

```
TECLA DE AUMENTO DE HEAP ( tecla A , i , )
1 si tecla < A [ i ]
2 entonces el error "la clave nueva es más pequeña que la clave actual"
3 Tecla A [ i ] ←
4 mientras que i > 1 y A [PADRE ( i )] < A [ i ]
5 do intercambio A [ i ] ↔ A [PADRE ( i )]
6                 i ← PADRE ( i )
```

La Figura 6.5 muestra un ejemplo de una operación HEAP-INCREASE-KEY. El tiempo de ejecución de HEAP-INCREASE-KEY en un montón de n elementos es $O(\lg n)$, ya que la ruta trazada desde el nodo actualizado en la línea 3 a la raíz tiene longitud $O(\lg n)$.

Figura 6.5: El funcionamiento de HEAP-INCREASE-KEY. *(a)* El montón máximo de la Figura 6.4 (a) con un nodo cuyo índice es *i* fuertemente sombreadas. *(b)* Este nodo tiene su clave aumentada a 15. *(c)* Después de una iteración del *tiempo* de bucle de las líneas 4-6, el nodo y su padre han intercambiado claves,

Página 124

y el índice *i* sube hasta el padre. *(d)* Max-heap después de una iteración más del *while* lazo. En este punto, A [PADRE (i)] ≥ A [i]. La propiedad max-heap ahora se mantiene y el finaliza el procedimiento.

El procedimiento MAX-HEAP-INSERT implementa la operación INSERT. Toma como entrada la clave del nuevo elemento a insertar en max-heap A . El procedimiento primero expande el max-heap agregando al árbol una nueva hoja cuya clave es -∞. Luego llama HEAP-INCREASE-KEY para establecer la clave de este nuevo nodo en su valor correcto y mantener la propiedad max-heap.

```
MAX-HEAP-INSERT ( A , tecla )
1 tamaño de pila [ A ] ← tamaño de pila [ A ] + 1
2 A [ tamaño de pila [ A ]] ← -∞
3 TECLA DE INCREMENTO DE HEAP ( A , tamaño de pila [ A ], clave )
```

El tiempo de ejecución de MAX-HEAP-INSERT en un montón de n elementos es O (lg n).

En resumen, un montón puede admitir cualquier operación de cola de prioridad en un conjunto de tamaño n en O (lg n) hora.

Ejercicios 6.5-1

Ilustre la operación de HEAP-EXTRACT-MAX en el montón A = 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1.

Ejercicios 6.5-2

Ilustre la operación de MAX-HEAP-INSERT (A , 10) en el montón A = 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1. Utilice el montón de la Figura 6.5 como modelo para la llamada HEAP-INCREASE-KEY.

Ejercicios 6.5-3

Escriba un pseudocódigo para los procedimientos HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY y MIN-HEAP-INSERT que implementan una cola de prioridad mínima con un min-heap.

Ejercicios 6.5-4

¿Por qué nos molestamos en configurar la clave del nodo insertado en -∞ en la línea 2 de MAX-HEAP-INSERTAR cuando lo siguiente que hacemos es aumentar su clave al valor deseado?

Página 125

Ejercicios 6.5-5

Argumente la exactitud de HEAP-INCREASE-KEY utilizando el siguiente invariante de bucle:

- Al comienzo de cada iteración del **tiempo** de bucle de las líneas 4-6, la matriz A [1 *heap-size* [A]] satisface la propiedad max-heap, excepto que puede haber una violación: A [i] puede ser mayor que A [PADRE (i)].

Ejercicios 6.5-6

Muestre cómo implementar una cola de primero en entrar, primero en salir con una cola de prioridad. Muestre cómo implementar una pila con una cola de prioridad. (Las colas y las pilas se definen en la Sección 10.1) .

Ejercicios 6.5-7

La operación de lixiviación en DELETE (A , i) elimina el elemento en el nodo i del montón A . Dar un implementación de HEAP-DELETE que se ejecuta en tiempo O (lg n) para un n- elemento max-heap.

Ejercicios 6.5-8

Dar un algoritmo de tiempo O (n lg k) para fusionar k listas ordenadas en una lista ordenada, donde n es el número total de elementos en todas las listas de entrada. (*Sugerencia:* use un min-heap para la combinación de k -way).

Problemas 6-1: creación de un montón mediante inserción

El procedimiento BUILD-MAX-HEAP en la Sección 6.3 se puede implementar usando repetidamente MAX-HEAP-INSERT para insertar los elementos en el montón. Considera lo siguiente implementación:

```
CONSTRUIR-MAX-HEAP '( A )
1 tamaño de pila [ A ] ← 1
2 para i ← 2 hasta la longitud [ A ]
3 hacer MAX-HEAP-INSERT ( A , A [ i ])
```

a. ¿Los procedimientos BUILD-MAX-HEAP y BUILD-MAX-HEAP 'siempre crean el mismo montón cuando se ejecuta en la misma matriz de entrada? Demuestre que lo hacen, o proporcione una contraejemplo.

Página 126

si. Muestre que en el peor de los casos, BUILD-MAX-HEAP 'requiere Θ (n lg n) tiempo para construir un montón de n elementos.

Problemas 6-2: Análisis de montones d-arios

Un montón d-ary es como un montón binario, pero (con una posible excepción) los nodos que no son hojas tienen d niños en lugar de 2 niños.

a. ¿Cómo representaría un montón d -ary en una matriz?
si. ¿Cuál es la altura de un d ary montón de n elementos en términos de n y d ?
C. Proporcione una implementación eficiente de EXTRACT-MAX en un d -ary max-heap. Analizar su tiempo de ejecución en términos de d y n .
re. Proporcione una implementación eficiente de INSERT en un d -ary max-heap. Analizar su funcionamiento tiempo en términos de d y n .
mi. Dar una implementación eficiente de INCREASE-KEY (A , i , k), que primero establece $A [i] \leftarrow$ max ($A [i]$, k) y luego actualiza la estructura d -ary max-heap apropiadamente. Analizar su tiempo de funcionamiento en términos de d y n .

Problemas 6-3: cuadros jóvenes

Un cuadro joven de $m \times n$ es una matriz de $m \times n$ tal que las entradas de cada fila están ordenadas orden de izquierda a derecha y las entradas de cada columna están ordenadas de arriba a abajo. Algunas de las entradas de un cuadro de Young pueden ser ∞, que tratamos como elementos inexistentes. Por lo tanto, se puede usar un cuadro de Young para contener $r \le mn$ números finitos.

a. Dibuja un cuadro de Young de 4 × 4 que contenga los elementos {9, 16, 3, 2, 4, 8, 5, 14, 12}.
si. Argumenta que un cuadro de Young $m \times n$ Y está vacío si $Y [1, 1] = \infty$. Argumenta que Y está lleno (contiene mn elementos) si $Y [m , n] < \infty$.
C. Dar un algoritmo para implementar EXTRACT-MIN en un $m \times n$ Young no vacío cuadro que se ejecuta en tiempo $O (m + n)$. Tu algoritmo debería usar una subrutina recursiva que resuelve un problema $m \times n$ resolviendo recursivamente un $(m - 1) \times n$ o un $m \times (n - 1)$ subproblema. (*Sugerencia:* piense en MAX-HEAPIFY). Defina $T (p)$, donde $p = m + n$, para ser el tiempo de ejecución máximo de EXTRACT-MIN en cualquier cuadro de $m \times n$ Young. Da y resuelve una recurrencia para $T (p)$ que produce el límite de tiempo $O (m + n)$.
re. Muestre cómo insertar un nuevo elemento en un cuadro de Young $m \times n$ *no* completo en $O (m + n)$ hora.
mi. Sin usar ningún otro método de clasificación como subrutina, muestre cómo usar un $n \times n$ Young tableau para ordenar n^2 números en $O (n^3)$ tiempo.
F. Dar un algoritmo de tiempo $O (m + n)$ para determinar si un número dado se almacena en un dado $m \times n$ Cuadro joven.

Notas del capítulo

Página 127

El algoritmo heapsort fue inventado por Williams [316] , quien también describió cómo implementar una cola de prioridad con un montón. Se sugirió el procedimiento BUILD-MAX-HEAP por Floyd [90] .

Usamos min-montones para implementar colas de prioridad mínima en los capítulos 16 , 23 y 24 . También damos una implementación con límites de tiempo mejorados para ciertas operaciones en los Capítulos 19 y 20 .

Es posible realizar implementaciones más rápidas de colas de prioridad para datos enteros. Una estructura de datos inventado por van Emde Boas [301] apoya las operaciones MÍNIMO, MÁXIMO, INSERTAR, ELIMINAR, BÚSQUEDA, EXTRACCIÓN-MIN, EXTRACCIÓN-MAX, PREDECESOR y SUCESOR en el peor de los casos O (lg lg C), sujeto a la restricción de que el universo de claves es el conjunto {1, 2,. . . , C }. Si los datos son enteros de b bits y la memoria de la computadora de palabras direccionables de bits b , Fredman y Willard [99] mostraron cómo implementar MÍNIMO en O (1) tiempo e INSERT y EXTRACT-MIN en hora. Thorup [299] ha mejorado el ligado a $O ((\lg \lg n)^2)$ tiempo. Este límite usa una cantidad de espacio ilimitada en n , pero se puede implementar en un espacio lineal mediante el uso de hash aleatorio.

Un caso especial importante de colas de prioridad ocurre cuando la secuencia de EXTRACT-MIN Las operaciones son monótonas , es decir, los valores devueltos por las sucesivas operaciones EXTRACT-MIN. están aumentando monótonamente con el tiempo. Este caso surge en varias aplicaciones importantes, como el algoritmo de rutas más cortas de fuente única de Dijkstra, que se analiza en el capítulo 24 , y en simulación de eventos discretos. Para el algoritmo de Dijkstra es particularmente importante que el La operación DISMINUIR-CLAVE se implemente de manera eficiente. Para el caso monótono, si los datos son números enteros en el rango 1, 2,. . . , C , Ahuja, Melhorn, Orlin y Tarjan [8] describen cómo implementar EXTRACT-MIN e INSERT in O (lg C) tiempo amortizado (consulte el Capítulo 17 para obtener más

en análisis amortizado) y DISMINUCIÓN-CLAVE en $O(1)$ tiempo, utilizando una estructura de datos llamada montón de radix. El límite de $O(\lg C)$ se puede mejorar para usando montones de Fibonacci (ver Capítulo 20) junto con montones de radix. La unión se mejoró aún más a $O(\lg 1/3 + C)$ esperado tiempo de Cherkassky, Goldberg y Silverstein [58] , que combinan el agrupamiento multinivel estructura de Denardo y Fox [72] con el montón de Thorup mencionado anteriormente. Raman [256] mejoró aún más estos resultados para obtener una unión de $O(\min(\lg 1/4C, \lg 1/3 + n))$, para cualquier fijo > 0. Se pueden encontrar discusiones más detalladas de estos resultados en los artículos de Raman [256] y Thorup [299] .

Capítulo 7: Clasificación rápida

Quicksort es un algoritmo de clasificación cuyo tiempo de ejecución en el peor de los casos es $\Theta(n^2)$ en una matriz de entrada de n números. A pesar de este lento tiempo de ejecución en el peor de los casos, la ordenación rápida es a menudo la mejor práctica elección para la clasificación porque es notablemente eficiente en promedio: su tiempo de ejecución esperado es $\Theta(n \lg n)$, y los factores constantes ocultos en la notación $\Theta(n \lg n)$ son bastante pequeños. También tiene la ventaja de ordenar en su lugar (consulte la página 16) y funciona bien incluso en la memoria virtual Ambientes.

La sección 7.1 describe el algoritmo y una subrutina importante utilizada por quicksort para fraccionamiento. Debido a que el comportamiento de la ordenación rápida es complejo, comenzamos con un intuitivo discusión de su desempeño en la Sección 7.2 y posponer su análisis preciso hasta el final del capítulo. La sección 7.3 presenta una versión de clasificación rápida que utiliza muestreo aleatorio. Este algoritmo tiene un buen tiempo de ejecución de caso promedio y ninguna entrada en particular provoca su comportamiento en el peor de los casos.

Página 128

El algoritmo aleatorio se analiza en la Sección 7.4 , donde se muestra que se ejecuta en un tiempo $\Theta(n^2)$ en el peor de los casos y en $O(n \lg n)$ tiempo de media.

7.1 Descripción de quicksort

La ordenación rápida, como la ordenación combinada, se basa en el paradigma dividir y conquistar presentado en la Sección 2.3.1 . Aquí está el proceso de dividir y conquistar de tres pasos para clasificar un subarreglo típico $A[p \ldots r]$.

- **Dividir:** particione (reorganice) el arreglo $A[p \ldots r]$ en dos subarreglos (posiblemente vacíos) $A[p \ldots q-1]$ y $A[q+1 \ldots r]$ tales que cada elemento de $A[p \ldots q-1]$ es menor que o igual a $A[q]$, que es, a su vez, menor o igual a cada elemento de $A[q+1 \ldots r]$. Calcule el índice q como parte de este procedimiento de partición.
- **Conquista:** ordena los dos subarreglos $A[p \ldots q-1]$ y $A[q+1 \ldots r]$ mediante llamadas recursivas a ordenación rápida.
- **Combinar:** dado que los subarreglos se ordenan en su lugar, no es necesario trabajar para combinar ellos: ahora se ordena toda la matriz $A[p \ldots r]$.

El siguiente procedimiento implementa la ordenación rápida.

```
QUICKSORT (A, p, r)
1 si p < r
2 luego q ← PARTICIÓN (A, p, r)
3        QUICKSORT (A, p, q - 1)
4        QUICKSORT (A, q + 1, r)
```

Para ordenar una matriz completa A , la llamada inicial es QUICKSORT (A , 1, *longitud* [A]).

Particionando la matriz

La clave del algoritmo es el procedimiento de PARTICIÓN, que reordena el subarreglo $A[p \ldots r]$ en su lugar.

```
PARTICIÓN (A, p, r)
1 x ← A [r]
2 i ← p - 1
3 para j ← p hasta r - 1
4 hacer si A [j] ≤ x
5        entonces yo ← yo + 1
6                 intercambiar A [i] ↔ A [j]
7 intercambiar A [i + 1] ↔ A [r]
```

8 **devuelve** $i + 1$

La figura 7.1 muestra el funcionamiento de PARTICIÓN en una matriz de 8 elementos. PARTICIÓN siempre selecciona un elemento $x = A[r]$ como un elemento pivote alrededor del cual dividir el subarreglo $A[pr]$. A medida que se ejecuta el procedimiento, la matriz se divide en cuatro regiones (posiblemente vacías). Al principio de cada iteración del bucle **for** en las líneas 3-6, cada región satisface ciertas propiedades, que puede declarar como un ciclo invariante:

Figura 7.1: La operación de PARTICIÓN en una matriz de muestra. Elementos de matriz ligeramente sombreados están todos en la primera partición con valores no mayores que x . Los elementos muy sombreados están en el segunda partición con valores mayores que x . Los elementos sin sombrear aún no se han colocado en una de las dos primeras particiones, y el elemento blanco final es el pivote. *(a)* La matriz inicial y configuraciones variables. Ninguno de los elementos se ha colocado en ninguna de las dos primeras particiones. *(b)* El valor 2 se "intercambia consigo mismo" y se coloca en la partición de valores más pequeños. *(c)* - *(d)* El los valores 8 y 7 se agregan a la partición de valores mayores. *(e)* Los valores 1 y 8 se intercambian, y la partición más pequeña crece. *(f)* Los valores 3 y 8 se intercambian, y la partición más pequeña crece. *(g)* - *(h)* La partición más grande crece para incluir 5 y 6 y el ciclo termina. *(i)* En líneas 7-8, el elemento pivote se intercambia para que quede entre las dos particiones.

- Al comienzo de cada iteración del ciclo de las líneas 3-6, para cualquier índice de matriz k ,
  1. Si $p \le k \le i$, entonces $A[k] \le x$.
  2. Si $i + 1 \le k \le j - 1$, entonces $A[k] > x$.
  3. Si $k = r$, entonces $A[k] = x$.

La figura 7.2 resume esta estructura. Los índices entre j y $r - 1$ no están cubiertos por ningún de los tres casos, y los valores de estas entradas no tienen una relación particular con el pivote x .

Figura 7.2: Las cuatro regiones mantenidas por el procedimiento PARTICIÓN en un subarreglo $A[p\ r]$. Los valores en $A[pi]$ son todos menores o iguales ax , los valores en $A[i + 1\ j - 1]$ son todos mayor que x , y $A[r] = x$. Los valores en $A[jr - 1]$ pueden tomar cualquier valor.

Necesitamos demostrar que este invariante de ciclo es verdadero antes de la primera iteración, que cada iteración de el bucle mantiene el invariante, y que el invariante proporciona una propiedad útil para mostrar corrección cuando el bucle termina.

Página 130

- **Inicialización:** antes de la primera iteración del bucle, $i = p - 1$ y $j = p$. No existen valores entre p e i , y ningún valor entre $i + 1$ y $j - 1$, por lo que los dos primeros las condiciones del ciclo invariante se satisfacen trivialmente. La asignación en la línea 1 satisface la tercera condición.
- **Mantenimiento:** como muestra la Figura 7.3 , hay dos casos a considerar, dependiendo del resultado de la prueba en la línea 4. La figura 7.3 (a) muestra lo que sucede cuando $A [j] > x$; el la única acción en el ciclo es incrementar j . Después de que j se incrementa, la condición 2 se cumple para $A [j - 1]$ y todas las demás entradas permanecen sin cambios. La figura 7.3 (b) muestra lo que sucede cuando $A [j] \le x$; i se incrementa, $A [i]$ y $A [j]$ se intercambian, y luego j se incrementa. Debido al intercambio, ahora tenemos que $A [i] \le x$, y se cumple la condición 1. Similar, también tenemos que $A [j - 1] > x$, ya que el elemento que se cambió a $A [j - 1]$ es, por el invariante de bucle, mayor que x .

Figura 7.3: Los dos casos para una iteración del procedimiento PARTICIÓN. *(a)* Si $A [j] > x$, la única acción es incrementar j , lo que mantiene invariante el ciclo. *(b)* Si $A [j] \le x$, el índice i se incrementa, $A [i]$ y $A [j]$ se intercambian, y luego j se incrementa. De nuevo, el ciclo invariante se mantiene.

- **Terminación:** Al terminar, $j = r$. Por lo tanto, cada entrada de la matriz está en uno de los tres conjuntos descritos por el invariante, y hemos dividido los valores en la matriz en tres conjuntos: los menores o iguales ax , los mayores que x , y un conjunto singleton que contiene x .

Las últimas dos líneas de PARTICIÓN mueven el elemento pivote a su lugar en el medio de la matriz intercambiándola con el elemento más a la izquierda que sea mayor que x .

La salida de PARTICIÓN ahora satisface las especificaciones dadas para el paso de división.

El tiempo de ejecución de PARTICIÓN en el subarreglo $A [pr]$ es $\Theta (n)$, donde $n = r - p + 1$ (ver Ejercicio 7.1-3).

Ejercicios 7.1-1

Usando la Figura 7.1 como modelo, ilustre la operación de PARTICIÓN en el arreglo A = 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21.

Página 131

Ejercicios 7.1-2

¿Qué valor de q devuelve PARTICIÓN cuando todos los elementos de la matriz $A [pr]$ tienen la mismo valor? Modifique PARTICIÓN para que $q = (p + r) / 2$ cuando todos los elementos de la matriz $A [pr]$ tienen el mismo valor.

Ejercicios 7.1-3

Dé un breve argumento de que el tiempo de ejecución de PARTICIÓN en un subarreglo de tamaño n es $\Theta(n)$.

Ejercicios 7.1-4

¿Cómo modificaría QUICKSORT para clasificar en orden no creciente?

7.2 Rendimiento de clasificación rápida

El tiempo de ejecución de la clasificación rápida depende de si la partición está equilibrada o no equilibrada, y esto, a su vez, depende de qué elementos se utilicen para la partición. Si la partición es equilibrado, el algoritmo se ejecuta de forma asintótica tan rápido como la ordenación por combinación. Si la partición es desequilibrado, sin embargo, puede ejecutarse asintóticamente tan lentamente como la ordenación por inserción. En esta sección, Deberá investigar informalmente qué tan rápido se desempeña bajo los supuestos de equilibrio versus particionamiento desequilibrado.

Partición en el peor de los casos

El peor comportamiento de caso para la ordenación rápida ocurre cuando la rutina de partición produce una subproblema con n - 1 elementos y uno con 0 elementos. (Esta afirmación se prueba en la Sección 7.4.1 .) Supongamos que esta partición desequilibrada surge en cada llamada recursiva. los costos de partición $\Theta(n)$ tiempo. Dado que la llamada recursiva en una matriz de tamaño 0 simplemente regresa, $T(0) = \Theta(1)$, y la recurrencia del tiempo de ejecución es

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n).$$

Intuitivamente, si sumamos los costos incurridos en cada nivel de la recursividad, obtenemos una aritmética serie (ecuación (A.2)), que se evalúa como $\Theta(n^2)$. De hecho, es sencillo utilizar el método de sustitución para demostrar que la recurrencia $T(n) = T(n-1) + \Theta(n)$ tiene la solución $T(n) = \Theta(n^2)$. (Vea el ejercicio 7.2-1 .)

Página 132

Por lo tanto, si la partición está desequilibrada al máximo en cada nivel recursivo del algoritmo, el el tiempo de ejecución es $\Theta(n^2)$. Por lo tanto, el peor tiempo de ejecución de quicksort no es mejor que el del tipo de inserción. Además, el tiempo de ejecución de $\Theta(n^2)$ ocurre cuando la matriz de entrada ya está completamente ordenada: una situación común en la que la ordenación por inserción se ejecuta en tiempo $O(n)$.

Partición en el mejor de los casos

En la división más pareja posible, PARTICIÓN produce dos subproblemas, cada uno de tamaño no más que $n/2$, ya que uno es de tamaño $\lfloor n/2 \rfloor$ y uno de tamaño $\lceil n/2 \rceil - 1$. En este caso, quicksort se ejecuta mucho Más rápido. La recurrencia del tiempo de ejecución es entonces

$$T(n) \le 2T(n/2) + \Theta(n),$$

que por el caso 2 del teorema maestro (Teorema 4.1) tiene la solución $T(n) = O(n \lg n)$. Así, el equilibrio equitativo de los dos lados de la partición en cada nivel de la recursividad produce una algoritmo asintóticamente más rápido.

Partición equilibrada

El tiempo de ejecución promedio de caso de clasificación rápida está mucho más cerca del mejor caso que del peor caso, como mostrarán los análisis de la Sección 7.4 . La clave para entender por qué es entender cómo el equilibrio de la partición se refleja en la recurrencia que describe la ejecución

hora.

Suponga, por ejemplo, que el algoritmo de particionamiento siempre produce un proporcional de 9 a 1 split, que a primera vista parece bastante desequilibrado. Luego obtenemos la recurrencia

$$T(n) \le T(9n/10) + T(n/10) + cn$$

en el tiempo de ejecución de quicksort, donde hemos incluido explícitamente la constante c oculta en el término $\Theta(n)$. La figura 7.4 muestra el árbol de recursividad para esta recurrencia. Note que cada nivel del árbol tiene un costo cn, hasta que se alcanza una condición de límite a la profundidad $\log_{10} n = \Theta(\lg n)$, y entonces los niveles han costado como máximo cn. La recursividad termina en la profundidad $\log_{10/9} n = \Theta(\lg n)$. los el costo total de la clasificación rápida es por lo tanto $O(n \lg n)$. Por lo tanto, con una división proporcional de 9 a 1 en cada nivel de recursividad, que intuitivamente parece bastante desequilibrado, quicksort se ejecuta en $O(n \lg n)$ time- asintóticamente lo mismo que si la división fuera por la mitad. De hecho, incluso una división de 99 a 1 produce un tiempo de ejecución $O(n \lg n)$. La razón es que cualquier división de proporcionalidad *constante* produce un árbol de recursividad de profundidad $\Theta(\lg n)$, donde el costo en cada nivel es $O(n)$. El tiempo de ejecución es por lo tanto, $O(n \lg n)$ siempre que la división tenga proporcionalidad constante.

Página 133

Figura 7.4: Un árbol de recursividad para QUICKSORT en el que PARTITION siempre produce un 9 a 1 división, lo que produce un tiempo de ejecución de $O(n \lg n)$. Los nodos muestran tamaños de subproblemas, con por nivel costos a la derecha. Los costos por nivel incluyen la constante c implícita en el término $\Theta(n)$.

Intuición para el caso promedio

Para desarrollar una noción clara del caso promedio de clasificación rápida, debemos hacer una suposición sobre la frecuencia con la que esperamos encontrar las diversas entradas. El comportamiento de quicksort es determinado por el orden relativo de los valores en los elementos de la matriz dados como entrada, y no por los valores particulares de la matriz. Como en nuestro análisis probabilístico del problema de contratación en la sección 5.2 , asumiremos por ahora que todas las permutaciones de los números de entrada son igualmente probable.

Cuando ejecutamos quicksort en una matriz de entrada aleatoria, es poco probable que la partición siempre sucede de la misma manera en todos los niveles, como ha supuesto nuestro análisis informal. Esperamos que algunas de las divisiones estarán razonablemente bien equilibradas y otras bastante desequilibradas. Por ejemplo, el ejercicio 7.2-6 le pide que demuestre que aproximadamente el 80 por ciento de las veces PARTICIÓN produce una división que es más equilibrada que 9 a 1, y aproximadamente el 20 por ciento del tiempo produce una división que está menos equilibrada que 9 a 1.

En el caso medio, PARTICIÓN produce una combinación de divisiones "buenas" y "malas". En una recursividad árbol para una ejecución de caso promedio de PARTICIÓN, las divisiones buenas y malas se distribuyen aleatoriamente en todo el árbol. Supongamos, por el bien de la intuición, sin embargo, que el bien y malas divisiones alternan niveles en el árbol, y que las buenas divisiones son en el mejor de los casos y las malas las divisiones son divisiones en el peor de los casos. La figura 7.5 (a) muestra las divisiones en dos niveles consecutivos en el

árbol de recursividad. En la raíz del árbol, el costo de la partición es n y los subarreglos producidos tienen tamaños n - 1 y 0: el peor de los casos. En el siguiente nivel, el subarreglo de tamaño n - 1 es el mejor de los casos particionado en subarreglos de tamaño (n - 1) / 2 - 1 y (n - 1) / 2. Supongamos que el límite
El costo de la condición es 1 para el subarreglo de tamaño 0.

Figura 7.5: *(a)* Dos niveles de un árbol de recursividad para ordenación rápida. La partición en la raíz cuesta n y produce una división "incorrecta": dos subarreglos de tamaños 0 y n - 1. La partición del subarreglo de tamaño n - 1 cuesta n - 1 y produce una división "buena": subarreglos de tamaño (n - 1) / 2 - 1 y (n -

1) / 2. *(b)* Un solo nivel de un árbol de recursividad que está muy bien equilibrado. En ambas partes, el
El costo de partición para los subproblemas que se muestran con sombreado elíptico es $\Theta(n)$. Sin embargo, el
Los subproblemas que quedan por resolver en (a), mostrados con sombreado cuadrado, no son mayores que el
los subproblemas correspondientes que quedan por resolver en (b).

La combinación de la división incorrecta seguida de la división buena produce tres submatrices de tamaños 0, (n - 1) / 2 - 1 y (n - 1) / 2 a un costo de partición combinado de $\Theta(n) + \Theta(n - 1) = \Theta(n)$.
Ciertamente, esta situación no es peor que la de la Figura 7.5 (b) , es decir, un solo nivel de particionamiento que produce dos subarreglos de tamaño (n - 1) / 2, a un costo de $\Theta(n)$. Sin embargo, este último ¡La situación está equilibrada! Intuitivamente, el costo $\Theta(n - 1)$ de la mala división se puede absorber en el $\Theta(n)$ costo de la buena división, y la división resultante es buena. Por lo tanto, el tiempo de ejecución de quicksort, cuando los niveles alternan entre divisiones buenas y malas, es como el tiempo de ejecución de divisiones buenas solo: todavía $O(n \lg n)$, pero con una constante ligeramente mayor oculta por la notación O. Deberíamos Dar un análisis riguroso del caso promedio en la Sección 7.4.2 .

Ejercicios 7.2-1

Utilice el método de sustitución para demostrar que la recurrencia $T(n) = T(n - 1) + \Theta(n)$ tiene la solución $T(n) = \Theta(n^2)$, como se afirma al comienzo de la Sección 7.2 .

Ejercicios 7.2-2

¿Cuál es el tiempo de ejecución de QUICKSORT cuando todos los elementos de la matriz A tienen el mismo valor?

Ejercicios 7.2-3

Muestre que el tiempo de ejecución de QUICKSORT es $\Theta(n^2)$ cuando el arreglo A contiene distintos elementos y se ordena en orden decreciente.

Ejercicios 7.2-4

Los bancos a menudo registran transacciones en una cuenta en el orden de los tiempos de las transacciones, pero A muchas personas les gusta recibir sus extractos bancarios con cheques listados en orden por cheque. número. Las personas suelen emitir cheques en orden por número de cheque, y los comerciantes suelen cobrar ellos con rapidez razonable. El problema de convertir los pedidos de tiempo de transacción a El orden de los números de control es, por lo tanto, el problema de clasificar las entradas casi ordenadas. Argumenta que el El procedimiento INSERTION-SORT tendería a superar al procedimiento QUICKSORT en este problema.

Página 135

Ejercicios 7.2-5

Suponga que las divisiones en cada nivel de clasificación rápida están en la proporción 1 - α a α , donde $0 < \alpha \leq 1/2$ es una constante. Muestre que la profundidad mínima de una hoja en el árbol de recursividad es aproximadamente - lg n / lg α y la profundidad máxima es aproximadamente -lg n / lg (1 - α). (No preocuparse por el redondeo de enteros.)

Ejercicios 7.2-6: ⋆

Argumente que para cualquier constante $0 < \alpha \leq 1/2$, la probabilidad es aproximadamente 1 - 2 α *de* que en un matriz de entrada aleatoria, PARTICIÓN produce una división más equilibrada que 1- α a α .

7.3 Una versión aleatoria de quicksort

Al explorar el comportamiento de caso promedio de ordenación rápida, hemos asumido que todos las permutaciones de los números de entrada son igualmente probables. En una situación de ingeniería, sin embargo, no siempre se puede esperar que se mantenga. (Vea el ejercicio 7.2-4 .) Como vimos en la sección 5.3 , podemos a veces agregan aleatorización a un algoritmo para obtener un buen promedio de casos rendimiento sobre todas las entradas. Mucha gente considera la versión aleatoria resultante de quicksort como el algoritmo de clasificación de elección para entradas lo suficientemente grandes.

En la Sección 5.3 , aleatorizamos nuestro algoritmo permutando explícitamente la entrada. Podríamos hacer por lo que también para clasificación rápida, pero una técnica de aleatorización diferente, llamada muestreo aleatorio , produce un análisis más sencillo. En lugar de usar siempre $A[r]$ como pivote, usaremos un elemento elegido al azar elemento del subarreglo $A[p..r]$. Lo hacemos intercambiando el elemento $A[r]$ con un elemento elegido al azar de $A[p..r]$. Esta modificación, en la que muestreamos aleatoriamente el rango $p, ..., r$, asegura que el elemento pivote $x = A[r]$ sea igualmente probable que sea cualquiera de $r - p + 1$ elementos en el subarreglo. Debido a que el elemento pivote se elige al azar, esperamos que la división de la matriz de entrada que esté razonablemente bien equilibrada en promedio.

Los cambios en PARTICIÓN y QUICKSORT son pequeños. En el nuevo procedimiento de partición, simplemente implemente el intercambio antes de particionar:

```
PARTICIÓN ALEATORIZADA ( A , p , r )
1 i ← ALEATORIO ( p , r )
2 intercambiar A [ r ] ↔ A [ i ]
3 devolver PARTICIÓN ( A , p , r )
```

La nueva ordenación rápida llama a PARTICIÓN ALEATORIA en lugar de PARTICIÓN:

```
QUICKSORT ALEATORIZADO ( A , p , r )
1 si p < r
2 luego q ← PARTICIÓN ALEATORIZADA ( A , p , r )
3                RANDOMIZED-QUICKSORT ( A , p , q - 1)
4                QUICKSORT ALEATORIZADO ( A , q + 1, r )
```

Página 136

Analizamos este algoritmo en la siguiente sección .

Ejercicios 7.3-1

¿Por qué analizamos el rendimiento de caso promedio de un algoritmo aleatorio y no su

rendimiento en el peor de los casos?

Ejercicios 7.3-2

Durante la ejecución del procedimiento RANDOMIZED-QUICKSORT, ¿cuántas llamadas se hecho al generador de números aleatorios RANDOM en el peor de los casos? Que tal en el mejor ¿caso? Da tu respuesta en términos de notación Θ.

7.4 Análisis de clasificación rápida

La sección 7.2 proporcionó algo de intuición sobre el peor comportamiento de ordenación rápida y por qué espere que se ejecute rápidamente. En esta sección, analizamos el comportamiento de quicksort de forma más rigurosa. Comenzamos con un análisis del peor de los casos, que se aplica a QUICKSORT o RANDOMIZED-QUICKSORT, y concluir con un análisis de caso promedio de RANDOMIZED-QUICKSORT.

7.4.1 Análisis del peor de los casos

Vimos en la Sección 7.2 que una división en el peor de los casos en cada nivel de recursividad en quicksort produce a $\Theta(n^2)$ tiempo de ejecución, que, intuitivamente, es el peor tiempo de ejecución del algoritmo. Nosotros ahora demuestre esta afirmación.

Usando el método de sustitución (ver Sección 4.1), podemos mostrar que el tiempo de ejecución de ordenación rápida es $O(n^2)$. Sea $T(n)$ el peor de los casos para el procedimiento QUICKSORT en un entrada de tamaño n . Tenemos la recurrencia

(7,1)

donde el parámetro q varía de 0 an - 1 porque el procedimiento PARTICIÓN produce dos subproblemas con tamaño total n - 1. Suponemos que $T(n) \le cn^2$ para alguna constante c . Sustituyendo esta suposición en recurrencia (7.1), obtenemos

La expresión $q^2 + (n - q - 1)^2$ alcanza un máximo en el rango del parámetro $0 \le q \le n - 1$ en cualquier punto final, como se puede ver ya que la segunda derivada de la expresión con respecto a q es positivo (vea el ejercicio 7.4-3). Esta observación nos da el límite máximo $_{\le q \le n-1}(q^2 + (n - q - 1)^2)$ $\le (n - 1)^2 = n^2 - 2n + 1$. Continuando con nuestro límite de $T(n)$, obtenemos

Página 137

$$T(norte) \le cn^2 - c(2\,norte - 1) + \Theta(norte)$$
$$\le cn^2,$$

ya que podemos elegir la constante c lo suficientemente grande como para que el término $c(2n - 1)$ domine el $\Theta(n)$ término. Por tanto, $T(n) = O(n^2)$. Vimos en la Sección 7.2 un caso específico en el que quicksort toma $\Theta(n^2)$ tiempo: cuando la partición está desequilibrada. Alternativamente, el ejercicio 7.4-1 le pide que muestre esa recurrencia (7.1) tiene una solución de $T(n) = \Theta(n^2)$. Por tanto, el tiempo de ejecución (en el peor de los casos) de ordenación rápida es $\Theta(n^2)$.

7.4.2 Tiempo de ejecución esperado

Ya hemos dado un argumento intuitivo de por qué el tiempo de ejecución promedio de casos de RANDOMIZED-QUICKSORT es $O(n \lg n)$: si, en cada nivel de recursividad, la división inducida por PARTICIÓN ALEATORIZADA coloca cualquier fracción constante de los elementos en un lado de la partición, entonces el árbol de recursividad tiene profundidad $\Theta(\lg n)$, y se realiza trabajo $O(n)$ en cada nivel. Incluso si agregamos nuevos niveles con la división más desequilibrada posible entre estos niveles, el el tiempo total sigue siendo $O(n \lg n)$. Podemos analizar el tiempo de ejecución esperado de RANDOMIZED-QUICKSORT precisamente al comprender primero cómo opera el procedimiento de partición y luego, usando este conocimiento para derivar un límite $O(n \lg n)$ en el tiempo de ejecución esperado. Esta límite superior del tiempo de ejecución esperado, combinado con el límite del mejor caso ($n \lg n$)

en la sección 7.2 , produce un tiempo de ejecución esperado $\Theta(n \lg n)$.

Tiempo de ejecución y comparaciones

El tiempo de ejecución de QUICKSORT está dominado por el tiempo pasado en la PARTICIÓN procedimiento. Cada vez que se llama al procedimiento PARTICIÓN, se selecciona un elemento pivote y este elemento nunca se incluye en futuras llamadas recursivas a QUICK-SORT y DIVIDIR. Por lo tanto, puede haber como máximo n llamadas a PARTITION durante toda la ejecución de el algoritmo de clasificación rápida. Una llamada a PARTICIÓN toma $O(1)$ tiempo más una cantidad de tiempo que es proporcional al número de iteraciones del ciclo **for** en las líneas 3-6. Cada iteración de este **for** loop realiza una comparación en línea 4, comparando el elemento pivote con otro elemento de la matriz A . Por lo tanto, si podemos contar el número total de veces que se ejecuta la línea 4, puede limitar el tiempo total empleado en el bucle **for** durante toda la ejecución de QUICKSORT.

Lema 7.1

Sea X el número de comparaciones realizadas en la línea 4 de PARTICIÓN en todo el ejecución de QUICKSORT en una matriz de n elementos. Entonces el tiempo de ejecución de QUICKSORT es $O(n + X)$.

Prueba Según la discusión anterior, hay n llamadas a PARTICIÓN, cada una de las cuales hace un cantidad constante de trabajo y luego ejecuta el ciclo **for** varias veces. Cada iteración del bucle **for** ejecuta la línea 4.

Página 138

Nuestro objetivo, por lo tanto, es calcular X , el número total de comparaciones realizadas en todas las llamadas a DIVIDIR. No intentaremos analizar cuántas comparaciones se hacen en *cada* llamada a DIVIDIR. Más bien, derivaremos un límite general del número total de comparaciones. A Para hacerlo, debemos entender cuándo el algoritmo compara dos elementos de la matriz y cuándo no es asi. Para facilitar el análisis, cambiamos el nombre de los elementos de la matriz A como $z_1, z_2, \ldots, z_n$, con z_i siendo el i- ésimo elemento más pequeño. También definimos el conjunto $Z_{ij} = \{ z_i, z_{i+1}, \ldots, z_j \}$ como el conjunto de elementos entre z_i y z_j , inclusive.

¿Cuándo compara el algoritmo z_i y z_j ? Para responder a esta pregunta, primero observamos que cada par de elementos se compara como máximo una vez. ¿Por qué? Los elementos se comparan solo con elemento pivote y, después de que finaliza una llamada particular de PARTICIÓN, el elemento pivote utilizado en esa llamada nunca más se compara con otros elementos.

Nuestro análisis utiliza indicadores de variables aleatorias (consulte la Sección 5.2). Definimos

$X_{ij} = \mathbf{I}\{ z_i \text{ se compara con } z_j \}$,

donde estamos considerando si la comparación tiene lugar en algún momento durante el ejecución del algoritmo, no solo durante una iteración o una llamada de PARTITION. Ya que cada par se compara como máximo una vez, podemos caracterizar fácilmente el número total de comparaciones realizadas por el algoritmo:

Tomando las expectativas de ambos lados, y luego usando la linealidad de la expectativa y el Lema 5.1 , obtener

(7,2)

Queda por calcular Pr { z_i se compara con z_j }.

Es útil pensar en cuándo *no se* comparan dos elementos . Considere una entrada para una variedad rápida de los números del 1 al 10 (en cualquier orden), y suponga que el primer elemento pivote es 7. Entonces el La primera llamada a PARTICIÓN separa los números en dos conjuntos: {1, 2, 3, 4, 5, 6} y {8, 9, 10}. Al hacerlo, el elemento pivote 7 se compara con todos los demás elementos, pero ningún número de la el primer conjunto (por ejemplo, 2) es o se comparará con cualquier número del segundo conjunto (por ejemplo, 9).

En general, una vez que se elige un pivote x con $z_i < x < z_j$, sabemos que z_i y z_j no se pueden comparar en cualquier momento posterior. Si, por otro lado, z_i se elige como pivote antes que cualquier otro elemento en Z_{ij} , entonces z_i se comparará con cada elemento de Z_{ij} , excepto por sí mismo. De manera similar, si z_j se elige como pivote antes de cualquier otro elemento en Z_{ij} , entonces z_j se comparará con cada elemento en Z_{ij} , excepto para sí mismo. En nuestro ejemplo, los valores 7 y 9 se comparan porque 7 es el primer elemento de $Z_{7,9}$ a

Página 139

ser elegido como pivote. Por el contrario, 2 y 9 nunca se compararán porque el primer pivote elemento elegido de $Z_{2,9}$ es 7. Por lo tanto, z_i y z_j se comparan si y sólo si el primer elemento a ser elegido como pivote de Z_{ij} es z_i o z_j .

Ahora calculamos la probabilidad de que ocurra este evento. Antes del punto en el que un elemento de Z_{ij} se ha elegido como pivote, todo el conjunto Z_{ij} está junto en la misma partición. Por lo tanto, cualquier elemento de Z_{ij} es igualmente probable que sea el primero elegido como pivote. Porque el conjunto Z_{ij} tiene $j - i + 1$ elementos, la probabilidad de que cualquier elemento dado sea el primero elegido como pivote es $1 / (j - i + 1)$. Por lo tanto, tenemos

(7,3)

La segunda línea sigue porque los dos eventos son mutuamente excluyentes. Combinando ecuaciones (7.2) y (7.3), obtenemos que

Podemos evaluar esta suma usando un cambio de variables ($k = j - i$) y el límite en el serie armónica en la ecuación (A.7):

(7,4)

Por lo tanto, llegamos a la conclusión de que, utilizando la PARTICIÓN ALEATORIA, el tiempo de ejecución esperado de ordenación rápida es $O (n \lg n)$.

Ejercicios 7.4-1

Demuestra eso en la recurrencia

Página 140

Ejercicios 7.4-2

Muestre que el mejor tiempo de ejecución de quicksort es $\Omega (n \lg n)$.

Ejercicios 7.4-3

Muestre que $q^2 + (n - q - 1)^2$ alcanza un máximo sobre $q = 0, 1, ..., n - 1$ cuando $q = 0$ o $q = n - 1$.

Ejercicios 7.4-4

Muestre que el tiempo de ejecución esperado de RANDOMIZED-QUICKSORT es $\Omega (n \lg n)$.

Ejercicios 7.4-5

El tiempo de ejecución de la clasificación rápida se puede mejorar en la práctica aprovechando la
tiempo de ejecución de la ordenación por inserción cuando su entrada está "casi" ordenada. Cuando se llama a ordenación rápida en un
subarreglo con menos de k elementos, deje que simplemente regrese sin ordenar el subarreglo. Después
la llamada de nivel superior a quicksort devuelve, ejecute la ordenación por inserción en toda la matriz para finalizar la ordenación
proceso. Argumente que este algoritmo de clasificación se ejecuta en el tiempo esperado $O (nk + n \lg (n / k))$. Cómo
¿Se debe *elegir* k , tanto en la teoría como en la práctica?

Ejercicios 7.4-6: ⋆

Considere la posibilidad de modificar el procedimiento de PARTICIÓN eligiendo al azar tres elementos de
matriz A y particionando sobre su mediana (el valor medio de los tres elementos).
Calcule la probabilidad de obtener, en el peor de los casos, una división de α a $(1 - \alpha)$, en función de α en la
rango $0 < \alpha < 1$.

Problemas 7-1: corrección de la partición Hoare

Página 141

La versión de PARTICIÓN dada en este capítulo no es el algoritmo de partición original.
Aquí está el algoritmo de partición original, que se debe a T. Hoare:

PARTICIÓN DE HOARE (A , p , r)

```
1 x ← A [ p ]
2 i ← p - 1
3 j ← r + 1
4 mientras es VERDADERO
5 repita j ← j - 1
6                 hasta que A [ j ] ≤ x
7             repetir i ← i + 1
8                 hasta que A [ i ] ≥ x
9             si yo < j
10                luego intercambia A [ i ] ↔ A [ j ]
11                si no regresa j
```

a. Demuestre el funcionamiento de HOARE-PARTITION en la matriz A = 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21, mostrando los valores de la matriz y los valores auxiliares después cada iteración del bucle **for** en las líneas 4-11.

Las siguientes tres preguntas le piden que dé un argumento cuidadoso de que el procedimiento HOARE-La PARTICIÓN es correcta. Demuestre lo siguiente:

si. Los índices i y j son tales que nunca accedemos a un elemento de A fuera del subarreglo A [pr].
C. Cuando termina HOARE-PARTITION, devuelve un valor j tal que $p \le j < r$.
re. Cada elemento de A [pj] es menor o igual que cada elemento de A [j +1 r] cuando HOARE-PARTITION termina.

El procedimiento de PARTICIÓN en la Sección 7.1 separa el valor pivote (originalmente en A [r]) de las dos particiones que forma. El procedimiento HOARE-PARTITION, por otro lado, siempre coloca el valor pivote (originalmente en A [p]) en una de las dos particiones A [pj] y A [j + 1 r]. Dado que $p \le j < r$, esta división no es siempre trivial.

mi. Vuelva a escribir el procedimiento QUICKSORT para usar HOARE-PARTITION.

Problemas 7-2: análisis de ordenación rápida alternativo

Un análisis alternativo del tiempo de ejecución de la clasificación rápida aleatoria se centra en el tiempo de ejecución de cada llamada recursiva individual a QUICKSORT, en lugar del número de comparaciones realizadas.

a. Argumenta que, dada una matriz de tamaño n , la probabilidad de que cualquier elemento en particular sea elegido como pivote es 1 / n . Use esto para definir las variables aleatorias del indicador X_i = I { i th el elemento más pequeño se elige como pivote}. ¿Qué es E [X_i]?
si. Sea T (n) una variable aleatoria que denota el tiempo de ejecución de quicksort en una matriz de tamaño n . Argumenta eso

Página 142

(7,5)

C. Demuestre que la ecuación (7.5) se simplifica a

(7,6)

re. Muestra esa

(7,7)

mi. (*Sugerencia:* divida la suma en dos partes, una para k = 1, 2, ..., ⌈ n / 2⌉ - 1 y otra para k = ⌈ n / 2⌉, ..., n - 1.)
F. Usando el límite de la ecuación (7.7), demuestre que la recurrencia en la ecuación (7.6) tiene

la solución E [$T(n)$] = Θ ($n \lg n$). (*Sugerencia:* demuestre, por sustitución, que E [$T(n)$] ≤ $an \log n - bn$ para algunas constantes positivas *una* y *b*).

Problemas 7-3: Tipo chiflado

Los profesores Howard, Fine y Howard han propuesto la siguiente clasificación "elegante" algoritmo:

```
CLASIFICACIÓN DE STOOGE ( A , i , j )
1 si A [ i ]> A [ j ]
2 luego intercambie A [ i ] ↔ A [ j ]
3 si i + 1 ≥ j
4 luego regresa
5 k ← ⌊ ( j - i + 1) / 3⌋          ▸ Redondea hacia abajo.
6 CLASIFICACIÓN DE STOOGE ( A , i , j - k )      ▸ Primeros dos tercios.
7 CLASIFICACIÓN DE STOOGE ( A , i + k , j )      ▸ Últimos dos tercios.
8 CLASIFICACIÓN DE STOOGE ( A , i , j - k )      ▸ Los primeros dos tercios de nuevo.
```

a. Argumente que, si n = *longitud* [A], entonces STOOGE-SORT (A , 1, *longitud* [A]) ordena correctamente matriz de entrada A [1 n].

si. Dar una recurrencia para el peor tiempo de ejecución de STOOGE-SORT y un ajustado asintótica (notación Θ) limitada al peor tiempo de ejecución.

C. Compare el peor tiempo de ejecución de STOOGE-SORT con el de la ordenación por inserción, fusionar ordenación, ordenación en pila y ordenación rápida. ¿Los profesores merecen la titularidad?

Problemas 7-4: profundidad de pila para ordenación rápida

Página 143

El algoritmo QUICKSORT de la Sección 7.1 contiene dos llamadas recursivas a sí mismo. Después de la llamada a PARTICIÓN, el subarreglo izquierdo se ordena de forma recursiva y luego el subarreglo derecho es ordenados de forma recursiva. La segunda llamada recursiva en QUICKSORT no es realmente necesaria; puede evitarse mediante el uso de una estructura de control iterativa. Esta técnica, llamada **recursividad de la cola** , es proporcionado automáticamente por buenos compiladores. Considere la siguiente versión de quicksort, que simula la recursividad de la cola.

```
QUICKSORT '( A , p , r )
1 mientras p < r
2       hacer ▸ Particionar y ordenar el subarreglo izquierdo.
3             q ← PARTICIÓN ( A , p , r )
4             QUICKSORT '( A , p , q - 1)
5             p ← q + 1
```

a. Argumentan que QUICKSORT '(A , 1, longitud de [A]) ordena correctamente la matriz A .

Los compiladores suelen ejecutar procedimientos recursivos utilizando una **pila** que contiene información pertinente información, incluidos los valores de los parámetros, para cada llamada recursiva. La información para el la llamada más reciente está en la parte superior de la pila, y la información para la llamada inicial está en el fondo. Cuando se invoca un procedimiento, su información se **inserta** en la pila; cuando termina, su información se **abre** . Dado que asumimos que los parámetros de la matriz están representados por punteros, la información para cada llamada a procedimiento en la pila requiere O (1) espacio de pila. La **profundidad de la pila** es la cantidad máxima de espacio de pila utilizada en cualquier momento durante una cálculo.

si. Describir un escenario en el que la profundidad de la pila de QUICKSORT' es Θ (n) en un n -elemento matriz de entrada.

C. Modifique el código de QUICKSORT 'para que la profundidad de pila en el peor de los casos sea Θ (lg n). Mantenga el tiempo de ejecución esperado O (n lg n) del algoritmo.

Problemas 7-5: Partición mediana de 3

Una forma de mejorar el procedimiento RANDOMIZED-QUICKSORT es dividir alrededor de un pivote que se elige con más cuidado que eligiendo un elemento aleatorio del subarreglo. Uno El enfoque común es el método de la **mediana de 3** : elija el pivote como la mediana elemento) de un conjunto de 3 elementos seleccionados aleatoriamente del subarreglo. (Vea el ejercicio 7.4-6 .) este problema, supongamos que los elementos de la matriz de entrada $A[1 \ldots n]$ son distintos y que $n \geq 3$. Denotamos la matriz de salida ordenada por $A'[1 \ldots n]$. Usando el método de la mediana de 3 para elegir el elemento pivote x , define $p_i = \Pr\{x = A'[i]\}$.

a. Dé una fórmula exacta para p_i en función de n y i para $i = 2, 3, \ldots, n - 1$. (Tenga en cuenta que $p_1 = p_n = 0$.)

si. ¿En qué cantidad hemos aumentado la probabilidad de elegir el pivote cuando $x = A'[\lfloor (n + 1)/2 \rfloor]$, la mediana de $A[1 \ldots n]$, en comparación con la implementación ordinaria? Asumir que $n \to \infty$, y dé la razón límite de estas probabilidades.

Página 144

C. Si definimos una división "buena" en el sentido de elegir el pivote como $x = A'[i]$, donde $n/ \leq i \leq 2n/3$, ¿En qué cantidad hemos aumentado la probabilidad de obtener una buena división en comparación con la implementación ordinaria? (*Sugerencia: calcule* la suma mediante una integral).

re. Argumente que en el tiempo de ejecución $\Omega(n \lg n)$ de clasificación rápida, el método de la mediana de 3 afecta solo el factor constante.

Problemas 7-6: Clasificación difusa de intervalos

Considere un problema de clasificación en el que los números no se conocen con exactitud. En cambio, para cada número, conocemos un intervalo en la línea real a la que pertenece. Es decir, se nos da n intervalos cerrados de la forma $[a_i, b_i]$, donde $a_i \leq b_i$. El objetivo es **ordenar** estos intervalos de forma **difusa** , es decir, producir una permutación $i_1, i_2, \ldots, i_n$ de los intervalos tales que existen , satisfactorio $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Diseñe un algoritmo para ordenar n intervalos difusos . Su algoritmo debe tener el estructura general de un algoritmo que quicksorts los puntos extremos izquierdo (el un_i's), pero debe aprovechar los intervalos superpuestos para mejorar el tiempo de ejecución. (Como el Los intervalos se superponen cada vez más, el problema de ordenar los intervalos de forma difusa se vuelve más fácil y más fácil. Su algoritmo debería aprovechar tal superposición, en la medida que existe.)

si. Argumenta que tu algoritmo se ejecuta en el tiempo esperado $\Theta(n \lg n)$ en general, pero se ejecuta en tiempo esperado $\Theta(n)$ cuando todos los intervalos se superponen (es decir, cuando existe un valor x tal que $x \in [a_i, b_i]$ para todo i). Su algoritmo no debería verificar este caso explícitamente; más bien, su rendimiento debería mejorar naturalmente a medida que la cantidad de superposición aumenta.

Notas del capítulo

El procedimiento de clasificación rápida fue inventado por Hoare [147] ; La versión de Hoare aparece en el problema 7-1 . El procedimiento de PARTICIÓN dado en la Sección 7.1 se debe a N. Lomuto. El análisis en La sección 7.4 se debe a Avrim Blum. Sedgewick [268] y Bentley [40] proporcionan una buena referencia sobre los detalles de la implementación y su importancia.

McIlroy [216] mostró cómo diseñar un "adversario asesino" que produzca una matriz en la que prácticamente cualquier implementación de clasificación rápida lleva $\Theta(n^2)$ tiempo. Si la implementación es aleatorio, el adversario produce la matriz después de ver las opciones aleatorias de la clasificación rápida algoritmo.

Capítulo 8: Ordenación en tiempo lineal

Visión general

Ahora hemos introducido varios algoritmos que pueden ordenar n números en $O(n \lg n)$ tiempo. Unir

sort y heapsort logran este límite superior en el peor de los casos; quicksort lo logra en promedio.

Página 145

Además, para cada uno de estos algoritmos, podemos producir una secuencia de n números de entrada que hace que el algoritmo se ejecute en $\Theta(n \lg n)$ tiempo.

Estos algoritmos comparten una propiedad interesante: *el orden de clasificación que determinan se basa solo en comparaciones entre los elementos de entrada* . Llamamos a estos algoritmos de clasificación comparación géneros . Todos los algoritmos de clasificación introducidos hasta ahora son tipos de comparación.

En la sección 8.1 , demostraremos que cualquier tipo de comparación debe hacer comparaciones $\Theta(n \lg n)$ en el peor de los casos para ordenar n elementos. Por lo tanto, la ordenación por combinación y la ordenación en pila son asintóticamente óptimas, y no existe ningún tipo de comparación que sea más rápido en más de un factor constante.

Las secciones 8.2 , 8.3 y 8.4 examinan tres algoritmos de ordenación: ordenación de recuento, ordenación de base y clasificación de cubos: que se ejecutan en tiempo lineal. No hace falta decir que estos algoritmos utilizan operaciones distintas de comparaciones para determinar el orden de clasificación. En consecuencia, el límite inferior $\Theta(n \lg n)$ no aplicarles.

8.1 Límites inferiores para ordenar

En una clasificación de comparación, usamos solo comparaciones entre elementos para obtener información de orden sobre una secuencia de entrada $a_1, a_2, \ldots, una$ Es decir, dados dos elementos a_i y a_j , realizamos una de las pruebas $a_i < un_j$, $un_i \le un_j$, $un_i = un_j$, $un_i \ge un_j$, o $un_i > una_j$ para determinar su orden relativo. Nosotros no puede inspeccionar los valores de los elementos ni obtener información de orden sobre ellos en ningún otro camino.

En esta sección, asumimos sin pérdida de generalidad que todos los elementos de entrada son distintos. Dada esta suposición, las comparaciones de la forma $a_i = a_j$ son inútiles, por lo que podemos suponer que no se hacen comparaciones de esta forma. También notamos que las comparaciones $a_i \le a_j$, $a_i \ge a_j$, $a_i > a_j$, y $a_i < a_j$ son todos equivalentes en el sentido de que producen información idéntica sobre el orden relativo de a_i y a_j . Por lo tanto, asumimos que todas las comparaciones tienen la forma $a_i \le a_j$.

El modelo de árbol de decisiones

Los tipos de comparación se pueden ver de forma abstracta en términos de árboles de decisión . Un árbol de decisiones es un árbol binario que representa las comparaciones entre elementos que realiza un algoritmo de clasificación particular que opera en una entrada de un tamaño dado. Control, movimiento de datos, y todos los demás aspectos del algoritmo se ignoran. La figura 8.1 muestra el árbol de decisiones correspondiente al algoritmo de ordenación por inserción de la Sección 2.1 que opera en una secuencia de entrada de tres elementos.

Figura 8.1: El árbol de decisión para la ordenación por inserción que opera en tres elementos. Un nodo interno anotado por $i:j$ indica una comparación entre una_i y una_j . Una hoja anotada por el permutación $\pi(1), \pi(2), \ldots, \pi(n)$ indica el orden $a_{\pi(1)} \le a_{\pi(2)} \le \cdots a_{\pi(n)}$. El sombreado ruta indica las decisiones tomadas al ordenar la secuencia de entrada $a_1 = 6$, $a_2 = 8$, $a_3 = 5$

Página 146

la permutación 3, 1, 2 en la hoja indica que el orden ordenado es $a_3 = 5$ $a_1 = 6$ $a_2 = 8$.
¡Hay 3! = 6 posibles permutaciones de los elementos de entrada, por lo que el árbol de decisión debe tener en

mínimo 6 hojas.

En un árbol de decisión, cada nodo interno está anotado por $i : j$ para algunos i y j en el rango $1 \le i , jn$, donde n es el número de elementos en la secuencia de entrada. Cada hoja está anotada por un permutación $\pi (1), \pi (2),. . ., \pi (n)$. (Consulte la Sección C.1 para conocer los antecedentes de las permutaciones). La ejecución del algoritmo de clasificación corresponde a trazar una ruta desde la raíz de la decisión. árbol a una hoja. En cada nodo interno, se hace una comparación $a_i a_j$. El subárbol izquierdo luego dicta comparaciones posteriores para $a_i a_j$, y el subárbol derecho dicta comparaciones posteriores para $a_i > una_j$. Cuando llegamos a una hoja, el algoritmo de ordenación ha establecido el ordenamiento $a_{\pi(1)} a_{\pi(2)} \cdots un_{\pi(n)}$. Porque cualquier algoritmo de ordenación correcto debe poder producir cada permutación de su entrada, una condición necesaria para que una clasificación de comparación sea correcta es que cada uno de los $n !$ permutaciones en n elementos deben aparecer como una de las hojas del árbol de decisión, y que cada una de estas hojas debe ser accesible desde la raíz por un camino correspondiente a un ejecución del tipo de comparación. (Nos referiremos a esas hojas como "alcanzables"). considerará solo árboles de decisión en los que cada permutación aparece como una hoja alcanzable.

Un límite inferior para el peor de los casos

La longitud del camino más largo desde la raíz de un árbol de decisión hasta cualquiera de sus hojas alcanzables. representa el número de comparaciones en el peor de los casos que el algoritmo de clasificación correspondiente realiza. En consecuencia, el número de comparaciones en el peor de los casos para un tipo de comparación dado algoritmo es igual a la altura de su árbol de decisión. Un límite inferior en las alturas de toda decisión árboles en los que cada permutación aparece como una hoja alcanzable es, por lo tanto, un límite inferior en el tiempo de ejecución de cualquier algoritmo de clasificación de comparación. El siguiente teorema establece tal límite inferior.

Teorema 8.1

Cualquier algoritmo de clasificación de comparación requiere comparaciones $\Omega (n \lg n)$ en el peor de los casos.

Prueba De la discusión anterior, es suficiente determinar la altura de un árbol de decisión en que cada permutación aparece como una hoja alcanzable. Considere un árbol de decisión de altura h con l hojas alcanzables correspondientes a una clasificación de comparación en n elementos. Porque cada uno de los $n !$ permutaciones de la entrada aparece como una hoja, tenemos $n ! \le l$. Dado que un árbol binario de altura h no tiene más de $2_{h de}$ hojas, tenemos

$n ! \le l 2_h$,

que, tomando logaritmos, implica

$h \le \lg (n !)$ (ya que la función lg es monótona creciente)
$= \Omega (n \lg n)$ (por la ecuación (3.18)).

Corolario 8.2

Página 147

Heapsort y merge sort son tipos de comparación óptimos asintóticamente.

Prueba Loslímites superiores $O (n \lg n)$ en los tiempos de ejecución para el ordenamiento en pila y el ordenamiento por combinación coinciden con el $\Omega (n \lg n)$ límite inferior del peor caso del teorema 8.1 .

Ejercicios 8.1-1

¿Cuál es la profundidad más pequeña posible de una hoja en un árbol de decisión para una clasificación de comparación?

Ejercicios 8.1-2

Obtenga límites asintóticamente ajustados en lg (n !) Sin usar la aproximación de Stirling. En lugar, evaluar la suma utilizando técnicas de la Sección A.2 .

Ejercicios 8.1-3

¡Demuestre que no hay ningún tipo de comparación cuyo tiempo de ejecución sea lineal durante al menos la mitad de n ! entradas de longitud n . ¿Qué pasa con una fracción de 1 / n de las entradas de longitud n ? Que tal un fracción $1/2^n$?

Ejercicios 8.1-4

Se le da una secuencia de n elementos para ordenar. La secuencia de entrada consta de n / k subsecuencias, cada una de las cuales contiene k elementos. Los elementos en una subsecuencia dada son todos más pequeño que los elementos en la subsecuencia siguiente y más grande que los elementos en el subsecuencia anterior. Por tanto, todo lo que se necesita para ordenar la secuencia completa de longitud n es ordenar los k elementos en cada una de las n / k subsecuencias. Muestre un límite inferior $\Omega (n \lg k)$ en el número de comparaciones necesarias para resolver esta variante del problema de clasificación. (*Pista:* no es riguroso para simplemente combinar los límites inferiores para las subsecuencias individuales).

8.2 Clasificación de conteo

El ordenamiento de conteo asume que cada uno de los n elementos de entrada es un número entero en el rango de 0 a k , por algún entero k . Cuando $k = O (n)$, la ordenación se ejecuta en $\Theta (n)$ tiempo.

La idea básica de contar ordenamiento es determinar, para cada elemento de entrada x , el número de elementos menores que x . Esta información se puede utilizar para colocar el elemento x directamente en su posición en la matriz de salida. Por ejemplo, si hay 17 elementos menos que x , entonces x pertenece en la salida

Página 148

posición 18. Este esquema debe modificarse ligeramente para manejar la situación en la que varios Los elementos tienen el mismo valor, ya que no queremos ponerlos todos en la misma posición.

En el código para contar ordenación, asumimos que la entrada es una matriz $A [1 \ldots n]$, y por lo tanto $longitud [A] = n$. Necesitamos otras dos matrices: la matriz $B [1 \ldots n]$ contiene la salida ordenada y la la matriz $C [0 \ldots k]$ proporciona almacenamiento de trabajo temporal.

```
CONTEO-CLASIFICACIÓN ( A , B , k )
 1 para i ← 0 a k
 2 hacer C [ i ] ← 0
 3 para j ← 1 a la longitud [ A ]
 4 hacer C [ A [ j ]] ← C [ A [ j ]] + 1
 5 ▸ C [ i ] ahora contiene el número de elementos igual a i .
 6 para i ← 1 a k
 7 hacer C [ i ] ← C [ i ] + C [ i - 1]
 8 ▸ C [ i ] ahora contiene el número de elementos menores o iguales que i .
 9 para j ← longitud [ A ] hacia abajo a 1
10 ¿ B [ C [ A [ j ]]] ← A [ j ]
11          C [ A [ j ]] ← C [ A [ j ]] - 1
```

La figura 8.2 ilustra la clasificación de conteo. Después de la inicialización en el bucle **for** de las líneas 1-2, inspeccione cada elemento de entrada en el bucle **for** de las líneas 3-4. Si el valor de un elemento de entrada es i , incremento $C [i]$. Por lo tanto, después de la línea 4, $C [i]$ tiene el número de elementos de entrada igual a i para cada entero $i = 0, 1, \ldots, k$. En las líneas 6-7, determinamos para cada $i = 0, 1, \ldots, k$, cuántas entradas elementos son menos que o igual a i , manteniendo una suma continua de la matriz C .

Figura 8.2: La operación de COUNTING-SORT en una matriz de entrada A [1 8], donde cada elemento de A es un número entero no negativo no mayor que $k = 5$. *(a)* La matriz A y el auxiliar matriz C después de la línea 4. *(b)* La matriz C después de la línea 7. *(c)* - *(e)* La matriz de salida B y el auxiliar matriz C después de una, dos y tres iteraciones del ciclo en las líneas 9-11, respectivamente. Solo el Se han completado los elementos ligeramente sombreados de la matriz B. *(f)* La matriz B *de* salida ordenada final .

Finalmente, en el ciclo **for** de las líneas 9-11, colocamos cada elemento A [j] en su posición ordenada correcta en la matriz de salida B . Si todos los n elementos son distintos, cuando ingresemos por primera vez en la línea 9, para cada A [j], el valor C [A [j]] es la posición final correcta de A [j] en la matriz de salida, ya que hay C [A [j]] elementos menores o iguales que A [j]. Dado que los elementos pueden no ser distintos, decrementa C [A [j]] cada vez que colocamos un valor A [j] en la matriz B. Decrementando C [A [j]] hace que el siguiente elemento de entrada con un valor igual a A [j], si existe, vaya a la posición inmediatamente antes de A [j] en la matriz de salida.

¿Cuánto tiempo requiere contar el orden? El bucle **for** de las líneas 1-2 lleva tiempo $\Theta(k)$, el **for** el ciclo de las líneas 3-4 lleva tiempo $\Theta(n)$, el ciclo **for** de las líneas 6-7 lleva tiempo time (k) y el ciclo **for** de las líneas 9-11 toman tiempo $\Theta(n)$. Por tanto, el tiempo total es $\Theta(k + n)$. En la práctica, solemos utilizar contando ordenar cuando tenemos $k = O(n)$, en cuyo caso el tiempo de ejecución es $\Theta(n)$.

Página 149

El ordenamiento de conteo supera el límite inferior de $\Omega(n \lg n)$ demostrado en la sección 8.1 porque no es un tipo de comparación. De hecho, no se producen comparaciones entre los elementos de entrada en ninguna parte del código. En su lugar, la ordenación por recuento utiliza los valores reales de los elementos para indexarlos en una matriz. El $\Theta(n \lg n)$ el límite inferior para la clasificación no se aplica cuando nos apartamos del modelo de clasificación por comparación.

Una propiedad importante del ordenamiento de conteo es que es estable : aparecen números con el mismo valor en la matriz de salida en el mismo orden que lo hacen en la matriz de entrada. Es decir, lazos entre dos los números se rompen por la regla de que el número que aparece primero en la matriz de entrada aparece primero en la matriz de salida. Normalmente, la propiedad de estabilidad es importante solo cuando el satélite los datos se transportan con el elemento que se ordena. Contar la estabilidad del género es importante para otra razón: la ordenación por conteo se usa a menudo como una subrutina en la ordenación por base. Como veremos en el En la siguiente sección , contar la estabilidad de la ordenación es crucial para la corrección de la ordenación de base.

Ejercicios 8.2-1

Utilizando la Figura 8.2 como modelo, ilustre la operación de COUNTING-SORT en el arreglo A = 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2.

Ejercicios 8.2-2

Demuestre que COUNTING-SORT es estable.

Ejercicios 8.2-3

Suponga que el encabezado del bucle **for** en la línea 9 del procedimiento COUNTING-SORT se reescribe como

9 **para** $j \leftarrow 1$ **a la** *longitud* [A]

Demuestre que el algoritmo aún funciona correctamente. ¿Es estable el algoritmo modificado?

Ejercicios 8.2-4

Describe un algoritmo que, dados n números enteros en el rango de 0 a k , preprocesa su entrada y luego responde cualquier consulta sobre cuántos de los n enteros caen en un rango [ab] en $O(1)$ tiempo. Su algoritmo debe usar un tiempo de preprocesamiento $\Theta(n + k)$.

8.3 Orden de radix

Radix sort es el algoritmo utilizado por las máquinas clasificadoras de tarjetas que ahora solo se encuentran en computadoras museos. Las cartas están organizadas en 80 columnas, y en cada columna se puede hacer un hueco.

Página 150

perforado en uno de los 12 lugares. El clasificador se puede "programar" mecánicamente para examinar un columna dada de cada carta en un mazo y distribuir la carta en uno de los 12 contenedores dependiendo de qué lugar ha sido perforado. Luego, un operador puede recoger las tarjetas bandeja por bandeja, de modo que las tarjetas con el primer lugar perforado están encima de las tarjetas con el segundo lugar perforado, y así sucesivamente.

Para los dígitos decimales, solo se utilizan 10 lugares en cada columna. (Los otros dos lugares se utilizan para codificación de caracteres no numéricos.) Un número de d dígitos ocuparía un campo de d columnas. Dado que el clasificador de tarjetas solo puede ver una columna a la vez, el problema de clasificar n tarjetas en un número de d dígitos requiere un algoritmo de clasificación.

Intuitivamente, uno podría querer ordenar los números por su dígito *más significativo* , ordenar cada uno de los bins resultantes de forma recursiva y, a continuación, combine los mazos en orden. Desafortunadamente, dado que las cartas en 9 de los 10 bins deben dejarse a un lado para clasificar cada uno de los bins, este procedimiento genera muchos pilas intermedias de tarjetas de las que se debe realizar un seguimiento. (Vea el ejercicio 8.3-5 .)

La ordenación por radix resuelve el problema de la ordenación de cartas de forma contraria a la intuición al ordenar por lo *menos* primer dígito *significativo* . Luego, las cartas se combinan en una sola baraja, con las cartas en el 0 bandeja que precede a las tarjetas en la bandeja 1 que precede a las tarjetas en la bandeja 2, y así sucesivamente. Entonces todo La baraja se ordena de nuevo en el segundo dígito menos significativo y se recombina de manera similar. El proceso continúa hasta que las tarjetas se hayan clasificado en todos los d dígitos. Sorprendentemente, en eso señale que las tarjetas están completamente ordenadas según el número d dígitos. Por lo tanto, solo d pasa a través del tablero se requieren para clasificar. La figura 8.3 muestra cómo funciona la ordenación por radix en una "plataforma" de siete dígitos de 3 dígitos. números.

Figura 8.3: La operación de ordenación por base en una lista de siete números de 3 dígitos. El más a la izquierda la columna es la entrada. Las columnas restantes muestran la lista después de sucesivas clasificaciones en posiciones de dígitos cada vez más importantes. El sombreado indica la posición de los dígitos ordenados en producir cada lista a partir de la anterior.

Es esencial que los ordenamientos de dígitos en este algoritmo sean estables. El tipo realizado por una tarjeta clasificador es estable, pero el operador debe tener cuidado de no cambiar el orden de las tarjetas como salen de un contenedor, aunque todas las tarjetas de un contenedor tienen el mismo dígito en la casilla elegida columna.

En una computadora típica, que es una máquina secuencial de acceso aleatorio, la ordenación por radix es a veces se utiliza para ordenar registros de información que están codificados por múltiples campos. Por ejemplo, podríamos desea ordenar las fechas por tres claves: año, mes y día. Podríamos ejecutar un algoritmo de clasificación con una función de comparación que, dadas dos fechas, compara años, y si hay un empate, compara meses, y si ocurre otro empate, compara días. Alternativamente, podríamos ordenar la información tres veces con un tipo estable: primero el día, luego el mes y finalmente el año.

Página 151

El código para la ordenación por radix es sencillo. El siguiente procedimiento asume que cada elemento en la matriz de n elementos, A tiene d dígitos, donde el dígito 1 es el dígito de orden más bajo y el dígito d es el dígito de orden más alto.

CLASIFICACIÓN DE RADIX (A , d)
1 **para** $i \leftarrow 1$ **a** d
2 **hacen** uso de una especie estable al tipo matriz A en dígitos i

Lema 8.3

Dados números de nd dígitos en los que cada dígito puede tomar hasta k valores posibles, RADIX-SORT ordena correctamente estos números en $\Theta (d (n + k))$ tiempo.

Prueba La corrección de la ordenación por radix sigue por inducción en la columna que se está ordenando (ver Ejercicio 8.3-3). El análisis del tiempo de ejecución depende del tipo estable utilizado como algoritmo de clasificación intermedio. Cuando cada dígito está en el rango de 0 a k -1 (para que pueda asumir k valores posibles), yk no es demasiado grande, contar ordenado es la elección obvia. Cada pase sobre n Los números de d- dígitos toman tiempo $\Theta (n + k)$. Hay d pasadas, por lo que el tiempo total para la ordenación por base es $\Theta (d (n + k))$.

Cuando d es constante y $k = O (n)$, la ordenación por base se ejecuta en tiempo lineal. De manera más general, tenemos algunos flexibilidad en cómo dividir cada tecla en dígitos.

Lema 8.4

Dados números de nb bits y cualquier entero positivo $r \leq b$, RADIX-SORT ordena correctamente estos números en $\Theta ((b / r) (n + 2 ^{r}))$ tiempo.

Prueba Para un valor $r \leq b$, consideramos que cada tecla tiene $d = \lceil b / r \rceil$ dígitos de r bits cada una. Cada dígito es un número entero en el rango de 0 a $2 ^{r}$ - 1, por lo que podemos usar la ordenación de conteo con $k = 2 ^{r}$ - 1. (Para Por ejemplo, podemos ver una palabra de 32 bits con 4 dígitos de 8 bits, de modo que $b = 32$, $r = 8$, $k = 2 ^{r}$ - 1 = 255, y $d = b / r = 4$.) Cada pasada de clasificación de conteo toma tiempo $\Theta (n + k) = \Theta (n + 2 ^{r})$ y hay d pasa, para un tiempo total de ejecución de $\Theta (d (n + 2 ^{r})) = \Theta ((b / r) (n + 2 ^{r}))$.

Para valores dados de n y b , deseamos elegir el valor de r , con $r \leq b$, que minimiza la expresión $(b / r) (n + 2 ^{r})$. Si $b < \lfloor \lg n \rfloor$, entonces para cualquier valor de rb , tenemos que $(n + 2 ^{r}) = \Theta (n)$. Por lo tanto, elegir $r = b$ produce un tiempo de ejecución de $(b / b) (n + 2 ^{b}) = \Theta (n)$, que es asintóticamente óptimo. Si $b \geq \lfloor \lg n \rfloor$, entonces elegir $r = \lfloor \lg n \rfloor$ da el mejor tiempo dentro de un factor constante, que podemos ver de la siguiente manera. Al elegir $r = \lfloor \lg n \rfloor$ se obtiene un tiempo de ejecución de $\Theta (bn / \lg n)$. Como nosotros aumenta r *por* encima de $\lfloor \lg n \rfloor$, el término $2 ^{r}$ en el numerador aumenta más rápido que el término r en el denominador, por lo que aumentar r *por* encima de $\lfloor \lg n \rfloor$ produce un tiempo de ejecución de $\Theta (bn / \lg n)$. Si en cambio

Página 152

debíamos disminuir r por debajo de $\lfloor \lg n \rfloor$, luego el término b / r aumenta y el término $n + 2 ^{r}$ permanece en $\Theta (n)$.

¿Es preferible la ordenación por radix a un algoritmo de ordenación basado en comparaciones, como la ordenación rápida? Si $b = O (\lg n)$, como suele ser el caso, y elegimos $r \approx \lg n$, entonces el tiempo de ejecución de radix sort es $\Theta (n)$, que parece ser mejor que el tiempo promedio de caso de Quicksort de $\Theta (n \lg n)$. El constante Sin embargo, los factores ocultos en la notación Θ difieren. Aunque la ordenación por radix puede hacer menos pasadas que la ordenación rápida sobre las n teclas, cada pasada de ordenación por base puede tardar mucho más. Cual El algoritmo de clasificación es preferible depende de las características de las implementaciones, de la máquina subyacente (por ejemplo, quicksort a menudo usa cachés de hardware de manera más efectiva que radix sort) y de los datos de entrada. Además, la versión de la ordenación de radix que utiliza la ordenación de conteo como El ordenamiento estable intermedio no ordena en su lugar, lo que muchas de las comparaciones de tiempo $\Theta (n \lg n)$ las clases hacen. Por lo tanto, cuando el almacenamiento de la memoria primaria es escaso, un algoritmo in situ como puede ser preferible la clasificación rápida.

Ejercicios 8.3-1

Utilizando la Figura 8.3 como modelo, ilustre el funcionamiento de RADIX-SORT en la siguiente lista de Palabras en inglés: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, GRANDE, TÉ, AHORA, ZORRO.

Ejercicios 8.3-2

¿Cuáles de los siguientes algoritmos de ordenación son estables: ordenación por inserción, ordenación por combinación, ordenación en pila y ¿ordenación rápida? Proporcione un esquema simple que establezca cualquier algoritmo de clasificación. Cuánto cuesta ¿tiempo y espacio adicional implica su esquema?

Ejercicios 8.3-3

Utilice la inducción para demostrar que la ordenación por radix funciona. ¿Dónde necesita su prueba la suposición de que el tipo intermedio es estable?

Ejercicios 8.3-4

Muestre cómo ordenar n números enteros en el rango 0 an ^{2} - 1 en el tiempo O (n).

Ejercicios 8.3-5: ⋆

Página 153

En el primer algoritmo de clasificación de tarjetas de esta sección, exactamente cuántas pasadas de clasificación se necesitan ordenar números decimales de d dígitos en el peor de los casos? ¿Cuántas pilas de cartas tendría un operador? necesita realizar un seguimiento en el peor de los casos?

8.4 Clasificación de cubos

La clasificación de cubos se ejecuta en tiempo lineal cuando la entrada se extrae de una distribución uniforme. Me gusta contando la clasificación, la clasificación por cubeta es rápida porque asume algo sobre la entrada. Mientras contar ordenación asume que la entrada consiste en enteros en un rango pequeño, la ordenación de cubeta asume que la entrada es generada por un proceso aleatorio que distribuye elementos uniformemente sobre el intervalo [0, 1). (Consulte la Sección C.2 para obtener una definición de distribución uniforme).

La idea de la clasificación de cubos es dividir el intervalo [0, 1) en n subintervalos de igual tamaño, o cubos y luego distribuya los n números de entrada en los cubos. Dado que las entradas son distribuidos uniformemente en [0, 1), no esperamos que caigan muchos números en cada segmento. A producir la salida, simplemente clasificamos los números en cada cubo y luego revisamos el cubos en orden, enumerando los elementos en cada uno.

Nuestro código para la ordenación de cubos asume que la entrada es una matriz A *de* n elementos y que cada elemento A [i] en la matriz satisface $0 \le A$ [i] < 1. El código requiere una matriz auxiliar B [0 n - 1] de listas enlazadas (cubos) y asume que existe un mecanismo para mantener dichas listas. (La Sección 10.2 describe cómo implementar operaciones básicas en listas enlazadas).

CLASIFICACIÓN CUBO (A)

```
1 n ← longitud [ A ]
2 para i ← 1 a n
3 hacer inserto A [ i ] en la lista B [⌊ n A [ i ] ⌋]
4 para i ← 0 a n - 1
5 do ordenar la lista de B [ i ] con ordenación por inserción
6 concatenar las listas B [0], B [1],. . ., B [ n - 1] juntos en orden
```

La figura 8.4 muestra la operación de clasificación de cubos en una matriz de entrada de 10 números.

Figura 8.4: El funcionamiento de BUCKET-SORT. *(a)* La matriz de entrada A [1 10]. *(b)* La matriz B [0 9] de listas ordenadas (cubos) después de la línea 5 del algoritmo. El cubo i contiene valores en el

Página 154

intervalo semiabierto [i / 10, (i + 1) / 10). La salida ordenada consiste en una concatenación en orden de las listas B [0], B [1],. . ., B [9].

Para ver que este algoritmo funciona, considere dos elementos A [i] y A [j]. Asumir sin pérdida de generalidad que A [i] ≤ A [j]. Dado que ⌊ n A [i] ⌋ ⌊ n A [j] ⌋, el elemento A [i] se coloca en el mismo cubo como A [j] o en un cubo con un índice más bajo. Si A [i] y A [j] se colocan en el mismo bucket, luego el bucle **for** de las líneas 4-5 las coloca en el orden correcto. Si A [i] y A [j] son colocados en diferentes cubos, luego la línea 6 los coloca en el orden correcto. Por lo tanto, cubo sort funciona correctamente.

Para analizar el tiempo de ejecución, observe que todas las líneas excepto la línea 5 toman O (n) tiempo en el peor caso. Queda por equilibrar el tiempo total que tardan las n llamadas en ordenar por inserción en la línea 5.

Para analizar el costo de las llamadas a la ordenación por inserción, sea n_i la variable aleatoria que denota el número de elementos colocados en el cubo B [i]. Dado que la ordenación por inserción se ejecuta en tiempo cuadrático (consulte Sección 2.2), el tiempo de ejecución de la clasificación de cubos es

Tomando las expectativas de ambos lados y usando la linealidad de la expectativa, tenemos

(8,1)

Afirmamos que

(8,2)

para i = 0, 1,. . ., n - 1. No sorprende que cada cubo i tenga el mismo valor, ya que cada es igualmente probable que el valor en la matriz de entrada A caiga en cualquier segmento. Para probar la ecuación (8.2) , definir variables aleatorias del indicador

X_{ij} = I { A [j] cae en el cubo i }

para i = 0, 1,. . ., n - 1 yj = 1, 2,. . ., n . Así,

Para calcular, expandimos el cuadrado y reagrupamos los términos:

Página 155

(8,3)

donde la última línea sigue por la linealidad de la expectativa. Evaluamos las dos sumas por separado. La variable aleatoria del indicador X_{ij} es 1 con probabilidad $1 / n$ y 0 en caso contrario, y por lo tanto

Cuando $k \neq j$, las variables X_{ij} y X_{ik} son independientes, y por lo tanto

Sustituyendo estos dos valores esperados en la ecuación (8.3) , obtenemos

lo que demuestra la ecuación (8.2) .

Usando este valor esperado en la ecuación (8.1) , concluimos que el tiempo esperado para el cubo ordenar es $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$. Por lo tanto, todo el algoritmo de clasificación de cubos se ejecuta en tiempo esperado.

Incluso si la entrada no se extrae de una distribución uniforme, la clasificación de cubos aún puede ejecutarse en línea hora. Siempre que la entrada tenga la propiedad de que la suma de los cuadrados de los tamaños de cubeta sea

Página 156

lineal en el número total de elementos, la ecuación (8.1) nos dice que la clasificación de cubos se ejecutará en lineal hora.

Ejercicios 8.4-1

Utilizando la Figura 8.4 como modelo, ilustre la operación de BUCKET-SORT en el arreglo A = .79, .13, .16, .64, .39, .20, .89, .53, .71, .42.

Ejercicios 8.4-2

¿Cuál es el peor tiempo de ejecución para el algoritmo de clasificación de cubos? Qué simple cambio en el El algoritmo conserva su tiempo de ejecución lineal esperado y hace que su tiempo de ejecución en el peor de los casos $O(n \lg n)$?

Ejercicios 8.4-3

Sea X una variable aleatoria que es igual al número de caras en dos lanzamientos de una moneda justa. ¿Qué es $E[X^2]$? ¿Qué es $E^2[X]$?

Ejercicios 8.4-4: ⋆

Se nos dan n puntos en el círculo unitario, $p_i = (x_i, y_i)$, tales que para $i = 1, 2, \ldots, n$. Suponga que los puntos están distribuidos uniformemente; es decir, la probabilidad de encontrar un punto en cualquier región del círculo es proporcional al área de esa región. Diseñe un tiempo esperado de $\Theta(n)$ algoritmo para ordenar los n puntos por sus distancias desde el origen. (*Sugerencia:* diseñe el tamaños de cubos en BUCKET-SORT para reflejar la distribución uniforme de los puntos en la unidad circulo.)

Ejercicios 8.4-5: ⋆

Una **función de distribución de probabilidad** $P(x)$ para una variable aleatoria X se define por $P(x) = \Pr\{X \leq x\}$. Supongamos que una lista de n variables aleatorias $X_1, X_2, \ldots, X_n$ se extrae de un continuo función de distribución de probabilidad P que es calculable en $O(1)$ tiempo. Muestre cómo ordenar estos números en tiempo lineal esperado.

Página 157

Problemas 8-1: límites inferiores de casos promedio en la clasificación por comparación

En este problema, probamos un límite inferior $\Omega(n \lg n)$ en el tiempo de ejecución esperado de cualquier Orden de comparación determinista o aleatoria en n elementos de entrada distintos. Empezamos por el examen de una comparación de orden determinista A con el árbol de decisión T_A. Asumimos que cada permutación de A insumos 's tiene la misma probabilidad.

a. Suponga que cada hoja de T_A está etiquetada con la probabilidad de que se alcance dada una entrada aleatoria. Demuestre que exactamente $n!$ las hojas están etiquetadas como $1/n!$ y que el resto estan etiquetados 0.

si. Sea $D(T)$ la longitud del camino externo de un árbol de decisión T; es decir, $D(T)$ es la suma de las profundidades de todas las hojas de t. Sea T un árbol de decisión con $k > 1$ hojas, y sea LT y RT sean los subárboles izquierdo y derecho del T. Muestre que $D(T) = D(LT) + D(RT) + k$.

C. Sea $d(k)$ el valor mínimo de $D(T)$ sobre todos los árboles de decisión T con $k > 1$ hojas. Muestre que $d(k) = \min_{1 \le i \le k-1} \{ d(i) + d(k - i) + k \}$. (*Sugerencia:* considere un árbol de decisión T con k deja que logre el mínimo. Sea i_0 el número de hojas en LT y $k - i_0$ el número de hojas en RT.)

re. Demuestre que para un valor dado de $k > 1$ e i en el rango $1 \le ik - 1$, la función $i \lg i + (k - i) \lg (k - i)$ se minimiza en $i = k / 2$. Concluya que $d(k) = \Theta(k \lg k)$.

mi. Demuestre que $D(T_A) = \Theta(n! \operatorname{Lg}(n!))$, Y concluya que el tiempo esperado para ordenar n elementos es $\Theta(n \lg n)$.

Ahora, considere una *aleatorizado* de comparación de orden B. Podemos extender el modelo de árbol de decisiones a manejar la aleatorización incorporando dos tipos de nodos: nodos de comparación ordinarios y Nodos de "aleatorización". Un nodo de aleatorización modela una elección aleatoria de la forma RANDOM (1, r) realizado por el algoritmo B; el nodo tiene r hijos, cada uno de los cuales es igualmente probable a elegir durante la ejecución del algoritmo.

F. Demuestre que para cualquier tipo de comparación aleatoria B, existe un determinista comparación de orden A que no hace más comparaciones en el promedio de B hace.

Problemas 8-2: Clasificación en el lugar en tiempo lineal

Suponga que tenemos una matriz de n registros de datos para ordenar y que la clave de cada registro tiene la valor 0 o 1. Un algoritmo para ordenar tal conjunto de registros podría poseer algún subconjunto del siguientes tres características deseables:

1. El algoritmo se ejecuta en tiempo $O(n)$.
2. El algoritmo es estable.
3. El algoritmo se ordena en su lugar, utilizando no más que una cantidad constante de espacio de almacenamiento en además de la matriz original.

a. Proporcione un algoritmo que satisfaga los criterios 1 y 2 anteriores.
si. Proporcione un algoritmo que satisfaga los criterios 1 y 3 anteriores.
C. Proporcione un algoritmo que satisfaga los criterios 2 y 3 anteriores.

Página 158

re. ¿Puede usarse alguno de sus algoritmos de clasificación de las partes (a) - (c) para clasificar n registros con b -? claves de bits que utilizan ordenación por radix en tiempo $O(bn)$? Explique cómo o por qué no.

mi. Suponga que los n registros tienen claves en el rango de 1 a k. Muestre cómo modificar contando ordenar para que los registros se puedan ordenar en su lugar en el tiempo $O(n + k)$. Puedes utilizar $O(k)$ almacenamiento fuera de la matriz de entrada. ¿Tu algoritmo es estable? (*Pista:* ¿Cómo hacerlo por $k = 3$?)

Problemas 8-3: Clasificación de elementos de longitud variable

a. Se le da una matriz de enteros, donde diferentes enteros pueden tener diferentes números de dígitos, pero el número total de dígitos de *todos* los números enteros de la matriz es n. Muestre cómo ordenar la matriz en tiempo $O(n)$.

si. Se le da una matriz de cadenas, donde diferentes cadenas pueden tener diferentes números de caracteres, pero el número total de caracteres en todas las cadenas es n. Muestre cómo ordenar las cadenas en tiempo $O(n)$.

(Tenga en cuenta que el orden deseado aquí es el orden alfabético estándar; por ejemplo, a <ab <b.)

Problemas 8-4: Jarras de agua

Suponga que le dan n jarras de agua rojas $y n$ azules, todas de diferentes formas y tamaños. Todas Las jarras rojas contienen diferentes cantidades de agua, al igual que las azules. Además, por cada jarra roja, hay una jarra azul que contiene la misma cantidad de agua y viceversa.

Su tarea es encontrar una agrupación de jarras en pares de jarras rojas y azules que contengan la misma Cantidad de agua. Para ello, puede realizar la siguiente operación: coger un par de jarras en cuál es rojo y cuál es azul, llene la jarra roja con agua y luego vierta el agua en el jarra azul. Esta operación le dirá si la jarra roja o azul puede contener más agua, o si son del mismo volumen. Suponga que tal comparación toma una unidad de tiempo. Tu meta es para encontrar un algoritmo que haga un número mínimo de comparaciones para determinar el agrupamiento. Recuerde que no puede comparar directamente dos jarras rojas o dos azules.

a. Describe un algoritmo determinista que usa comparaciones $\Theta(n^2)$ para agrupar las jarras en pares.
si. Demuestre un límite inferior de $\Theta(n \lg n)$ para el número de comparaciones que resuelve un algoritmo este problema debe hacer.
C. Dar un algoritmo aleatorio cuyo número esperado de comparaciones sea $O(n \lg n)$, y demuestre que este límite es correcto. ¿Cuál es el peor número de comparaciones para tu algoritmo?

Problemas 8-5: clasificación promedio

Página 159

Suponga que, en lugar de ordenar una matriz, solo requerimos que los elementos aumenten en promedio. Más precisamente, llamamos a una matriz de n elementos A k-ordenada si, para todo $i = 1, 2, \ldots, n - k$, el siguientes retenciones:

a. ¿Qué significa que una matriz esté ordenada en 1?
si. Da una permutación de los números 1, 2,. . ., 10 que está clasificado en 2, pero no clasificado.
C. Demuestre que una matriz de n elementos está k- ordenada si y sólo si $A[i] \le A[i + k]$ para todo $i = 1, 2, \ldots, n - k$.
re. Proporcione un algoritmo que ordene k una matriz de n elementos en el tiempo $O(n \lg(n / k))$.

También podemos mostrar un límite inferior en el tiempo para producir una matriz ordenada k, cuando k es una constante.

mi. Demuestre que una matriz k- ordenada de longitud n se puede ordenar en $O(n \lg k)$ tiempo. (*Sugerencia:* use el solución al ejercicio 6.5-8 .)
F. Muestre que cuando k es una constante, se requiere $\Theta(n \lg n)$ tiempo para ordenar k una matriz de n elementos. (*Sugerencia:* use la solución de la parte anterior junto con el límite inferior en la comparación ordena.)

Problemas 8-6: límite inferior en la fusión de listas ordenadas

El problema de fusionar dos listas ordenadas surge con frecuencia. Se utiliza como subrutina de MERGE-SORT, y el procedimiento para fusionar dos listas ordenadas se da como MERGE en la Sección 2.3.1. En este problema, mostraremos que hay un límite inferior de $2n - 1$ en el peor de los casos número de comparaciones necesarias para fusionar dos listas ordenadas, cada una de las cuales contiene n elementos.

Primero mostraremos un límite inferior de $2n - o(n)$ comparaciones utilizando un árbol de decisión.

a. Demuestre que, dados $2n$ números, hay formas posibles de dividirlos en dos listas, cada una con n números.
si. Usando un árbol de decisión, muestre que cualquier algoritmo que combine correctamente dos listas ordenadas utiliza al menos $2n - o(n)$ comparaciones.

Ahora mostraremos un límite $2n - 1$ ligeramente más estrecho .

C. Muestre que si dos elementos son consecutivos en el orden ordenado y de listas opuestas, entonces deben compararse.

re. Utilice su respuesta a la parte anterior para mostrar un límite inferior de 2 n - 1 comparaciones para fusionando dos listas ordenadas.

Notas del capítulo

Ford y Johnson introdujeron el modelo de árbol de decisiones para estudiar tipos de comparación. [94] . El extenso tratado de Knuth sobre clasificación [185] cubre muchas variaciones sobre la clasificación problema, incluido el límite inferior de la teoría de la información sobre la complejidad de la clasificación dada aquí. Se estudiaron los límites inferiores para ordenar usando generalizaciones del modelo de árbol de decisión integralmente por Ben-Or [36] .

Knuth le da crédito a HH Seward por haber inventado la ordenación del conteo en 1954, y también por la idea de combinando ordenamiento de conteo con ordenamiento de base. Clasificación por radix comenzando con el dígito menos significativo parece ser un algoritmo popular ampliamente utilizado por los operadores de máquinas clasificadoras de tarjetas mecánicas. Según Knuth, la primera referencia publicada al método es un documento de 1929 de LJ Comrie describiendo el equipo de tarjetas perforadas. La clasificación por cubos se ha utilizado desde 1956, cuando la idea básica fue propuesta por EJ Isaac y RC Singleton.

Munro y Raman [229] proporcionan un algoritmo de ordenación estable que realiza comparaciones $O(n^{1+\epsilon})$ en el peor de los casos, donde $0 < \epsilon \leq 1$ es cualquier constante fija. Aunque cualquiera de los tiempos $O(n \lg n)$ los algoritmos hacen menos comparaciones, el algoritmo de Munro y Raman solo mueve datos $O(n)$ veces y opera en su lugar.

Muchos investigadores han considerado el caso de clasificar números enteros de nb bits en $o(n \lg n)$ tiempo. Se han obtenido varios resultados positivos, cada uno bajo supuestos ligeramente diferentes sobre el modelo de cálculo y las restricciones impuestas al algoritmo. Todos los resultados asumen que la memoria de la computadora está dividida en palabras direccionables de b bits. Fredman y Willard [99] introdujo la estructura de datos del árbol de fusión y la usó para ordenar n enteros en $O(n \lg n / \lg \lg n)$ tiempo. Este límite se mejoró más tarde para tiempo de Andersson [16] . Estos algoritmos requieren el uso de la multiplicación y varias constantes precalculadas. Andersson, Hagerup, Nilsson, y Raman [17] han mostrado cómo ordenar n enteros en $O(n \lg \lg n)$ tiempo sin usar multiplicación, pero su método requiere un almacenamiento que puede ser ilimitado en términos de n . Utilizando hash multiplicativo, se puede reducir el almacenamiento necesario a $O(n)$, pero el $O(n \lg \lg n)$ peor caso ligado en el tiempo de ejecución se convierte en un límite de tiempo esperado. Generalizando el árboles de búsqueda exponenciales de Andersson [16] , Thorup [297] dio una clasificación de tiempo $O(n (\lg \lg n)^2)$ algoritmo que no usa multiplicación ni aleatorización y usa espacio lineal. Combinatorio estas técnicas con algunas ideas nuevas, Han [137] mejoró el límite para clasificar a $O(n \lg \lg n \lg \lg \lg n)$ tiempo. Aunque estos algoritmos son importantes avances teóricos, son todo bastante complicado y en la actualidad parece poco probable que compita con la clasificación existente algoritmos en la práctica.

Capítulo 9: Medianas y estadísticas de orden

Visión general

La **estadística** de i- ésimo **orden** de un conjunto de n elementos es el i- ésimo elemento más pequeño. Por ejemplo, el **El mínimo** de un conjunto de elementos es la estadística de primer orden ($i = 1$), y el **máximo** es el n- ésimo estadística de orden ($i = n$). Una **mediana** , informalmente, es el "punto medio" del conjunto. Cuando n es impar, la mediana es única y ocurre en $i = (n + 1) / 2$. Cuando n es par, hay dos medianas, que ocurren en $i = n / 2$ $ei = n / 2 + 1$. Por lo tanto, independientemente de la paridad de n , las medianas ocurren en $i = \lfloor (n + 1) / 2 \rfloor$ (la **mediana inferior**) e $i = \lceil (n + 1) / 2 \rceil$ (la **mediana superior**). Por simplicidad en esto texto, sin embargo, usamos consistentemente la frase "la mediana" para referirnos a la mediana inferior.

Página 161

Este capítulo aborda el problema de seleccionar el estadístico de *i*- ésimo orden de un conjunto de *n* distintos números. Suponemos por conveniencia que el conjunto contiene números distintos, aunque prácticamente todo lo que hacemos se extiende a la situación en la que un conjunto contiene repetidos valores. El problema de selección se puede especificar formalmente de la siguiente manera:

Entrada: Un conjunto A de n números (distintos) y un número i , con $1 \le i \le n$.

Salida: El elemento x A que es más grande que exactamente i - 1 otros elementos de A.

El problema de selección se puede resolver en $O(n \lg n)$ tiempo, ya que podemos ordenar los números usando heapsort o merge sort y luego simplemente indexe el elemento i en la matriz de salida. Existen algoritmos más rápidos, sin embargo.

En la sección 9.1 , examinamos el problema de seleccionar el mínimo y el máximo de un conjunto de elementos. Más interesante es el problema de selección general, que se investiga en el las dos secciones siguientes. La sección 9.2 analiza un algoritmo práctico que logra un $O(n)$ limitado en el tiempo de ejecución en el caso medio. La sección 9.3 contiene un algoritmo de más interés teórico que logra el tiempo de ejecución $O(n)$ en el peor de los casos.

9.1 Mínimo y máximo

¿Cuántas comparaciones son necesarias para determinar el mínimo de un conjunto de n elementos? Nosotros puede obtener fácilmente un límite superior de n - 1 comparaciones: examine cada elemento del conjunto en gire y controle el elemento más pequeño visto hasta ahora. En el siguiente procedimiento, asumimos que el conjunto reside en la matriz A , donde *longitud* [A] = n .

```
MÍNIMO ( A )
1 min ← A [1]
2 para i ← 2 hasta la longitud [ A ]
3                 hacer si min > A [ i ]
4                                 luego min ← A [ i ]
5 minutos de retorno
```

Encontrar el máximo, por supuesto, también se puede lograr con n - 1 comparaciones.

Es esto lo mejor que podemos hacer? Sí, ya que podemos obtener un límite inferior de n - 1 comparaciones para el problema de determinar el mínimo. Piense en cualquier algoritmo que determine la mínimo como un torneo entre los elementos. Cada comparación es un partido del torneo. en el que gana el más pequeño de los dos elementos. La observación clave es que cada elemento excepto que el ganador debe perder al menos un partido. Por tanto, las comparaciones n - 1 son necesarias para determinar el mínimo, y el algoritmo MINIMO es óptimo con respecto al número de comparaciones realizadas.

Mínimo y máximo simultáneos

En algunas aplicaciones, debemos encontrar tanto el mínimo como el máximo de un conjunto de n elementos. Por ejemplo, un programa de gráficos puede necesitar escalar un conjunto de datos (x , y) para que quepan en un pantalla de visualización rectangular u otro dispositivo de salida gráfica. Para hacerlo, el programa debe primero determinar el mínimo y el máximo de cada coordenada.

Página 162

No es difícil idear un algoritmo que pueda encontrar tanto el mínimo como el máximo de n elementos utilizando comparaciones $\Theta(n)$, que es asintóticamente óptimo. Simplemente encuentra el mínimo y máximo de forma independiente, usando n - 1 comparaciones para cada uno, para un total de 2 n - 2 comparaciones.

De hecho, como máximo 3 $\lfloor n / 2 \rfloor$ comparaciones son suficientes para encontrar tanto el mínimo como el máximo. La estrategia es mantener los elementos mínimos y máximos vistos hasta ahora. En lugar de procesar cada elemento de la entrada comparándolo con el actual mínimo y máximo, a un costo de 2 comparaciones por elemento, procesamos elementos en pares. Primero comparamos pares de elementos de la entrada *entre sí* , y luego

comparar el menor con el mínimo actual y el mayor con el máximo actual, a un costo de 3 comparaciones por cada 2 elementos.

La configuración de valores iniciales para el mínimo y el máximo actuales depende de si n es impar o incluso. Si n es impar, establecemos tanto el mínimo como el máximo al valor del primer elemento, y luego procesamos el resto de los elementos por parejas. Si n es par, realizamos 1 comparación en los 2 primeros elementos para determinar los valores iniciales del mínimo y máximo, y luego procesar el resto de los elementos en pares como en el caso de n impar .

Analicemos el número total de comparaciones. Si n es impar, realizamos $3 \lfloor n / 2 \rfloor$ comparaciones. Si n es par, realizamos 1 comparación inicial seguida de $3 (n - 2) / 2$ comparaciones, para un total de $3 n / 2 - 2$. Por lo tanto, en cualquier caso, el número total de comparaciones es de la mayoría $3 \lfloor n / 2 \rfloor$.

Ejercicios 9.1-1

Demuestre que el segundo más pequeño de n elementos se puede encontrar con $n + \lceil \lg n \rceil - 2$ comparaciones en el peor caso. (*Sugerencia:* busque también el elemento más pequeño).

Ejercicios 9.1-2: ⋆

Demuestre que $\lceil 3 n / 2 \rceil - 2$ comparaciones son necesarias en el peor de los casos para encontrar tanto el máximo y mínimo de n números. (*Sugerencia:* considere cuántos números son potencialmente el máximo o mínimo, e investigue cómo una comparación afecta estos recuentos).

9.2 Selección en tiempo lineal esperado

El problema de selección general parece más difícil que el simple problema de encontrar un mínimo. Sin embargo, sorprendentemente, el tiempo de ejecución asintótico para ambos problemas es el mismo: $\Theta (n)$. En esta sección, presentamos un algoritmo de divide y vencerás para el problema de selección. los El algoritmo RANDOMIZED-SELECT se basa en el algoritmo de ordenación rápida del Capítulo 7 . Como en el ordenamiento rápido, la idea es dividir la matriz de entrada de forma recursiva. Pero a diferencia de Quicksort, que procesa de forma recursiva ambos lados de la partición, RANDOMIZED-SELECT solo funciona en

Página 163

un lado de la partición. Esta diferencia se muestra en el análisis: mientras que quicksort tiene una tiempo de ejecución esperado de $\Theta (n \lg n)$, el tiempo esperado de SELECCIÓN ALEATORIA es $\Theta (n)$.

RANDOMIZED-SELECT utiliza el procedimiento RANDOMIZED-PARTITION introducido en Sección 7.3. Por tanto, como RANDOMIZED-QUICKSORT, es un algoritmo aleatorio, ya que su el comportamiento está determinado en parte por la salida de un generador de números aleatorios. El seguimiento el código para RANDOMIZED-SELECT devuelve el i- ésimo elemento más pequeño de la matriz $A [p .. r]$.

```
SELECCIÓN ALEATORIZADA ( A , p , r , i )
1 si p = r
2 luego regresa A [ p ]
3 q ← PARTICIÓN ALEATORIZADA ( A , p , r )
4 k ← q - p + 1
5 si yo = k                          ▸ el valor pivote es la respuesta
6 luego regresa A [ q ]
7 elseif i < k
8 luego regrese RANDOMIZED-SELECT ( A , p , q - 1, i )
9 de lo contrario, devuelva SELECCIÓN ALEATORIA ( A , q + 1, r , i - k )
```

Después de que se ejecuta RANDOMIZED-PARTITION en la línea 3 del algoritmo, la matriz $A [p .. r]$ está dividido en dos subarreglos (posiblemente vacíos) $A [p .. q - 1]$ y $A [q + 1 .. r]$ de modo que cada elemento de $A [p .. q - 1]$ es menor o igual que $A [q]$, que a su vez es menor que cada elemento de $A [q + 1 .. r]$. Como en quicksort, nos referiremos a $A [q]$ como el elemento pivote . Línea 4 de RANDOMIZED-SELECT calcula el número k de elementos en el subarreglo $A [p .. q]$, que es decir, el número de elementos en el lado bajo de la partición, más uno para el elemento pivote. Línea 5 luego verifica si $A [q]$ es el i- ésimo elemento más pequeño. Si es así, se devuelve $A [q]$.

De lo contrario, el algoritmo determina en cuál de los dos subarreglos $A[p..q-1]$ y $A[q+1..r]$ se encuentra el i-ésimo elemento más pequeño. Si $i < k$, entonces el elemento deseado se encuentra en el lado bajo del partición, y se selecciona recursivamente del subarreglo en la línea 8. Si $i > k$, sin embargo, entonces el El elemento deseado se encuentra en el lado alto de la partición. Como ya conocemos k valores que son más pequeño que el i-ésimo elemento más pequeño de $A[p..r]$ —a saber, los elementos de $A[p..q]$ —el El elemento deseado es el $(i-k)$ ésimo elemento más pequeño de $A[q+1..r]$, que se encuentra recursivamente en línea 9. El código parece permitir llamadas recursivas a subarreglos con 0 elementos, pero Ejercicio 9.2-1 le pide que demuestre que esta situación no puede suceder.

El peor tiempo de ejecución para RANDOMIZED-SELECT es $\Theta(n^2)$, incluso para encontrar el mínimo, porque podríamos ser extremadamente desafortunados y siempre dividir alrededor de los elemento restante, y la partición toma $\Theta(n)$ tiempo. El algoritmo funciona bien en promedio caso, sin embargo, y debido a que es aleatorio, ninguna entrada en particular provoca el peor comportamiento de caso.

El tiempo requerido por RANDOMIZED-SELECT en una matriz de entrada $A[p..r]$ de n elementos es un variable aleatoria que denotamos por $T(n)$, y obtenemos un límite superior en $E[T(n)]$ como sigue. El procedimiento RANDOMIZED-PARTITION tiene la misma probabilidad de devolver cualquier elemento como pivote. Por lo tanto, para cada k tal que $1 \le k \le n$, el subarreglo $A[p..q]$ tiene k elementos (todos menores que o igual al pivote) con probabilidad $1/n$. Para $k = 1, 2, ..., n$, definimos indicador aleatorio variables X_k donde

$X_k = I\{\text{el subarreglo } A[p..q] \text{ tiene exactamente } k \text{ elementos}\}$,

y así tenemos

(9,1)

Página 164

Cuando llamamos RANDOMIZED-SELECT y elegimos $A[q]$ como elemento pivote, no saber, a priori, si terminamos inmediatamente con la respuesta correcta, recurrir al subarreglo $A[p..q-1]$, o recursivo en el subarreglo $A[q+1..r]$. Esta decisión depende de dónde el i-ésimo elemento más pequeño cae en relación con $A[q]$. Suponiendo que $T(n)$ aumenta monótonamente, podemos limitar el tiempo necesario para la llamada recursiva por el tiempo necesario para la llamada recursiva en la mayor entrada posible. En otras palabras, asumimos, para obtener un límite superior, que el i-ésimo El elemento siempre está del lado de la partición con el mayor número de elementos. Para una dada llamada de RANDOMIZED-SELECT, el indicador variable aleatoria X_k tiene el valor 1 para exactamente un valor de k, y es 0 para todos los demás k. Cuando $X_k = 1$, los dos subarreglos en los que podríamos los recursivos tienen tamaños $k-1$ y $n-k$. Por lo tanto, tenemos la recurrencia

Tomando los valores esperados, tenemos

Para aplicar la ecuación (C.23), confiamos en que X_k y $T(\max(k-1, n-k))$ sean independientes variables aleatorias. El ejercicio 9.2-2 le pide que justifique esta afirmación.

Consideremos la expresión $\max(k-1, n-k)$. Tenemos

Si n es par, cada término desde $T(\lceil n/2 \rceil)$ hasta $T(n-1)$ aparece exactamente dos veces en la suma,

y si n es impar, todos estos términos aparecen dos veces y $T(\lfloor n/2 \rfloor)$ aparece una vez. Por lo tanto, tenemos

Resolvemos la recurrencia por sustitución. Suponga que $T(n) \leq cn$ para alguna constante c que satisface las condiciones iniciales de la recurrencia. Suponemos que $T(n) = O(1)$ para n menor que alguna constante; elegiremos esta constante más tarde. También elegimos una constante a tal que función descrita por el término $O(n)$ anterior (que describe el componente no recursivo de el tiempo de ejecución del algoritmo) está delimitado desde arriba por un para todo $n > 0$. Usando este hipótesis inductiva, tenemos

Página 165

Para completar la demostración, necesitamos demostrar que para n suficientemente grande , este último expresión es como máximo cn o, de manera equivalente, que $cn / 4 - c / 2 - an \geq 0$. Si sumamos $c / 2$ a ambos lados y factorizar n , obtenemos $n (c / 4 - a) \geq c / 2$. Siempre que elijamos la constante c *de* modo que $c / 4 - a > 0$, es decir, $c > 4a$, podemos dividir ambos lados por $c / 4 - a$, dando

Por lo tanto, si asumimos que $T(n) = O(1)$ para $n < 2c / (c - 4a)$, tenemos $T(n) = O(n)$. Concluimos que cualquier orden estadístico, y en particular la mediana, se puede determinar en promedio en tiempo lineal.

Ejercicios 9.2-1

Muestre que en RANDOMIZED-SELECT, nunca se realiza una llamada recursiva a una matriz de longitud 0.

Ejercicios 9.2-2

Argumenta que la variable aleatoria del indicador X_k y el valor $T(\max(k - 1, n - k))$ son independiente.

Ejercicios 9.2-3

Página 166

Escriba una versión iterativa de RANDOMIZED-SELECT.

Ejercicios 9.2-4

Supongamos que usamos RANDOMIZED-SELECT para seleccionar el elemento mínimo de la matriz A = 3, 2, 9, 0, 7, 5, 4, 8, 6, 1. Describe una secuencia de particiones que resulte en el peor de los casos. rendimiento de RANDOMIZED-SELECT.

9.3 Selección en el tiempo lineal más desfavorable

Ahora examinamos un algoritmo de selección cuyo tiempo de ejecución es $O(n)$ en el peor de los casos. Me gusta RANDOMIZED-SELECT, el algoritmo SELECT encuentra el elemento deseado recursivamente particionando la matriz de entrada. La idea detrás del algoritmo, sin embargo, es *garantizar* una buena dividir cuando la matriz está particionada. SELECT usa el algoritmo de partición determinista PARTICIÓN de ordenación rápida (consulte la Sección 7.1), modificado para llevar el elemento a la partición alrededor como parámetro de entrada.

El algoritmo SELECT determina el i- ésimo más pequeño de una matriz de entrada de $n > 1$ elementos por ejecutando los siguientes pasos. (Si $n = 1$, SELECT simplemente devuelve su único valor de entrada como el i ésimo más pequeño.)

1. Divida los n elementos de la matriz de entrada en $\lfloor n / 5 \rfloor$ grupos de 5 elementos cada uno y en la mayoría de un grupo compuesto por los n mod 5 elementos restantes .
2. Encuentre la mediana de cada uno de los grupos $\lceil n / 5 \rceil$ por primera inserción ordenando los elementos de cada grupo (de los cuales hay como máximo 5) y luego seleccionando la mediana del orden lista de elementos del grupo.
3. Utilice SELECT de forma recursiva para encontrar la mediana x de las medianas $\lceil n / 5 \rceil$ encontradas en el paso 2. (Si hay un número par de medianas, entonces por nuestra convención, x es el menor mediana.)
4. Divida la matriz de entrada alrededor de la mediana de las medianas x utilizando la versión modificada de DIVIDIR. Sea k uno más que el número de elementos en el lado bajo del partición, de modo que x es el k- ésimo elemento más pequeño y hay $n - k$ elementos en el alto lado de la partición.
5. Si $i = k$, devuelve x . De lo contrario, use SELECT de forma recursiva para encontrar el i ésimo más pequeño elemento en el lado bajo si $i < k$, o el ($i - k$) th elemento más pequeño en el lado alto si $i > k$.

Para analizar el tiempo de ejecución de SELECT, primero determinamos un límite inferior en el número de elementos que son mayores que el elemento de partición x . La figura 9.1 es útil para visualizar esta contabilidad. Al menos la mitad de las medianas encontradas en el paso 2 son mayores que [1] la mediana de medianas x . Por tanto, al menos la mitad de los grupos $\lceil n / 5 \rceil$ aportan 3 elementos que son mayores que x , excepto para el grupo que tiene menos de 5 elementos si 5 no divide n exactamente, y el único grupo que contiene x mismo. Descontando estos dos grupos, se deduce que el número de elementos mayor que x es al menos

Página 167

Figura 9.1: Análisis del algoritmo SELECT. Los n elementos están representados por pequeños círculos, y cada grupo ocupa una columna. Las medianas de los grupos están blanqueadas y las la mediana de las medianas x está etiquetada. (Al encontrar la mediana de un número par de elementos, usamos la mediana inferior.) Las flechas se dibujan de elementos más grandes a más pequeños, de los cuales se puede ver que 3 de cada grupo completo de 5 elementos a la derecha de x son mayores que x , y 3 de cada grupo de 5 elementos a la izquierda de x son menores que x . Los elementos mayores que x se muestran sobre un fondo sombreado.

De manera similar, el número de elementos que son menores que x es al menos $3 n / 10 - 6$. Por lo tanto, en el peor En este caso, SELECT se llama de forma recursiva en un máximo de $7 n / 10 + 6$ elementos en el paso 5.

Ahora podemos desarrollar una recurrencia para el peor tiempo de ejecución $T (n)$ del algoritmo SELECCIONE. Los pasos 1, 2 y 4 toman $O (n)$ tiempo. (El paso 2 consta de $O (n)$ llamadas de ordenación por inserción en conjuntos de tamaño $O (1)$.) El paso 3 toma tiempo $T (\lceil n / 5 \rceil)$, y el paso 5 toma tiempo como máximo $T (7 n / 10 + 6)$, asumiendo que T aumenta monótonamente. Hacemos la suposición, que parece desmotivado al principio, que cualquier entrada de 140 elementos o menos requiere $O (1)$ tiempo; el origen de la constante mágica 140 se aclarará en breve. Por tanto, podemos obtener la recurrencia

Mostramos que el tiempo de ejecución es lineal por sustitución. Más específicamente, mostraremos que $T (n) \leq cn$ para alguna constante c adecuadamente grande y todo $n > 0$. Comenzamos asumiendo que $T (n) \leq cn$ para alguna constante c adecuadamente grande y todo $n \leq 140$; esta suposición se cumple si c es grande suficiente. También elegimos una constante a tal que la función descrita por el término $O (n)$ anterior (que describe el componente no recursivo del tiempo de ejecución del algoritmo) es acotado arriba por an para todo $n > 0$. Sustituyendo esta hipótesis inductiva en la derecha lado de los rendimientos de recurrencia

$$\begin{aligned} T (n) &\leq c \lceil n / 5 \rceil + c (7 n / 10 + 6) + an \\ &\leq cn / 5 + c + 7 cn / 10 + 6 c + an \\ &= 9 cn / 10 + 7 c + an \\ &= cn + (- cn / 10 + 7 c + an), \end{aligned}$$

Página 168

que es como mucho cn si

(9,2)

La desigualdad (9.2) es equivalente a la desigualdad $c \geq 10 a (n / (n - 70))$ cuando $n > 70$. Porque suponga que $n \geq 140$, tenemos $n / (n - 70) \leq 2$, por lo que elegir $c \geq 20 a$ satisfará la desigualdad (9.2) . (Tenga en cuenta que no hay nada especial en la constante 140; podríamos reemplazarla por cualquier entero estrictamente mayor que 70 y luego elija c *en* consecuencia.) El tiempo de ejecución del peor caso de SELECT es, por tanto, lineal.

Como en una clasificación de comparación (ver Sección 8.1), SELECT y RANDOMIZED-SELECT determinan información sobre el orden relativo de los elementos solo mediante la comparación de elementos. Recordar de Capítulo 8 que la clasificación requiere $\Omega (n \lg n)$ tiempo en el modelo de comparación, incluso en promedio (ver Problema 8-1). Los algoritmos de clasificación en tiempo lineal del capítulo 8 hacen suposiciones sobre entrada. Por el contrario, los algoritmos de selección de tiempo lineal de este capítulo no requieren suposiciones sobre la entrada. No están sujetos al límite inferior de $\Omega (n \lg n)$ porque Lograr resolver el problema de selección sin clasificar.

Por tanto, el tiempo de ejecución es lineal porque estos algoritmos no clasifican; el comportamiento del tiempo lineal no es el resultado de suposiciones sobre la entrada, como fue el caso de los algoritmos de clasificación en Capítulo 8 . La clasificación requiere un tiempo de $\Omega (n \lg n)$ en el modelo de comparación, incluso en promedio (ver Problema 8-1) y, por tanto, el método de clasificación e indexación presentado en la introducción a este capítulo es asintóticamente ineficiente.

Ejercicios 9.3-1

En el algoritmo SELECT, los elementos de entrada se dividen en grupos de 5. ¿El algoritmo trabajar en tiempo lineal si se dividen en grupos de 7? Argumenta que SELECT no se ejecuta en tiempo lineal si se utilizan grupos de 3.

Ejercicios 9.3-2

Analice SELECT para mostrar que si $n \geq 140$, entonces al menos $\lceil n / 4 \rceil$ elementos son mayores que el la mediana de las medianas x y al menos $\lceil n / 4 \rceil$ elementos son menores que x .

Ejercicios 9.3-3

Muestre cómo se puede ejecutar la ordenación rápida en tiempo $O (n \lg n)$ en el peor de los casos.

Ejercicios 9.3-4: ⋆

Página 169

Suponga que un algoritmo usa solo comparaciones para encontrar el i- ésimo elemento más pequeño en un conjunto de n elementos. Demuestre que también puede encontrar los elementos i - 1 más pequeños y los n - i elementos más grandes sin realizar comparaciones adicionales.

Ejercicios 9.3-5

Suponga que tiene una subrutina mediana de tiempo lineal en el peor de los casos de "caja negra". Dar un algoritmo simple de tiempo lineal que resuelve el problema de selección para una estadística de orden arbitrario.

Ejercicios 9.3-6

Los k- ésimos cuantiles de un conjunto de n elementos son las estadísticas de k -1 orden que dividen el conjunto ordenado en k *conjuntos de* igual tamaño (dentro de 1). Dar un algoritmo de tiempo $O (n \lg k)$ para enumerar los k- ésimos cuantiles de un conjunto.

Ejercicios 9.3-7

Describe un algoritmo de tiempo $O (n)$ que, dado un conjunto S de n números distintos y un número entero $k \leq n$, determina los k números en S que están más cerca de la mediana de S .

Ejercicios 9.3-8

Sean $X [1 .. n]$ e $Y [1 .. n]$ dos matrices, cada una de las cuales contiene n números ya ordenados.

Dé un $O(\lg n)$ -Tiempo algoritmo para encontrar la mediana de los 2 n elementos en las matrices X e Y .

Ejercicios 9.3-9

El profesor Olay es consultor para una compañía petrolera, que está planeando un gran oleoducto en funcionamiento de este a oeste a través de un campo petrolero de n pozos. Desde cada pozo, se conectará una tubería de derivación directamente a la tubería principal a lo largo de un camino más corto (norte o sur), como se muestra en la Figura 9.2 . Dado x - y y coordenadas x de los pozos, ¿cómo debe el profesor escoger la óptima ubicación de la tubería principal (la que minimiza la longitud total de las espuelas)? Muestra esa la ubicación óptima se puede determinar en tiempo lineal.

Página 170

Figura 9.2: El profesor Olay necesita determinar la posición del oleoducto este-oeste que minimiza la longitud total de las estribaciones norte-sur.

Problemas 9-1: números i más grandes en orden ordenado

Dado un conjunto de n números, deseamos encontrar el i más grande en orden ordenado usando una comparación: algoritmo basado. Encuentre el algoritmo que implementa cada uno de los siguientes métodos con el mejor tiempo de ejecución asintótico en el peor de los casos, y analizar los tiempos de ejecución de los algoritmos en términos de n y i .

a. Ordenar los números, y la lista de la i grande.
si. Cree una cola de máxima prioridad a partir de los números y llame a EXTRACT-MAX i veces.
C. Utilice un algoritmo estadístico de orden para encontrar el i- ésimo número más grande, particione alrededor de ese número y ordenar los i números más grandes.

Problemas 9-2: mediana ponderada

Para n elementos distintos $x_1, x_2, ..., x_n$ con pesos positivos $w_1, w_2, ..., w_n$ tales que , el la mediana ponderada (inferior) es el elemento $x_{k \text{ que}}$ satisface

y

a. Argumenta que la mediana de $x_1, x_2, ..., x_n$ es la mediana ponderada de x_i con pesos $w_i = 1 / n$ para $i = 1,2, ..., n$.
si. Muestre cómo calcular la mediana ponderada de n elementos en $O(n \lg n)$ tiempo en el peor de los casos usando clasificación.

Página 171

C. Muestre cómo calcular la mediana ponderada en el tiempo del peor de los casos (*n*) utilizando un método lineal algoritmo de mediana de tiempo como SELECT de la Sección 9.3.

El problema de la ubicación de la oficina postal se define de la siguiente manera. Se nos dan *n* puntos $p_1, p_2, ..., p_n$ con pesos asociados $w_1, w_2, ..., w_n$. Deseamos encontrar un punto *p* (no necesariamente uno de los puntos de entrada) que minimiza la suma , donde *d* (*a* , *b*) es la distancia entre puntos *una* y *b* .

re. Argumenta que la mediana ponderada es la mejor solución para la oficina de correos unidimensional problema de ubicación, en el que los puntos son simplemente números reales y la distancia entre señala *una* y *b* es *d* (*un* , *b*) = | *a* - *b* |.
mi. Encuentre la mejor solución para el problema de ubicación bidimensional de la oficina de correos, en el que los puntos son pares de coordenadas (*x* , *y*) y la distancia entre los puntos *a* = (x_1 , y_1) y *b* = (x_2 , y_2) es la distancia de Manhattan dada por *d* (*a* , *b*) = | x_1 - x_2 | + | y_1 - y_2 |.

Problemas 9-3: Estadísticas de pedidos pequeños

El número de caso más desfavorable *T* (*n*) de las comparaciones utilizadas por SELECT para seleccionar el estadístico de *i*- ésimo orden de *n* números se demostró que satisface *T* (*n*) = Θ (*n*), pero la constante oculta por la notación Θ es bastante grande. Cuando *i* es pequeño en relación con *n* , podemos implementar un procedimiento diferente que usa SELECCIONAR como una subrutina, pero hace menos comparaciones en el peor de los casos.

a. Describe un algoritmo que usa comparaciones U_i (*n*) para encontrar el *i*- ésimo más pequeño de *n* elementos, donde

(*Sugerencia:* comience con ⌊ *n* / 2⌋ comparaciones de pares disjuntos y repita en el conjunto que contiene el elemento más pequeño de cada par.)

si. Muestre que, si *i* < *n* / 2, entonces U_i (*n*) = *n* + *O* (*T* (2 *i*) lg (*n* / *i*)).
C. Demuestre que si *i* es una constante menor que *n* / 2, entonces U_i (*n*) = *n* + *O* (lg *n*).
re. Demuestre que si *i* = *n* / *k* para *k* ≥ 2, entonces U_i (*n*) = *n* + *O* (*T* (2 *n* / *k*) lg *k*).

[1] Debido a nuestra suposición de que los números son distintos, podemos decir "mayor que" y "menor que "sin preocuparse por la igualdad.

Notas del capítulo

El algoritmo de búsqueda de la mediana en tiempo lineal del peor de los casos fue ideado por Blum, Floyd, Pratt, Rivest y Tarjan [43] . La versión de tiempo medio rápido se debe a Hoare [146] . Floyd y Rivest

Página 172

[92] han desarrollado una versión mejorada de tiempo medio que se divide alrededor de un elemento seleccionados recursivamente de una pequeña muestra de los elementos.

Todavía se desconoce exactamente cuántas comparaciones se necesitan para determinar la mediana. UNA Bent y John [38] dieron el límite inferior de 2 *n* comparaciones para la mediana del hallazgo . Un El límite superior de 3 *n* lo dieron Schonhage, Paterson y Pippenger [265] . Dor y Zwick

[79] han mejorado en ambos límites; su límite superior es ligeramente menor que 2,95 N y el límite inferior es un poco más de 2 n . Paterson [239] describe estos resultados junto con otros trabajos relacionados.

Parte III: Estructuras de datos

Lista de capítulos

Capítulo 10: Estructuras de datos elementales
Capítulo 11: Tablas hash
Capítulo 12: Árboles de búsqueda binaria
Capítulo 13: Árboles Rojo-Negro
Capítulo 14: Aumento de las estructuras de datos

Introducción

Los conjuntos son tan fundamentales para la informática como lo son para las matemáticas. Mientras Los conjuntos matemáticos no cambian, los conjuntos manipulados por algoritmos pueden crecer, encogerse o de lo contrario, cambiará con el tiempo. A estos conjuntos los llamamos *dinámicos* . Los siguientes cinco capítulos presentan algunos técnicas básicas para representar conjuntos dinámicos finitos y manipularlos en una computadora.

Los algoritmos pueden requerir que se realicen varios tipos diferentes de operaciones en conjuntos. por Por ejemplo, muchos algoritmos solo necesitan la capacidad de insertar elementos, eliminar elementos de, y probar la pertenencia a un conjunto. Un conjunto dinámico que admite estas operaciones se denomina *diccionario* . Otros algoritmos requieren operaciones más complicadas. Por ejemplo, prioridad mínima colas, que se introdujeron en el Capítulo 6 en el contexto de la estructura de datos del montón, admiten las operaciones de insertar un elemento y extraer el elemento más pequeño de un conjunto. los La mejor forma de implementar un conjunto dinámico depende de las operaciones que se deben soportar.

Elementos de un conjunto dinámico

En una implementación típica de un conjunto dinámico, cada elemento está representado por un objeto cuyo Los campos se pueden examinar y manipular si tenemos un puntero al objeto. (Sección 10.3 analiza la implementación de objetos y punteros en entornos de programación que no no contenerlos como tipos de datos básicos.) Algunos tipos de conjuntos dinámicos suponen que uno de los Los campos del objeto es un campo *clave de* identificación . Si las claves son todas diferentes, podemos pensar en el conjunto dinámico como un conjunto de valores clave. El objeto puede contener *datos de satélite* , que son transportados en otros campos de objeto, pero no son utilizados por la implementación del conjunto. Eso también puede tener campos que son manipulados por las operaciones de conjunto; estos campos pueden contener datos o punteros a otros objetos del conjunto.

Algunos conjuntos dinámicos presuponen que las claves se extraen de un conjunto totalmente ordenado, como el números reales, o el conjunto de todas las palabras bajo el orden alfabético habitual. (Un conjunto totalmente ordenado satisface la propiedad de tricotomía, definida en la página 49.) Un orden total nos permite definir el elemento mínimo del conjunto, por ejemplo, o hablar del siguiente elemento más grande que un determinado elemento en un conjunto.

Operaciones en conjuntos dinámicos

Las operaciones en un conjunto dinámico se pueden agrupar en dos categorías: *consultas* , que simplemente devuelven información sobre el conjunto y *operaciones de modificación* que cambian el conjunto. Aquí hay una lista de operaciones típicas. Cualquier aplicación específica generalmente requerirá que solo algunos de estos sean implementado.

BÚSQUEDA (S , k)

- Una consulta que, dado un conjunto S y un valor clave k , devuelve un puntero x a un elemento en S de tal manera que *la tecla* [x] = k , o NIL si hay tal elemento pertenece a S .

INSERTAR (S , x)

- Una operación de modificación que aumenta el conjunto S con el elemento apuntado por x. Nosotros generalmente se asume que cualquier campo en el elemento x necesario para la implementación del conjunto tiene ya se ha inicializado.

BORRAR (S , x)

- Una operación de modificación que, dado un puntero x a un elemento del conjunto S , elimina x de S . (Tenga en cuenta que esta operación utiliza un puntero a un elemento x , no un valor clave).

MÍNIMO (S)

- Una consulta en un conjunto S totalmente ordenado que devuelve un puntero al elemento de S con el llave más pequeña.

MÁXIMO (S)

- Una consulta en un conjunto S totalmente ordenado que devuelve un puntero al elemento de S con el clave más grande.

SUCESOR (S , x)

- Una consulta que, dado un elemento x cuya clave es de un conjunto S totalmente ordenado , devuelve un puntero al siguiente elemento más grande en S , o NIL si x es el elemento máximo.

PREDECESOR (S , x)

- Una consulta que, dado un elemento x cuya clave es de un conjunto S totalmente ordenado , devuelve un puntero al siguiente elemento más pequeño en S , o NIL si x es el elemento mínimo.

Página 174

Las consultas SUCCESSOR y PREDECESSOR a menudo se extienden a conjuntos con no distintos llaves. Para un conjunto en n teclas, la presunción normal es que una llamada a MINIMUM seguida de n - 1 llamadas a SUCCESSOR enumera los elementos del conjunto en orden ordenado.

El tiempo necesario para ejecutar una operación de conjunto generalmente se mide en términos del tamaño del conjunto dado como uno de sus argumentos. Por ejemplo, el Capítulo 13 describe una estructura de datos que puede admite cualquiera de las operaciones enumeradas anteriormente en un conjunto de tamaño n en el tiempo O (lg n).

Resumen de la Parte III

Los capítulos 10 a 14 describen varias estructuras de datos que pueden usarse para implementar conjuntos dinámicos; muchos de estos se utilizarán más adelante para construir algoritmos eficientes para una variedad de problemas. Otra estructura de datos importante, el montón, ya se presentó en el Capítulo 6 .

El capítulo 10 presenta lo esencial para trabajar con estructuras de datos simples, como pilas, colas, listas enlazadas y árboles enraizados. También muestra cómo los objetos y punteros pueden ser implementados en entornos de programación que no los admiten como primitivas. Gran parte de Este material debe ser familiar para cualquiera que haya realizado una introducción a la programación. curso.

El capítulo 11 presenta tablas hash, que admiten las operaciones de diccionario INSERT, BORRAR y BUSCAR. En el peor de los casos, el hash requiere Θ (n) tiempo para realizar una BÚSQUEDA operación, pero el tiempo esperado para operaciones de tabla hash es O (1). El análisis de hash se basa en la probabilidad, pero la mayor parte del capítulo no requiere antecedentes en el tema.

Los árboles de búsqueda binaria, que se tratan en el Capítulo 12 , admiten todas las operaciones de conjuntos dinámicos listados arriba. En el peor de los casos, cada operación toma Θ (n) tiempo en un árbol con n elementos, pero en un árbol de búsqueda binario construido aleatoriamente, el tiempo esperado para cada operación es O (lg n). Binario Los árboles de búsqueda sirven como base para muchas otras estructuras de datos.

Los árboles rojo-negro, una variante de los árboles de búsqueda binaria, se presentan en el Capítulo 13 . diferente a árboles de búsqueda binaria ordinarios, se garantiza que los árboles rojo-negro funcionan bien: las operaciones toman O (lg n) tiempo en el peor de los casos. Un árbol rojo-negro es un árbol de búsqueda equilibrado; El capítulo 18 presenta otro tipo de árbol de búsqueda equilibrado, llamado árbol B. Aunque la mecánica del rojo-negro Los árboles son algo intrincados, puede obtener la mayoría de sus propiedades del capítulo sin

estudiando la mecánica en detalle. Sin embargo, recorrer el código puede resultar bastante instructivo.

En el Capítulo 14 , mostramos cómo aumentar los árboles rojo-negro para respaldar operaciones distintas de las los básicos enumerados anteriormente. Primero, los aumentamos para que podamos mantener dinámicamente el orden estadísticas para un conjunto de claves. Luego, los aumentamos de una manera diferente para mantener intervalos de numeros reales.

Capítulo 10: Estructuras de datos elementales

En este capítulo, examinamos la representación de conjuntos dinámicos mediante estructuras de datos simples que utilizar punteros. Aunque se pueden crear muchas estructuras de datos complejas utilizando punteros, presente solo los redimentarios: pilas, colas, listas enlazadas y árboles enraizados. Nosotros también discuta un método mediante el cual los objetos y punteros se pueden sintetizar a partir de matrices.

Página 175

10.1 Pilas y colas

Las pilas y las colas son conjuntos dinámicos en los que el elemento eliminado del conjunto por La operación DELETE está predefinida. En una pila , el elemento eliminado del conjunto es el insertada más recientemente: la pila implementa una política de último en entrar , primero en salir o LIFO . Del mismo modo, en una cola , el elemento eliminado es siempre el que ha estado en el conjunto durante más tiempo: el queue implementa una política de primero en entrar , primero en salir o FIFO . Hay varias formas eficientes de Implementar pilas y colas en una computadora. En esta sección mostramos cómo utilizar un sencillo matriz para implementar cada uno.

Pilas

La operación INSERT en una pila a menudo se llama PUSH, y la operación DELETE, que no toma un argumento de elemento, a menudo se llama POP. Estos nombres son alusiones a lo físico. pilas, como las pilas de platos con resorte que se utilizan en las cafeterías. El orden en que platos se extraen de la pila es el orden inverso al que se colocaron en el apilar, ya que solo se puede acceder a la placa superior.

Como se muestra en la Figura 10.1 , podemos implementar una pila de n elementos como máximo con una matriz S [1, n]. La matriz tiene un atributo top [S] que indexa el elemento insertado más recientemente. los la pila consta de elementos S [1 $superior$ [S]], donde S [1] es el elemento en la parte inferior de la pila y S [top [S]] es el elemento en la parte superior.

Figura 10.1: Una implementación array de una pila S . Los elementos de la pila aparecen solo en la posiciones sombreadas. *(a) La* pila S tiene 4 elementos. El elemento superior es 9. *(b)* Pila S después de las llamadas PUSH (S , 17) y PUSH (S , 3). *(c)* Apilar S después de que la llamada POP (S) haya devuelto el elemento 3, que es la más reciente. Aunque el elemento 3 todavía aparece en la matriz, no es más tiempo en la pila; la parte superior es el elemento 17.

Cuando top [S] = 0, la pila no contiene elementos y está vacía . La pila se puede probar para vacío por la operación de consulta STACK-EMPTY. Si aparece una pila vacía, decimos que stack underflows , que normalmente es un error. Si top [S] excede n , la pila se desborda . (En nuestro implementación de pseudocódigo, no nos preocupamos por el desbordamiento de la pila).

Cada una de las operaciones de pila se puede implementar con unas pocas líneas de código.

```
STACK-VACÍO ( S )
1 si top [ S ] = 0
2 luego devuelve VERDADERO
3 si no, devuelve FALSO

EMPUJE ( S , x )
1 arriba [ S ] ← arriba [ S ] + 1
2 S [ arriba [ S ]] ← x

POP ( S )
1 si PILA-VACÍA ( S )
2 luego error "subdesbordamiento"
```

Página 176

3 **más** *arriba* [*S*] ← *arriba* [*S*] - 1
4 **volver** *S* [*arriba* [*S*] + 1]

La Figura 10.1 muestra los efectos de las operaciones de modificación PUSH y POP. Cada uno de los tres las operaciones de pila toman *O* (1) tiempo.

Colas

Llamamos a la operación INSERT en una cola ENQUEUE, y llamamos a la operación DELETE DEQUEUE; como la operación de pila POP, DEQUEUE no toma ningún argumento de elemento. El FIFO La propiedad de una cola hace que funcione como una fila de personas en la oficina de registro. los La cola tiene una cabeza y una cola . Cuando un elemento se pone en cola, ocupa su lugar en la cola del cola, al igual que un estudiante recién llegado ocupa un lugar al final de la fila. El elemento Dequeued es siempre el que encabeza la cola, como el estudiante que encabeza la fila. quien ha esperado más tiempo. (Afortunadamente, no tenemos que preocuparnos por el cálculo elementos cortados en línea.)

La figura 10.2 muestra una forma de implementar una cola de como máximo *n* - 1 elementos usando una matriz *Q* [1 *n*]. La cola tiene un atributo *head* [*Q*] que indexa, o apunta a, su encabezado. El atributo *tail* [*Q*] indexa la siguiente ubicación en la que se insertará un elemento recién llegado en el cola. Los elementos de la cola están en las ubicaciones *head* [*Q*], *head* [*Q*] +1, ..., *tail* [*Q*] - 1, donde "envolvemos" en el sentido de que la ubicación 1 sigue inmediatamente a la ubicación *n* en un círculo orden. Cuando *head* [*Q*] = *tail* [*Q*], la cola está vacía. Inicialmente, tenemos *cabeza* [*Q*] = *cola* [*Q*] = 1. Cuando la cola está vacía, un intento de quitar de la cola un elemento hace que la cola se desborde. Cuando *head* [*Q*] = *tail* [*Q*] + 1, la cola está llena y un intento de poner en cola un elemento provoca la cola para desbordar.

Figura 10.2: Una cola implementada usando una matriz *Q* [1 12]. Los elementos de la cola aparecen solo en las posiciones ligeramente sombreadas. *(a)* La cola tiene 5 elementos, en las ubicaciones *Q* [7 11]. *(b)* El configuración de la cola después de las llamadas ENQUEUE (*Q* , 17), ENQUEUE (*Q* , 3) y ENQUEUE (*Q* , 5). *(c)* La configuración de la cola después de la llamada DEQUEUE (*Q*) devuelve el valor clave 15 anteriormente al principio de la cola. El nuevo cabezal tiene clave 6.

En nuestros procedimientos ENQUEUE y DEQUEUE, la comprobación de errores por subdesbordamiento y desbordamiento ha sido omitido. (El ejercicio 10.1-4 le pide que proporcione un código que verifique estos dos errores condiciones.)

ENQUEUE (*Q* , *x*)
1 *Q* [*cola* [*Q*]] ← *x*

Página 177

2 **si** *cola* [*Q*] = *longitud* [*Q*]
3 **luego** *cola* [*Q*] ← 1
4 **más** *cola* [*Q*] ← *cola* [*Q*] + 1

DEQUEUE (*Q*)

```
1 x ← Q [ cabeza [ Q ]]
2 si cabeza [ Q ] = longitud [ Q ]
3 luego dirígete [ Q ] ← 1
4 más cabeza [ Q ] ← cabeza [ Q ] + 1
5 devuelve x
```

La figura 10.2 muestra los efectos de las operaciones ENQUEUE y DEQUEUE. Cada operación toma $O(1)$ tiempo.

Ejercicios 10.1-1

Utilizando la Figura 10.1 como modelo, ilustre el resultado de cada operación en la secuencia PUSH (S , 4), PUSH (S , 1), PUSH (S , 3), POP (S), PUSH (S , 8) y POP (S) en una pila inicialmente vacía S almacenado en la matriz S [1 6].

Ejercicios 10.1-2

Explique cómo implementar dos pilas en una matriz A [1 n] de tal manera que ninguna pila se desborda a menos que el número total de elementos en ambas pilas juntas sea n . El PUSH y Las operaciones POP deben ejecutarse en tiempo $O(1)$.

Ejercicios 10.1-3

Utilizando la Figura 10.2 como modelo, ilustre el resultado de cada operación en la secuencia ENQUEUE (Q , 4), ENQUEUE (Q , 1), ENQUEUE (Q , 3), DEQUEUE (Q), ENQUEUE (Q , 8), y DEQUEUE (Q) en una cola Q inicialmente vacía almacenada en la matriz Q [16].

Ejercicios 10.1-4

Vuelva a escribir ENQUEUE y DEQUEUE para detectar subdesbordamiento y desbordamiento de una cola.

Ejercicios 10.1-5

Página 178

Mientras que una pila permite la inserción y eliminación de elementos en un solo extremo, y una cola permite inserción en un extremo y eliminación en el otro extremo, una *deque* (cola de dos extremos) permite inserción y eliminación en ambos extremos. Escriba cuatro procedimientos de tiempo $O(1)$ para insertar elementos en y eliminar elementos de ambos extremos de una deque construida a partir de una matriz.

Ejercicios 10.1-6

Muestre cómo implementar una cola usando dos pilas. Analizar el tiempo de ejecución de la cola operaciones.

Ejercicios 10.1-7

Muestre cómo implementar una pila usando dos colas. Analizar el tiempo de ejecución de la pila operaciones.

10.2 Listas vinculadas

Una lista vinculada es una estructura de datos en la que los objetos se organizan en un orden lineal. A diferencia de un matriz, sin embargo, en la que el orden lineal está determinado por los índices de matriz, el orden en una La lista enlazada está determinada por un puntero en cada objeto. Las listas enlazadas proporcionan una simple y flexible representación para conjuntos dinámicos, apoyando (aunque no necesariamente de manera eficiente) todas las operaciones enumeradas en la página 198.

Como se muestra en la Figura 10.3 , cada elemento de una lista L doblemente enlazada es un objeto con un campo *clave* y otros dos campos de puntero: *siguiente* y *anterior* . El objeto también puede contener otros datos de satélite. Dado un elemento x en la lista, *next* [x] apunta a su sucesor en la lista vinculada, y *prev* [x] apunta a su predecesor. Si *prev* [x] = NIL, el elemento x no tiene predecesor y por lo tanto es el primer elemento, o encabezado , de la lista. Si *next* [x] = NIL, el elemento x no tiene sucesor y es por lo tanto, el último elemento, o cola , de la lista. Un *encabezado de* atributo [L] apunta al primer elemento de la lista. Si *head* [L] = NIL, la lista está vacía.

Figura 10.3: *(a)* Una lista L doblemente enlazada que representa el conjunto dinámico {1, 4, 9, 16}. Cada elemento en la lista es un objeto con campos para la clave y punteros (mostrados por flechas) al objetos siguientes y anteriores. El *siguiente* campo de la cola y el campo *anterior* de la cabeza son NIL, indicado por una barra diagonal. El atributo *head* [L] apunta a la cabeza. *(b)* Siguiendo el ejecución de LIST-INSERT (L, x), donde *clave* [x] = 25, la lista vinculada tiene un nuevo objeto con clave 25 como el nuevo jefe. Este nuevo objeto apunta a la cabeza anterior con la clave 9. *(c)* El resultado de la llamada posterior LIST-DELETE (L, x), donde x apunta al objeto con la tecla 4.

Página 179

Una lista puede tener una de varias formas. Puede estar enlazado de forma simple o doble, puede estar ordenado o no, y puede ser circular o no. Si una lista está vinculada individualmente , omitimos la *anterior.* puntero en cada elemento. Si se ordena una lista , el orden lineal de la lista corresponde al lineal orden de claves almacenadas en elementos de la lista; el elemento mínimo es el encabezado de la lista, y el elemento máximo es la cola. Si la lista no está ordenada , los elementos pueden aparecer en cualquier orden. En una lista circular , el puntero *anterior* del encabezado de la lista apunta a la cola y el puntero *siguiente* de la cola de la lista apunta a la cabeza. Por tanto, la lista puede verse como un anillo de elementos. En En el resto de esta sección, asumimos que las listas con las que estamos trabajando no están ordenadas. y doblemente enlazados.

Buscando una lista vinculada

El procedimiento LIST-SEARCH (L, k) encuentra el primer elemento con la clave k en la lista L mediante un simple búsqueda lineal, devolviendo un puntero a este elemento. Si no aparece ningún objeto con la tecla k en la lista, luego se devuelve NIL. Para la lista enlazada en la Figura 10.3 (a) , la llamada LIST-SEARCH (L , 4) devuelve un puntero al tercer elemento y la llamada LIST-SEARCH (L , 7) devuelve NIL.

```
BÚSQUEDA EN LISTA ( L , k )
1 x ← cabeza [ L ]
2 mientras x ≠ NIL y la tecla [ x ] ≠ k
3 hacer x ← siguiente [ x ]
4 volver x
```

Para buscar una lista de n objetos, el procedimiento LIST-SEARCH toma Θ (n) tiempo en el peor de los casos, ya que puede tener que buscar en toda la lista.

Insertar en una lista vinculada

Dado un elemento x cuyo campo *clave* ya se ha establecido, el procedimiento LIST-INSERT "empalma" x al frente de la lista vinculada, como se muestra en la Figura 10.3 (b) .

```
INSERTAR LISTA ( L , x )
1 siguiente [ x ] ← cabeza [ L ]
2 si cabecera [ L ] ≠ NIL
3 luego anterior [ cabeza [ L ]] ← x
```

```
4 cabezas [ L ] ← x
5 anterior [ x ] ← NULO
```

El tiempo de ejecución de LIST-INSERT en una lista de n elementos es $O(1)$.

Eliminar de una lista vinculada

Elimina lista elimine el procedimiento un elemento x de una lista enlazada L . Se le debe dar un puntero ax , y luego "empalma" x fuera de la lista actualizando punteros. Si deseamos borrar un elemento con una clave dada, primero debemos llamar a LIST-SEARCH para recuperar un puntero al elemento.

```
BORRAR LISTA ( L , x )
1 si anterior [ x ] ≠ NIL
2 luego siguiente [ anterior [ x ]] ← siguiente [ x ]
3 else encabeza [ L ] ← siguiente [ x ]
4 si sigue [ x ] ≠ NIL
5 luego anterior [ siguiente [ x ]] ← anterior [ x ]
```

Página 180

La figura 10.3 (c) muestra cómo se elimina un elemento de una lista vinculada. LIST-DELETE se ejecuta en $O(1)$ tiempo, pero si deseamos borrar un elemento con una clave dada, se requiere $\Theta(n)$ tiempo en el en el peor de los casos porque primero debemos llamar a LIST-SEARCH.

Centinelas

El código para LIST-DELETE sería más simple si pudiéramos ignorar las condiciones de contorno en la cabeza y la cola de la lista.

```
LISTA-BORRAR ′ ( L , x )
1 siguiente [ anterior [ x ]] ← siguiente [ x ]
2 anterior [ siguiente [ x ]] ← anterior [ x ]
```

Un centinela es un objeto ficticio que nos permite simplificar las condiciones de contorno. Por ejemplo, supongamos que proporcionamos con la lista L un objeto *nil* [L] que representa NIL pero tiene todos los campos de los otros elementos de la lista. Siempre que tengamos una referencia a NIL en el código de lista, la reemplazamos por un referencia al centinela *cero* [L]. Como se muestra en la Figura 10.4 , esto convierte una lista regular doblemente enlazada en una lista circular, doblemente enlazada con un centinela , en la que se coloca el centinela *cero* [L] entre la cabeza y la cola; el campo *siguiente* [*nil* [L]] apunta al encabezado de la lista, y *anterior* [*nil* [L]] apunta a la cola. Del mismo modo, tanto el campo *siguiente* de la cola como el campo *anterior* del punto de la cabeza a *cero* [L]. Dado que el *siguiente* [*nil* [L]] apunta a la cabeza, podemos eliminar el atributo *cabeza* [L] en conjunto, reemplazando las referencias a él por referencias al *siguiente* [*cero* [L]]. Una lista vacía consta de solo el centinela, ya que tanto el *siguiente* [*nulo* [L]] como el *anterior* [*nulo* [L]] se pueden establecer en *cero* [L].

Figura 10.4: Una lista circular, doblemente enlazada con un centinela. El centinela *cero* [L] aparece entre la cabeza y la cola. El atributo *head* [L] ya no es necesario, ya que podemos acceder al head de la lista *junto a* [*nil* [L]]. *(a)* Una lista vacía. *(b)* La lista enlazada de la Figura 10.3 (a), con la clave 9 en la cabeza y la llave 1 en la cola. *(c)* La lista después de ejecutar LIST-INSER ′ (L , x), donde *clave* [x] = 25. El nuevo objeto se convierte en el encabezado de la lista. *(d)* La lista después de eliminar el objeto con la tecla 1. La nueva cola es el objeto con la clave 4.

El código para LIST-SEARCH sigue siendo el mismo que antes, pero con las referencias a NIL y *la cabeza* [L] cambió como se especifica arriba.

```
LIST-SEARC ′ ( L , k )
1 x ← siguiente [ cero [ L ]]
2 mientras x ≠ cero [ L ] y la tecla [ x ] ≠ k
3 hacer x ← siguiente [ x ]
4 volver x
```

Usamos el procedimiento de dos líneas LIST-DELET ′ para eliminar un elemento de la lista. Usamos el siguiente procedimiento para insertar un elemento en la lista.

```
LISTA DE INSERCIÓN ′ ( L , x )
1 siguiente [ x ] ← siguiente [ cero [ L ]]
2 anterior [ siguiente [ nulo [ L ]]] ← x
```

Página 181

```
3 siguiente [ cero [ L ]] ← x
4 anterior [ x ] ← cero [ L ]
```

La figura 10.4 muestra los efectos de LIST-INSER ′ y LIST-DELET ′ en una lista de muestra.

Los centinelas rara vez reducen los límites de tiempo asintóticos de las operaciones de la estructura de datos, pero pueden reducir factores constantes. La ganancia de usar centinelas dentro de los bucles suele ser una cuestión de claridad de código en lugar de velocidad; el código de la lista enlazada, por ejemplo, se simplifica mediante el uso de centinelas, pero solo ahorramos O (1) tiempo en los procedimientos LIST-INSER ′ y LIST-DELET ′. En En otras situaciones, sin embargo, el uso de centinelas ayuda a ajustar el código en un bucle, por lo que reduciendo el coeficiente de, digamos, n o n_2 en el tiempo de ejecución.

Los centinelas no deben usarse indiscriminadamente. Si hay muchas listas pequeñas, el almacenamiento adicional utilizado por sus centinelas puede representar una importante pérdida de memoria. En este libro usamos centinelas solo cuando realmente simplifican el código.

Ejercicios 10.2-1

¿Se puede implementar la operación INSERT de conjunto dinámico en una lista enlazada individualmente en tiempo O (1)? ¿Qué tal BORRAR?

Ejercicios 10.2-2

Implementar una pila utilizando una lista enlazada L . Las operaciones PUSH y POP aún deberían tomar O (1) tiempo.

Ejercicios 10.2-3

Implementar una cola por una lista enlazada L . Las operaciones ENQUEUE y DEQUEUE todavía debería tomar O (1) tiempo.

Ejercicios 10.2-4

Como está escrito, cada iteración de bucle en el procedimiento LIST-SEARC ′ requiere dos pruebas: una para $x \neq$ *cero* [L] y uno para la *tecla* [x] $\neq k$. Muestre cómo eliminar la prueba para $x \neq$ *nil* [L] en cada iteración.

Ejercicios 10.2-5

Página 182

Implemente las operaciones de diccionario INSERT, DELETE y SEARCH usando enlaces sencillos, listas circulares. ¿Cuáles son los tiempos de ejecución de sus procedimientos?

Ejercicios 10.2-6

La operación de conjuntos dinámicos UNION toma dos conjuntos disjuntos S_1 y S_2 como entrada, y devuelve un establecer $S = S_1 \cup S_2$ que consta de todos los elementos de S_1 y S_2 . Los conjuntos S_1 y S_2 suelen ser destruido por la operación. Muestre cómo apoyar a UNION en $O(1)$ tiempo usando una lista adecuada estructura de datos.

Ejercicios 10.2-7

Proporcione un procedimiento no recursivo de tiempo $\Theta(n)$ que invierta una lista enlazada de n elementos. los El procedimiento no debería utilizar más que un almacenamiento constante más allá del necesario para la lista misma.

Ejercicios 10.2-8:

Explique cómo implementar listas doblemente enlazadas usando solo un valor de puntero $np[x]$ por elemento en lugar de los dos habituales (*siguiente* y *anterior*). Suponga que todos los valores de puntero se pueden interpretar como k -bits enteros, y definir $np[x]$ como $np[x] = siguiente[x]$ XOR $prev[x]$, el k -bit "exclusivo-o" de $siguiente[x]$ y $anterior[x]$. (El valor NIL está representado por 0.) Asegúrese de describir qué información es necesario para acceder al encabezado de la lista. Muestre cómo implementar la BÚSQUEDA, INSERTAR y BORRAR operaciones en dicha lista. También muestre cómo invertir dicha lista en el tiempo $O(1)$.

10.3 Implementación de punteros y objetos

¿Cómo implementamos punteros y objetos en lenguajes, como Fortran, que no proporcionan ¿ellos? En esta sección, veremos dos formas de implementar estructuras de datos vinculadas sin un tipo de datos de puntero explícito. Sintetizaremos objetos y punteros a partir de matrices y matrices índices.

Una representación de objetos de matriz múltiple

Podemos representar una colección de objetos que tienen los mismos campos usando una matriz para cada campo. Como ejemplo, la Figura 10.5 muestra cómo podemos implementar la lista enlazada de Figura 10.3 (a) con tres matrices. La *clave de* matriz contiene los valores de las claves actualmente en la dinámica set, y los punteros se almacenan en las matrices *next* y *prev* . Para un índice de matriz dado x , $clave[x]$, $next[x]$ y $prev[x]$ representan un objeto en la lista vinculada. Bajo esta interpretación, un puntero x es simplemente un índice común en las matrices *key* , *next* y *prev* .

Página 183

Figura 10.5: La lista enlazada de la Figura 10.3 (a) representada por la *clave de* matrices *, siguiente* y *anterior* . Cada segmento vertical de las matrices representa un solo objeto. Los punteros almacenados corresponden a la índices de matriz mostrados en la parte superior; las flechas muestran cómo interpretarlas. Objeto ligeramente sombreado las posiciones contienen elementos de lista. La variable L mantiene el índice de la Cabeza.

En la Figura 10.3 (a) , el objeto con la clave 4 sigue al objeto con la clave 16 en la lista vinculada. En Figura 10.5 , la clave 4 aparece en la *clave* [2] y la clave 16 aparece en la *clave* [5], por lo que tenemos *siguiente* [5] = 2 y

prev [2] = 5. Aunque la constante NIL aparece en el *siguiente* campo de la cola y el campo *anterior* de la cabeza, usualmente usamos un número entero (como 0 o -1) que posiblemente no pueda representar un indexar en las matrices. Una variable *L* contiene el índice del encabezado de la lista.

En nuestro pseudocódigo, hemos estado usando corchetes para denotar tanto la indexación de un matriz y la selección de un campo (atributo) de un objeto. De cualquier manera, los significados de la *clave* [*x*], *siguiente* [*x*] y *anterior* [*x*] son coherentes con la práctica de implementación.

Una representación de una matriz única de objetos

Las palabras en la memoria de una computadora generalmente se direccionan mediante números enteros de 0 a *M* - 1, donde *M* es un número entero suficientemente grande. En muchos lenguajes de programación, un objeto ocupa un conjunto contiguo de ubicaciones en la memoria de la computadora. Un puntero es simplemente la dirección del la primera ubicación de memoria del objeto, y otras ubicaciones de memoria dentro del objeto pueden ser indexado agregando un desplazamiento al puntero.

Podemos utilizar la misma estrategia para implementar objetos en entornos de programación que no no proporcionar tipos de datos de puntero explícitos. Por ejemplo, la Figura 10.6 muestra cómo una sola matriz *A* se puede utilizar para almacenar la lista enlazada de las Figuras 10.3 (a) y 10.5 . Un objeto ocupa un subconjunto contiguo *A* [*jk*]. Cada campo del objeto corresponde a un desplazamiento en el rango de 0 a *k* - *j* , y un puntero al objeto es el índice *j* . En la Figura 10.6 , las compensaciones correspondientes a *key* , *next* y *prev* son 0, 1 y 2, respectivamente. Para leer el valor de *prev* [*i*], dado un puntero *i* , agregamos el valor *i* del puntero al desplazamiento 2, leyendo así *A* [*i* + 2].

Figura 10.6: La lista enlazada de las figuras 10.3 (A) y 10.5 representados en una única matriz *A* . Cada El elemento de lista es un objeto que ocupa un subarreglo contiguo de longitud 3 dentro del arreglo. los tres campos *key, next* y *prev* corresponden a las compensaciones 0, 1 y 2, respectivamente. Un puntero a un objeto es un índice del primer elemento del objeto. Los objetos que contienen elementos de lista son sombreado ligeramente, y las flechas muestran el orden de la lista.

Página 184

La representación de matriz única es flexible porque permite que objetos de diferentes longitudes sean almacenado en la misma matriz. El problema de gestionar una colección de objetos tan heterogénea es más difícil que el problema de gestionar una colección homogénea, donde todos los objetos tienen los mismos campos. Dado que la mayoría de las estructuras de datos que consideraremos se componen de elementos homogéneos, será suficiente para nuestros propósitos utilizar la matriz múltiple representación de objetos.

Asignar y liberar objetos

Para insertar una clave en un conjunto dinámico representado por una lista doblemente enlazada, debemos asignar un puntero a un objeto no utilizado actualmente en la representación de lista vinculada. Por tanto, es útil gestionar el almacenamiento de objetos que no se utilizan actualmente en la representación de lista enlazada de modo que se puede asignar. En algunos sistemas, un *recolector de basura* es responsable de determinar qué los objetos no se utilizan. Muchas aplicaciones, sin embargo, son lo suficientemente simples como para soportar responsabilidad de devolver un objeto no utilizado a un administrador de almacenamiento. Ahora exploraremos el problema de asignar y liberar (o desasignar) objetos homogéneos usando el ejemplo de una lista doblemente enlazada representada por múltiples matrices.

Suponga que las matrices en la representación de matrices múltiples tienen una longitud *my* que en algún momento el conjunto dinámico contiene $n \le m$ elementos. Entonces *n* objetos representan elementos actualmente en el conjunto dinámico, y los restantes $m - n$ objetos son *libres* ; los objetos libres se pueden utilizar para representan elementos insertados en el conjunto dinámico en el futuro.

Mantenemos los objetos libres en una lista enlazada individualmente, que llamamos *lista libre* . La lista gratuita utiliza solo la *siguiente* matriz, que almacena los *siguientes* punteros dentro de la lista. El encabezado de la lista gratuita es mantenido en la variable global *libre* . Cuando el conjunto dinámico representado por la lista enlazada *L* es no vacía, la lista libre puede estar entrelazada con la lista *L* , como se muestra en la Figura 10.7 . Tenga en cuenta que cada El objeto de la representación está en la lista *L* o en la lista libre, pero no en ambas.

Figura 10.7: El efecto de los procedimientos ALLOCATE-OBJECT y FREE-OBJECT. *(a)* El lista de la Figura 10.5 (ligeramente sombreada) y una lista libre (muy sombreada). Las flechas muestran la lista libre estructura. *(b)* El resultado de llamar a ALLOCATE-OBJECT () (que devuelve el índice 4), estableciendo *tecla* [4] a 25, y llamando a LIST-INSERT (L , 4). El nuevo encabezado de lista libre es el objeto 8, que tenía sido el *siguiente* [4] en la lista gratuita. *(c)* Después de ejecutar LIST-DELETE (L , 5), llamamos FREE-OBJETO (5). El objeto 5 se convierte en el nuevo encabezado de lista libre, con el objeto 8 siguiéndolo en el lista.

La lista libre es una pila: el siguiente objeto asignado es el último liberado. Podemos usar una lista implementación de las operaciones de pila PUSH y POP para implementar los procedimientos para asignar y liberar objetos, respectivamente. Suponemos que la variable global *free* utilizada en los siguientes procedimientos apuntan al primer elemento de la lista libre.

Página 185

```
ALLOCATE-OBJECT ()
1 si es gratis = NULO
2 luego error "sin espacio"
3 más x ← gratis
4                   gratis ← siguiente [ x ]
5                   volver x
OBJETO LIBRE ( x )
1 siguiente [ x ] ← gratis
2 gratis ← x
```

La lista libre contiene inicialmente todos los n objetos no asignados. Cuando la lista gratuita se haya agotado, el procedimiento ALLOCATE-OBJECT señala un error. Es habitual utilizar una única lista gratuita para servicio de varias listas enlazadas. La figura 10.8 muestra dos listas enlazadas y una lista libre entrelazadas a través de *las* matrices *key, next* y *prev* .

Figura 10.8: Dos listas enlazadas, L_1 (ligeramente sombreada) y L_2 (muy sombreada), y una lista libre (oscurecido) entrelazado.

Los dos procedimientos se ejecutan en tiempo $O(1)$, lo que los hace bastante prácticos. Ellos pueden ser modificado para funcionar con cualquier colección homogénea de objetos al permitir que cualquiera de los campos el objeto actúa como un campo *siguiente* en la lista libre.

Ejercicios 10.3-1

Haga un dibujo de la secuencia 13, 4, 8, 19, 5, 11 almacenada como una lista doblemente enlazada usando el representación de matriz múltiple. Haga lo mismo para la representación de matriz única.

Ejercicios 10.3-2

Escriba los procedimientos ALLOCATE-OBJECT y FREE-OBJECT para una homogénea colección de objetos implementados por la representación de matriz única.

Ejercicios 10.3-3

¿Por qué no necesitamos establecer o restablecer los campos *prev* de los objetos en la implementación del ¿Procedimientos ALLOCATE-OBJECT y FREE-OBJECT?

Ejercicios 10.3-4

Página 186

A menudo es deseable mantener todos los elementos de una lista doblemente enlazada compacta en almacenamiento, usando, por ejemplo, las primeras *m* ubicaciones de índice en la representación de matriz múltiple. (Este es el caso en un entorno informático de memoria virtual paginada.) Explique cómo se asignan los procedimientos> - OBJETO y OBJETO LIBRE se pueden implementar para que la representación sea compacta. Suponga que no hay punteros a elementos de la lista vinculada fuera de la propia lista. (*Pista:* Utilice la implementación de matriz de una pila).

Ejercicios 10.3-5

Sea *L* una lista doblemente enlazada de longitud *m* almacenada en matrices *key* , *prev* y *next* de longitud *n* . Supongamos que estas matrices son administradas por ALLOCATE-OBJECT y FREE-OBJECT procedimientos que mantienen una lista doblemente enlazada libre *F* . Suponga además que de los *n* elementos, exactamente *m* están en la lista *L* y *nm* están en la lista libre. Escriba un procedimiento COMPACTIFICAR-LISTA (*L* , *F*) que, dada la lista *L* y la lista libre *F* , mueve los elementos en *L* para que ocupen las posiciones de la matriz 1, 2, ..., *my* ajusta la lista libre *F* para que permanezca correcta, ocupando las posiciones de la matriz *m* + 1, *m* + 2, ..., *n* . El tiempo de ejecución de su procedimiento debe ser Θ (*m*), y debe usar solo un cantidad constante de espacio extra. Dé un argumento cuidadoso para la corrección de su procedimiento.

10.4 Representación de árboles enraizados

Los métodos para representar listas dados en la sección anterior se extienden a cualquier homogéneo estructura de datos. En esta sección, analizamos específicamente el problema de representar árboles enraizados por estructuras de datos vinculadas. Primero miramos árboles binarios, y luego presentamos un método para árboles enraizados en los que los nodos pueden tener un número arbitrario de hijos.

Representamos cada nodo de un árbol por un objeto. Al igual que con las listas enlazadas, asumimos que cada nodo contiene un campo *clave* . Los campos de interés restantes son indicadores de otros nodos y varían según el tipo de árbol.

Árboles binarios

Como se muestra en la Figura 10.9 , usamos los campos *p* , *izquierda* y *derecha* para almacenar punteros al padre, izquierda hijo, y el hijo derecho de cada nodo en un árbol binario *T* . Si *p* [*x*] = NIL, entonces *x* es la raíz. Si nodo *x* no tiene hijo izquierdo, entonces *left* [*x*] = NIL, y de manera similar para el hijo derecho. La raíz de todo el árbol *T* es señalado por el atributo *raíz* [*T*]. Si *root* [*T*] = NIL, entonces el árbol está vacío.

Página 187

Figura 10.9: La representación de un árbol binario T . Cada nodo x tiene los campos p [x] (arriba), *izquierda* [x] (parte inferior izquierda) y *derecha* [x] (parte inferior derecha). Los *principales* campos no se muestran.

Árboles enraizados con ramificación ilimitada

El esquema para representar un árbol binario se puede extender a cualquier clase de árboles en los que el El número de hijos de cada nodo es como máximo una constante k : reemplazamos la *izquierda* y la *derecha* campos por $hijo_1$, $hijo_2$, ..., $hijo_k$. Este esquema ya no funciona cuando el número de niños de un nodo es ilimitado, ya que no sabemos cuántos campos (matrices en la matriz múltiple representación) para asignar por adelantado. Además, incluso si el número de hijos k está acotado por una gran constante, pero la mayoría de los nodos tienen una pequeña cantidad de hijos, podemos desperdiciar una gran cantidad de memoria.

Afortunadamente, existe un esquema inteligente para usar árboles binarios para representar árboles con arbitrarias número de niños. Tiene la ventaja de usar solo espacio O (n) para cualquier árbol con raíz de n nodos. La representación del hijo izquierdo y del hermano derecho se muestra en la Figura 10.10 . Como antes, cada nodo contiene un puntero padre p , y la *raíz* [T] apunta a la raíz del árbol T . En lugar de tener un puntero a cada uno de sus hijos, sin embargo, cada nodo x tiene solo dos punteros:

Figura 10.10: El niño-izquierda, derecha-hermanos representación de un árbol T . Cada nodo x tiene campos p [x] (arriba), *hijo izquierdo* [x] (abajo a la izquierda) y *hermano derecho* [x] (abajo a la derecha). No se muestran las claves.

1. *El hijo izquierdo* [x] apunta al hijo situado más a la izquierda del nodo x , y
2. *hermano derecho* [x] señala al hermano de x inmediatamente a la derecha.

Si el nodo x no tiene hijos, entonces *el hijo izquierdo* [x] = NIL, y si el nodo x es el hijo más a la derecha de su padre, luego *hermano derecho* [x] = NULO.

Otras representaciones de árboles

Página 188

A veces representamos árboles enraizados de otras formas. En el capítulo 6 , por ejemplo, representamos un montón, que se basa en un árbol binario completo, mediante una única matriz más un índice. Los árboles que aparecen en el Capítulo 21 se recorren solo hacia la raíz, por lo que solo los punteros principales son presente; no hay consejos para los niños. Son posibles muchos otros esquemas. Que esquema es lo mejor depende de la aplicación.

Ejercicios 10.4-1

Dibuje el árbol binario con raíz en el índice 6 que está representado por los siguientes campos.

índice	tecla de	izquierda	derecha
	t		

1 12 7 3
2 15 8 NULO
3 4 10 NULO
4 10 5 9
5 2 NIL NIL
6 18 1 4
7 7 NIL NIL
8 14 6 2
9 21 NIL NIL
10 5 NULO NULO

Ejercicios 10.4-2

Escriba un procedimiento recursivo de tiempo $O(n)$ que, dado un árbol binario de n nodos, imprima la clave de cada nodo del árbol.

Ejercicios 10.4-3

Escriba un procedimiento no recursivo de tiempo $O(n)$ que, dado un árbol binario de n nodos, imprima el clave de cada nodo del árbol. Utilice una pila como estructura de datos auxiliar.

Ejercicios 10.4-4

Escriba un procedimiento de tiempo $O(n)$ que imprima todas las claves de un árbol con raíces arbitrarias con n nodos, donde se almacena el árbol utilizando la representación del hijo izquierdo y del hermano derecho.

Página 189

Ejercicios 10.4-5:

Escriba un procedimiento no recursivo de tiempo $O(n)$ que, dado un árbol binario de n nodos, imprima el clave de cada nodo. No use más que un espacio extra constante fuera del árbol mismo y no modificar el árbol, incluso temporalmente, durante el procedimiento.

Ejercicios 10.4-6:

La representación del hijo izquierdo y del hermano derecho de un árbol con raíces arbitrarias utiliza tres punteros en cada nodo: *hijo izquierdo, hermano derecho* y *padre* . Desde cualquier nodo, se puede acceder a su padre y identificado en tiempo constante y todos sus hijos pueden ser alcanzados e identificados en el tiempo lineal en el número de hijos. Muestre cómo usar solo dos punteros y un valor booleano en cada nodo para que el padre de un nodo o todos sus hijos puedan ser alcanzados e identificados a tiempo lineal en el número de hijos.

Problemas 10-1: Comparaciones entre listas

Para cada uno de los cuatro tipos de listas de la siguiente tabla, ¿cuál es el peor caso asintótico? tiempo de ejecución para cada operación de conjunto dinámico enumerada?

	sin clasificar, individualmente vinculado	ordenado, individualmente vinculado	sin clasificar, doblemente vinculado	clasificado, doblemente vinculado
BÚSQUEDA (L , k)				
INSERTAR (L , x)				
BORRAR (L , x)				
SUCESOR (L , x)				
PREDECESOR (L , x)				
MÍNIMO (L)				
MÁXIMO (L)				

Problemas 10-2: montones fusionables mediante listas enlazadas

Un *montón fusionable* admite las siguientes operaciones: MAKE-HEAP (que crea un montón fusionable), INSERT, MINIMUM, EXTRACT-MIN y UNION. [1] Muestre cómo implemente montones fusionables usando listas enlazadas en cada uno de los siguientes casos. Tratar de hacer

Página 190

cada operación lo más eficiente posible. Analizar el tiempo de ejecución de cada operación en términos de el tamaño de los conjuntos dinámicos sobre los que se está operando.

a. Las listas están ordenadas.
si. Las listas no están ordenadas.
C. Las listas no están ordenadas y los conjuntos dinámicos que se van a fusionar están separados.

Problemas 10-3: búsqueda en una lista compacta ordenada

El ejercicio 10.3-4 preguntó cómo podríamos mantener una lista de n elementos de manera compacta en el primer n posiciones de una matriz. Supondremos que todas las claves son distintas y que la lista compacta es también ordenado, es decir, *clave* [i] < *clave* [*siguiente* [i]] para todo i = 1, 2, ..., n tal que *siguiente* [i] ≠ NIL. Debajo estas suposiciones, demostrará que el siguiente algoritmo aleatorio se puede utilizar para buscar la lista en tiempo esperado.

```
BÚSQUEDA-LISTA-COMPACTA ( L , n , k )
 1 i ← cabeza [ L ]
 2 mientras i ≠ NIL y la tecla [ i ] < k
 3 hacer j ← ALEATORIO (1, n )
 4                 si tecla [ i ] < tecla [ j ] y tecla [ j ] ≤ k
 5                       entonces yo ← j
 6                                 si clave [ i ] = k
 7                                       luego regrese yo
 8                       i ← siguiente [ i ]
 9 si i = NIL o clave [ i ]> k
10 luego devuelve NIL
11 más regreso i
```

Si ignoramos las líneas 3 a 7 del procedimiento, tenemos un algoritmo ordinario para buscar un lista enlazada, en la que el índice i apunta a cada posición de la lista sucesivamente. La búsqueda termina una vez que el índice i "cae" al final de la lista o una vez, la *tecla* [i] ≥ k . En el último caso, si *clave* [i] = k , claramente hemos encontrado una clave con el valor k . Sin embargo, si la *tecla* [i]> k , nunca encontraremos un con el valor k , por lo que finalizar la búsqueda fue lo correcto.

Las líneas 3 a 7 intentan saltar hacia una posición elegida al azar j . Tal salto es beneficioso si *la clave* [j] es mayor que la *clave* [i] y no mayor que k ; en tal caso, j marca una posición en la lista que *yo* tendría que alcanzar durante la búsqueda de la lista ordinaria. Porque la lista es compacta, sabemos que cualquier elección de j entre 1 y n indexa algún objeto en la lista en lugar de una ranura en el lista libre.

En lugar de analizar el rendimiento de COMPACT-LIST-SEARCH directamente, vamos a

analizar un algoritmo relacionado, COMPACT-LIST-SEARC ′, que ejecuta dos ciclos separados. Este algoritmo toma un parámetro adicional t que determina un límite superior en el número de iteraciones del primer ciclo.

```
BUSCAR-LISTA-COMPACTA ′ ( L , n , k , t )
 1 i ← cabeza [ L ]
 2 para q ← 1 a t
```

Página 191

```
 3 hacer j ← ALEATORIO (1, n )
 4                 si tecla [ i ] < tecla [ j ] y tecla [ j ] ≤ k
 5                     entonces yo ← j
 6                             si clave [ i ] = k
 7                                 luego regrese yo
 8 mientras i ≠ NIL y la tecla [ i ] < k
 9 ¿ i ← siguiente [ i ]
10 si i = NULO o clave [ i ]> k
11 luego devuelve NIL
12 más regrese yo
```

Comparar la ejecución de los algoritmos COMPACT-LIST-SEARCH (L , k) y COMPACT-LIST-SEARC ′ (L , k , t), suponga que la secuencia de enteros devueltos por las llamadas de RANDOM (1, n) es el mismo para ambos algoritmos.

a. Supongamos que COMPACT-List-SEARCH (L , k) realiza t iteraciones del **mientras** bucle de líneas 2-8. Argumenta que COMPACT-LIST-SEARC ′ (L , k , t) devuelve la misma respuesta y que el número total de iteraciones tanto de la **para** y **mientras** bucles dentro Compact-LIST-SEARC ′ es al menos t .

En la llamada COMPACT-LIST-SEARC ′ (L , k , t), sea X_i la variable aleatoria que describe la distancia en la lista vinculada (es decir, a través de la cadena de punteros *siguientes*) desde la posición i al clave deseada k después de que se hayan producido t iteraciones del bucle **for** de las líneas 2-7.

si. Argumente que el tiempo de ejecución esperado de COMPACT-LIST-SEARC ′ (L , k , t) es O (t + E [X_i]).
C. Muestra esa . (*Sugerencia:* utilice la ecuación (C.24) .)
re. Muestra esa .
mi. Demuestre que E [X_i] ≤ n / (t + 1).
F. Demuestre que COMPACT-LIST-SEARC ′ (L , k , t) se ejecuta en O (t + n / t) tiempo esperado.
gramo. Concluya que COMPACT-LIST-SEARCH se ejecuta en tiempo esperado.
h. ¿Por qué suponemos que todas las claves son distintas en COMPACT-LIST-SEARCH? Discutir que los saltos aleatorios no necesariamente ayudan de forma asintótica cuando la lista contiene valores clave repetidos.

[1] Como hemos definido un montón fusionable para admitir MINIMUM y EXTRACT-MIN, también podemos referirnos a él como min-heap fusionable . Alternativamente, si es compatible con MAXIMUM y EXTRACT-MAX, sería un montón máximo fusionable .

Notas del capítulo

Aho, Hopcroft y Ullman [6] y Knuth [182] son excelentes referencias para datos elementales. estructuras. Muchos otros textos cubren tanto las estructuras de datos básicas como su implementación en un lenguaje de programación particular. Ejemplos de este tipo de libros de texto incluyen Goodrich y Tamassia [128] , Main [209] , Shaffer [273] y Weiss [310 , 312 , 313] . Gonnet [126] proporciona datos experimentales sobre el rendimiento de muchas operaciones de estructura de datos.

El origen de las pilas y colas como estructuras de datos en la informática no está claro, ya que las nociones correspondientes ya existían en matemáticas y prácticas comerciales basadas en papel antes de la introducción de las computadoras digitales. Knuth [182] cita a AM Turing para el desarrollo de pilas para el enlace de subrutinas en 1947.

Página 192

Las estructuras de datos basadas en punteros también parecen ser una invención popular. Según Knuth, punteros aparentemente se utilizaron en las primeras computadoras con memorias de batería. El lenguaje A-1 desarrollado por GM Hopper en 1951 representó fórmulas algebraicas como árboles binarios. Knuth acredita el Lenguaje IPL-II, desarrollado en 1956 por A. Newell, JC Shaw y HA Simon, para reconociendo la importancia y promoviendo el uso de punteros. Su lenguaje IPL-III, desarrollado en 1957, incluía operaciones de pila explícitas.

Capítulo 11: Tablas hash

Visión general

Muchas aplicaciones requieren un conjunto dinámico que admita solo las operaciones del diccionario. INSERTAR, BUSCAR y ELIMINAR. Por ejemplo, un compilador para un lenguaje informático. mantiene una tabla de símbolos, en la que las claves de los elementos son cadenas de caracteres arbitrarias que corresponden a identificadores en el idioma. Una tabla hash es una estructura de datos eficaz para implementación de diccionarios. Aunque buscar un elemento en una tabla hash puede llevar tanto tiempo como buscar un elemento en una lista vinculada, — (n) tiempo en el peor de los casos, en la práctica, hash funciona muy bien. Bajo supuestos razonables, el tiempo esperado para buscar un elemento en una tabla hash es O (1).

Una tabla hash es una generalización de la noción más simple de una matriz ordinaria. Abordar directamente en una matriz ordinaria hace un uso efectivo de nuestra capacidad para examinar una posición arbitraria en un matriz en tiempo O (1). La sección 11.1 analiza el direccionamiento directo con más detalle. Direccionamiento directo es aplicable cuando podemos permitirnos asignar una matriz que tiene una posición para cada posible llave.

Cuando el número de claves realmente almacenadas es pequeño en relación con el número total de claves posibles, las tablas hash se convierten en una alternativa eficaz para abordar directamente una matriz, ya que una tabla hash normalmente utiliza una matriz de tamaño proporcional al número de claves realmente almacenadas. En vez de utilizando la clave como un índice de matriz directamente, el índice de matriz se *calcula a* partir de la clave. Sección 11.2 presenta las ideas principales, centrándose en el "encadenamiento" como una forma de manejar "colisiones" en las que más de una clave se asigna al mismo índice de matriz. La sección 11.3 describe cómo los índices de matriz se puede calcular a partir de claves utilizando funciones hash. Presentamos y analizamos varias variaciones sobre el tema básico. La sección 11.4 analiza el "direccionamiento abierto", que es otra forma de tratar con colisiones. La conclusión es que el hash es una herramienta extremadamente efectiva y práctica. técnica: las operaciones básicas del diccionario requieren solo O (1) tiempo en promedio. Sección 11.5 explica cómo el "hash perfecto" puede admitir búsquedas en O (1) el *peor de los casos* , cuando el conjunto de las claves que se almacenan son estáticas (es decir, cuando el conjunto de claves nunca cambia una vez almacenado).

11.1 Tablas de direcciones directas

El direccionamiento directo es una técnica simple que funciona bien cuando el universo U de teclas está razonablemente pequeño. Suponga que una aplicación necesita un conjunto dinámico en el que cada elemento tiene una clave extraída del universo $U = \{0, 1, ..., m - 1\}$, donde m no es demasiado grande. Deberíamos suponga que no hay dos elementos que tengan la misma clave.

Para representar el conjunto dinámico, usamos una matriz o **tabla de direcciones directas** , denotada por $T[0 \; m - 1]$, en el que cada posición, o **de ranura** , corresponde a una tecla en el universo U . Figura 11.1

ilustra el enfoque; la ranura k apunta a un elemento del conjunto con la tecla k . Si el conjunto no contiene elemento con clave k , luego $T[k] =$ NIL.

Figura 11.1: Aplicación de un conjunto dinámico de una tabla de dirección directa T . Cada clave del universo $U = \{0, 1, ..., 9\}$ corresponde a un índice en la tabla. El conjunto $K = \{2, 3, 5, 8\}$ de claves reales determina los espacios en la tabla que contienen punteros a elementos. Las otras ranuras, fuertemente sombreado, contiene NIL.

Las operaciones del diccionario son triviales de implementar.

```
BÚSQUEDA DIRECTA DE DIRECCIÓN ( T , k )
    volver T [ k ]

INSERTAR DIRECCIÓN DIRECTA ( T , x )
    T [ tecla [ x ]] ← x

BORRAR DIRECCIÓN DIRECTA ( T , x )
    T [ tecla [ x ]] ← NULO
```

Cada una de estas operaciones es rápida: solo se requiere O (1) tiempo.

Para algunas aplicaciones, los elementos del conjunto dinámico se pueden almacenar en la dirección directa mesa en sí. Es decir, en lugar de almacenar la clave de un elemento y los datos satelitales en un objeto externo a la tabla de direcciones directas, con un puntero desde una ranura en la tabla al objeto, podemos almacenar el objeto en la propia ranura, ahorrando así espacio. Además, a menudo no es necesario almacenar la clave campo del objeto, ya que si tenemos el índice de un objeto en la tabla, tenemos su clave. Si llaves no están almacenados, sin embargo, debemos tener alguna forma de saber si la ranura está vacía.

Ejercicios 11.1-1

Suponga que un conjunto dinámico S está representado por una tabla de direcciones directas T de longitud m . Describe un procedimiento que encuentra el elemento de máxima de S . ¿Cuál es el peor rendimiento de su ¿procedimiento?

Ejercicios 11.1-2

Un *vector de bits* es simplemente una matriz de bits (0 y 1). Un vector de bits de longitud m tarda mucho menos espacio que una matriz de m punteros. Describir cómo utilizar un vector de bits para representar una dinámica.

Página 194

Conjunto de elementos distintos sin datos de satélite. Las operaciones de diccionario deben ejecutarse en O (1) Hora.

Ejercicios 11.1-3

Sugerir cómo implementar una tabla de direcciones directas en la que las claves de los elementos almacenados no deben ser distintos y los elementos pueden tener datos de satélite. Las tres operaciones de diccionario (INSERT, DELETE, and SEARCH) debe ejecutarse en O (1) tiempo. (No olvide que BORRAR toma como argumento un puntero a un objeto a eliminar, no una clave.)

Ejercicios 11.1-4: ⋆

Deseamos implementar un diccionario usando direccionamiento directo en una *gran* variedad. Al principio, las entradas de la matriz pueden contener basura, e inicializar toda la matriz no es práctico porque de su tamaño. Describe un esquema para implementar un diccionario de direcciones directas en una matriz enorme. Cada objeto almacenado debe usar espacio O (1); las operaciones BUSCAR, INSERTAR y ELIMINAR debe tomar O (1) vez cada uno; y la inicialización de la estructura de datos debería llevar O (1) tiempo. (*Sugerencia:* use una pila adicional, cuyo tamaño sea el número de claves realmente almacenadas en el diccionario, para ayudar a determinar si una entrada determinada en la gran matriz es válida o no).

11.2 Tablas hash

La dificultad con el direccionamiento directo es obvia: si el universo U es grande, almacenar una tabla T de tamaño $|U|$ puede ser impráctico, o incluso imposible, dada la memoria disponible en un típico computadora. Además, el conjunto K de claves *realmente almacenadas* puede ser tan pequeño en relación con U que la mayor parte del espacio asignado para T se desperdiciaría.

Cuando el conjunto K de claves almacenadas en un diccionario es mucho menor que el universo U de todos claves posibles, una tabla hash requiere mucho menos almacenamiento que una tabla de direcciones directas. Específicamente, los requisitos de almacenamiento se pueden reducir a $\Theta(|K|)$ mientras mantenemos el beneficio de que la búsqueda para un elemento en la tabla hash todavía requiere solo $O(1)$ tiempo. El único inconveniente es que este límite es para el *tiempo promedio* , mientras que para el direccionamiento directo se mantiene para el *peor de los casos* .

Con direccionamiento directo, un elemento con la clave k se almacena en la ranura k . Con hash, este elemento es almacenado en la ranura $h(k)$; es decir, usamos una **función hash** h para calcular la ranura a partir de la clave k . Aquí h mapea el universo U de claves en las ranuras de una **tabla hash** $T[0..m-1]$:

$h : U \rightarrow \{0, 1, ..., m - 1\}$.

Decimos que un elemento con clave k *tiene un* **hash** en la ranura $h(k)$; también decimos que $h(k)$ es el **valor hash** de la clave k . La figura 11.2 ilustra la idea básica. El objetivo de la función hash es reducir la rango de índices de matriz que deben manejarse. En lugar de $|U|$ valores, necesitamos manejar solo m valores. Los requisitos de almacenamiento se reducen en consecuencia.

Página 195

Figura 11.2: Uso de una función hash h para asignar claves a los espacios de la tabla hash. las teclas k_2 y $k_{5\,se}$ asignan al misma ranura, por lo que chocan.

Hay un problema: dos claves pueden introducirse en la misma ranura. A esta situación la llamamos **colisión** . Afortunadamente, existen técnicas efectivas para resolver el conflicto creado por las colisiones.

Por supuesto, la solución ideal sería evitar las colisiones por completo. Podríamos intentar lograr este objetivo eligiendo una función hash adecuada h . Una idea es hacer que h parezca "aleatorio", evitando así colisiones o al menos minimizando su número. El mismo término "hash", que evoca imágenes de mezcla y picado al azar, captura el espíritu de este enfoque. (Por supuesto, un La función hash h debe ser determinista en el sentido de que una entrada k dada siempre debe producir el mismo salida $h(k)$.) Dado que $|U| > m$, sin embargo, debe haber al menos dos claves que tengan el mismo hash valor; Por tanto, evitar las colisiones es imposible. Por lo tanto, aunque bien diseñado, La función hash de aspecto "aleatorio" puede minimizar el número de colisiones, todavía necesitamos una método para resolver las colisiones que ocurren.

El resto de esta sección presenta la técnica de resolución de colisiones más simple, llamada encadenamiento. La sección 11.4 presenta un método alternativo para resolver colisiones, llamado abierto direccionamiento.

Resolución de colisiones por encadenamiento

En el **encadenamiento** , colocamos todos los elementos que tienen hash en la misma ranura en una lista vinculada, como se muestra en Figura 11.3 . La ranura j contiene un puntero al encabezado de la lista de todos los elementos almacenados que tienen un hash en j ; si no existen tales elementos, la ranura j contiene NIL.

Figura 11.3: Resolución de colisiones por encadenamiento. Cada ranura de tabla hash $T[j]$ contiene una lista enlazada de todas las claves cuyo valor hash es j. Por ejemplo, $h(k_1) = h(k_4)$ y $h(k_5) = h(k_2) = h(k_7)$.

Las operaciones de diccionario en una tabla hash T son fáciles de implementar cuando las colisiones son resuelto por encadenamiento.

Página 196

```
INSERCIÓN DE HASH ENCADENADO ( T , x )
    inserte x al principio de la lista T [ h ( tecla [ x ])]

BÚSQUEDA DE HASH ENCADENADO ( T , k )
    buscar un elemento con la clave k en la lista T [ h ( k )]

BORRAR HASH ENCADENADO ( T , x )
    eliminar x de la lista T [ h ( tecla [ x ])]
```

El peor tiempo de ejecución para la inserción es $O(1)$. El procedimiento de inserción es rápido en parte porque supone que el elemento x *que* se inserta no está ya presente en la tabla; esta La suposición se puede verificar si es necesario (a un costo adicional) realizando una búsqueda antes inserción. Para la búsqueda, el tiempo de ejecución en el peor de los casos es proporcional a la longitud de la lista; analizaremos esta operación más de cerca a continuación. La supresión de un elemento x puede ser logrado en $O(1)$ tiempo si las listas están doblemente enlazadas. (Tenga en cuenta que CHAINED-HASH-DELETE toma como entrada un elemento x y no su clave k, por lo que no tenemos que buscar x primero. Si las listas estaban vinculadas individualmente, no sería de gran ayuda tomar como entrada el elemento x *en* lugar de que la clave k. Todavía tendríamos que encontrar x en la lista $T[h(clave[x])]$, por lo que el *siguiente* eslabón de x 's predecesor se podría configurar correctamente para empalmar x hacia fuera. En este caso, la eliminación y la búsqueda tienen esencialmente el mismo tiempo de ejecución).

Análisis de hash con encadenamiento

¿Qué tan bien funciona el hash con encadenamiento? En particular, ¿cuánto tiempo se tarda en buscar para un elemento con una clave determinada?

Dada una tabla hash T con m ranuras que almacena n elementos, definimos el factor de carga α para T como n / m, es decir, el número medio de elementos almacenados en una cadena. Nuestro análisis será en términos de α, que puede ser menor, igual o mayor que 1.

El peor comportamiento de hash con encadenamiento es terrible: todas las n claves se hash en la misma ranura, creando una lista de longitud n. El peor de los casos para la búsqueda es (n) más el tiempo para Calcule la función hash, no mejor que si usáramos una lista enlazada para todos los elementos. Claramente, las tablas hash no se utilizan en el peor de los casos. (Hashing perfecto, descrito en la Sección 11.5, sin embargo, proporciona un buen rendimiento en el peor de los casos cuando el conjunto de claves es estático.)

El rendimiento medio del hash depende de qué tan bien distribuye la función hash h conjunto de claves que se almacenarán entre las m ranuras, en promedio. La sección 11.3 analiza estos problemas, pero por ahora asumiremos que cualquier elemento dado tiene la misma probabilidad de hash en cualquiera de los m ranuras, independientemente de dónde se haya aplicado el hash de cualquier otro elemento. A esto lo llamamos la suposición de hash uniforme simple.

Para $j = 0, 1, ..., m - 1$, denotemos la longitud de la lista $T[j]$ por n_j, de modo que

(11,1)

y el valor medio de n_j es $E[n_j] = \alpha = n / m$.

Suponemos que el valor hash $h(k)$ se puede calcular en $O(1)$ tiempo, de modo que el tiempo requerido para la búsqueda de un elemento con la clave k depende linealmente de la longitud $n_{h(k)}$ de la lista $T[h(k)]$. Ajuste Aparte del tiempo $O(1)$ requerido para calcular la función hash y acceder a la ranura $h(k)$, permítanos

Página 197

considerar el número esperado de elementos examinados por el algoritmo de búsqueda, es decir, el número de elementos de la lista $T[h(k)]$ que se comprueban para ver si sus claves son iguales a k. Nosotros considerará dos casos. En el primero, la búsqueda no tiene éxito: ningún elemento de la tabla ha clave k. En el segundo, la búsqueda encuentra con éxito un elemento con la clave k.

Teorema 11.1

En una tabla hash en la que las colisiones se resuelven mediante encadenamiento, una búsqueda fallida lleva tiempo esperado $\Theta(1 + \alpha)$, bajo el supuesto de hash uniforme simple.

Prueba Bajo el supuesto de hash uniforme simple, cualquier clave k *que* no esté almacenada es igualmente probable que la tabla hash en cualquiera de las m ranuras. El tiempo esperado para buscar sin éxito para una clave k es el tiempo esperado para buscar al final de la lista $T[h(k)]$, que tiene longitud esperada $E[n_{h(k)}] = \alpha$. Por tanto, el número esperado de elementos examinados en un búsqueda fallida es α, y el tiempo total requerido (incluido el tiempo para calcular $h(k)$) es $\Theta(1 + \alpha)$.

La situación para una búsqueda exitosa es ligeramente diferente, ya que no es probable que cada lista ser registrado. En cambio, la probabilidad de que se busque en una lista es proporcional al número de elementos que contiene. No obstante, el tiempo de búsqueda esperado sigue siendo $\Theta(1 + \alpha)$.

Teorema 11.2

En una tabla hash en la que las colisiones se resuelven mediante el encadenamiento, una búsqueda exitosa lleva tiempo $\Theta(1 + \alpha)$, en promedio, bajo el supuesto de hash uniforme simple.

Prueba Suponemos que el elemento que se busca es igualmente probable que sea cualquiera de los n elementos almacenados en la tabla. El número de elementos examinados durante una búsqueda exitosa de un elemento x es 1 más que el número de elementos que aparecen antes de x en la lista de x. Elementos antes de x en la lista se insertaron todos después de que se insertó x, porque los nuevos elementos se colocan en al frente de la lista. Para encontrar el número esperado de elementos examinados, tomamos el promedio, sobre los n elementos x en la tabla, de 1 más el número esperado de elementos agregados a la lista de x después de que x se agregó a la lista. Sea x_i el i- ésimo elemento insertado en la tabla, para $i = 1, 2, \ldots, n$, y sea $k_i = clave[x_i]$. Para las claves k_i y k_j, definimos la variable aleatoria del indicador $X_{ij} = \text{Yo}\{h(k_i) = h(k_j)\}$. Bajo el supuesto de hash uniforme simple, tenemos $\Pr\{h(k_i) = h(k_j)\} = 1/m$, y así, según el Lema 5.1, $E[X_{ij}] = 1/m$. Por tanto, el número esperado de elementos examinados en una búsqueda exitosa es

Página 198

Por lo tanto, el tiempo total requerido para una búsqueda exitosa (incluido el tiempo para calcular el función hash) es $\Theta (2 + \alpha / 2 - \alpha / 2 n) = \Theta (1 + \alpha)$.

¿Qué significa este análisis? Si el número de ranuras de la tabla hash es al menos proporcional al número de elementos en la tabla, tenemos $n = O (m)$ y, en consecuencia, $\alpha = n / m = O (m) / m = O (1)$. Por tanto, la búsqueda lleva un tiempo constante en promedio. Dado que la inserción toma $O (1)$ en el peor de los casos tiempo y la eliminación toma $O (1)$ tiempo en el peor de los casos cuando las listas están doblemente vinculadas, todo el diccionario las operaciones se pueden realizar en $O (1)$ tiempo en promedio.

Ejercicios 11.2-1

Supongamos que usamos una función hash h para dividir n claves distintas en una matriz T de longitud m . Suponiendo hash uniforme simple, ¿cuál es el número esperado de colisiones? Más precisamente, ¿cuál es la cardinalidad esperada de $\{\{ k , l \}: k \neq l \text{ y } h (k) = h (l)\}$?

Ejercicios 11.2-2

Demuestre la inserción de las claves 5, 28, 19, 15, 20, 33, 12, 17, 10 en una tabla hash con colisiones resueltas por encadenamiento. Deje que la tabla tenga 9 espacios y que la función hash sea $h (k) = k \mod 9$.

Ejercicios 11.2-3

Página 199

El profesor Marley plantea la hipótesis de que se pueden obtener mejoras sustanciales en el rendimiento si modifique el esquema de encadenamiento para que cada lista se mantenga ordenada. ¿Cómo el profesor La modificación afecta el tiempo de ejecución para búsquedas exitosas, búsquedas no exitosas, inserciones, y eliminaciones?

Ejercicios 11.2-4

Sugerir cómo se puede asignar y desasignar el almacenamiento de elementos dentro de la propia tabla hash vinculando todas las ranuras no utilizadas en una lista gratuita. Suponga que una ranura puede almacenar una bandera y cualquiera elemento más un puntero o dos punteros. Todas las operaciones de diccionario y lista libre deben ejecutarse en $O (1)$ tiempo esperado. ¿Es necesario que la lista gratuita esté doblemente vinculada o haya una lista gratuita lista es suficiente?

Ejercicios 11.2-5

Demuestre que si $| U | > nm$, hay un subconjunto de U de tamaño n *que* consta de claves que todo hash al mismo intervalo, de modo que el peor tiempo de búsqueda de hash con encadenamiento es $\Theta (n)$.

11.3 Funciones hash

En esta sección, discutimos algunos problemas relacionados con el diseño de buenas funciones hash y luego presentan tres esquemas para su creación. Dos de los esquemas, hash por división y hash por multiplicación, son de naturaleza heurística, mientras que el tercer esquema, hash universal, utiliza aleatorización para proporcionar un rendimiento demostrablemente bueno.

¿Qué hace una buena función hash?

Una buena función hash satisface (aproximadamente) el supuesto de hash uniforme simple: cada clave tiene la misma probabilidad de hash en cualquiera de las ranuras m , independientemente de dónde se encuentre cualquier otra clave ha codificado. Desafortunadamente, normalmente no es posible verificar esta condición, ya que uno rara vez conoce la distribución de probabilidad según la cual se dibujan las claves, y las claves no se puede dibujar de forma independiente.

De vez en cuando conocemos la distribución. Por ejemplo, si se sabe que las claves son aleatorias números reales k distribuidos de forma independiente y uniforme en el rango $0 \le k < 1$, el hash función

$h (k) = \lfloor km \rfloor$

satisface la condición de hash uniforme simple.

Página 200

En la práctica, las técnicas heurísticas se pueden utilizar a menudo para crear una función hash que funcione bien. La información cualitativa sobre la distribución de claves puede ser útil en este proceso de diseño. por ejemplo, considere la tabla de símbolos de un compilador, en la que las claves son cadenas de caracteres que representan identificadores en un programa. Los símbolos estrechamente relacionados, como pt y pts, suelen aparecer en el mismo programa. Una buena función hash minimizaría la posibilidad de que tales variantes hash en la misma ranura.

Un buen enfoque es derivar el valor hash de una manera que se espera que sea independiente de cualquier patrones que pueden existir en los datos. Por ejemplo, el "método de división" (discutido en la Sección 11.3.1) calcula el valor hash como el resto cuando la clave se divide por un valor especificado número primo. Este método frecuentemente da buenos resultados, asumiendo que el número primo es elegido para no estar relacionado con ningún patrón en la distribución de claves.

Finalmente, observamos que algunas aplicaciones de funciones hash pueden requerir propiedades más fuertes que se proporcionan mediante un simple hash uniforme. Por ejemplo, podríamos querer claves que sean "close" en cierto sentido para producir valores hash que están muy separados. (Esta propiedad es especialmente deseable cuando utilizamos sondeo lineal, definido en la Sección 11.4.) Hash universal, descrito en la Sección 11.3.3 , a menudo proporciona las propiedades deseadas.

Interpretar claves como números naturales

La mayoría de las funciones hash asumen que el universo de claves es el conjunto $\mathbf{N} = \{0, 1, 2, ...\}$ de números. Por lo tanto, si las claves no son números naturales, se encuentra una manera de interpretarlas como naturales. números. Por ejemplo, una cadena de caracteres se puede interpretar como un entero expresado en notación de base. Por lo tanto, el identificador pt podría interpretarse como el par de enteros decimales (112, 116), ya que p = 112 y t = 116 en el juego de caracteres ASCII; luego, expresado como radix-128 entero, pt se convierte en $(112 \cdot 128) + 116 = 14452$. Por lo general, es sencillo en una aplicación para idear algún método de este tipo para interpretar cada clave como un número natural (posiblemente grande). En En lo que sigue, asumimos que las claves son números naturales.

11.3.1 El método de división

En el *método de división* para crear funciones hash, mapeamos una clave k en una de m ranuras por tomando el resto de k dividido por m . Es decir, la función hash es

$h (k) = k \bmod m$.

Por ejemplo, si la tabla hash tiene un tamaño $m = 12$ y la clave es $k = 100$, entonces $h (k) = 4$. Dado que requiere solo una operación de división única, el hash por división es bastante rápido.

Cuando usamos el método de división, generalmente evitamos ciertos valores de m . Por ejemplo, m debería no ser una potencia de 2, ya que si $m = 2^p$, entonces $h(k)$ son solo los p bits de orden más bajo de k . A menos que sea Si se sabe que todos los patrones de bits p *de* bajo orden son igualmente probables, es mejor hacer que la función hash Dependen de todos los bits de la clave. Como el ejercicio 11.3-3 le pide que muestre, elegir $m = 2^p - 1$ cuando k es una cadena de caracteres interpretada en la base 2^p puede ser una mala elección, porque permutar los caracteres de k no cambian su valor hash.

Un número primo no demasiado cercano a una potencia exacta de 2 suele ser una buena opción para m . Por ejemplo, Supongamos que deseamos asignar una tabla hash, con las colisiones resueltas por encadenamiento, para mantener aproximadamente $n = 2000$ cadenas de caracteres, donde un carácter tiene 8 bits. No nos importa examinar un promedio

Página 201

de 3 elementos en una búsqueda fallida, por lo que asignamos una tabla hash de tamaño $m = 701$. La se elige el número 701 porque es un primo cerca de 2000/3 pero no cerca de ninguna potencia de 2. Tratando cada tecla k como un número entero, nuestra función hash sería

$h(k) = k \bmod 701.$

11.3.2 El método de la multiplicación

El *método de multiplicación* para crear funciones hash opera en dos pasos. Primero nosotros multiplique la clave k por una constante A en el rango $0 < A < 1$ y extraiga la parte fraccionaria de kA . Luego, multiplicamos este valor por m*y* tomamos el piso del resultado. En resumen, la función hash es

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

donde " $kA \bmod 1$" significa la parte fraccionaria de kA , es decir, $kA - \lfloor kA \rfloor$.

Una ventaja del método de multiplicación es que el valor de m no es crítico. Normalmente elija que sea una potencia de 2 ($m = 2^p$ para algún entero p) ya que entonces podemos implementar fácilmente la función en la mayoría de las computadoras de la siguiente manera. Suponga que el tamaño de la palabra de la máquina es w bits y que k cabe en una sola palabra. Restringimos A para que sea una fracción de la forma $s / 2^w$, donde s es un número entero en el rango $0 < s < 2^w$. Con referencia a la figura 11.4 , primero multiplicamos k por el bit w entero $s = A \cdot 2^w$. El resultado es un valor de 2 w bits $r_1 2^w + r_0$, donde r_1 es la palabra de orden superior del producto y r_0 es la palabra de orden inferior del producto. El valor hash de p -bit deseado consiste en los p bits más significativos de r_0 .

Figura 11.4: El método de multiplicación de hash. La representación de w bits de la clave k es multiplicado por el valor de w -bit $s = A \cdot 2^w$. Los p bits de orden más alto de la mitad inferior de w -bit del producto forma el valor hash deseado $h(k)$.

Aunque este método funciona con cualquier valor de la constante A , funciona mejor con algunos valores que con otros. La elección óptima depende de las características de los datos que se Troceado. Knuth [185] sugiere que

(11,2)

es probable que funcione razonablemente bien.

Como ejemplo, suponga que tenemos $k = 123456$, $p = 14$, $m = 2^{14} = 16384$ y $w = 32$. Adaptando Sugerencia de Knuth, elegimos A para que sea la fracción de la forma $s / 2^{32}$ más cercana a , de modo que $A = 2654435769/2^{32}$. Entonces $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, y entonces $r_1 = 76300$ y $r_0 = 17612864$. Los 14 bits más significativos de r_0 producen el valor $h(k) = 67$.

Página 202

11.3.3 Hash universal

Si un adversario malintencionado elige las claves para ser codificadas por alguna función hash fija, entonces puede elegir n claves con hash en la misma ranura, lo que arroja un tiempo de recuperación promedio de $\Theta(n)$. Cualquier función hash fija es vulnerable a un comportamiento tan terrible en el peor de los casos; el único efectivo Una forma de mejorar la situación es elegir la función hash *al azar* de una manera que sea *independiente* de las claves que realmente se van a almacenar. Este enfoque, llamado universal hash, puede producir un rendimiento comprobablemente bueno en promedio, sin importar qué teclas elija el adversario.

La idea principal detrás del hash universal es seleccionar la función hash al azar de una clase de funciones cuidadosamente diseñadas al comienzo de la ejecución. Como en el caso de clasificación rápida, la aleatorización garantiza que ninguna entrada única evocará siempre el peor de los casos comportamiento. Debido a la aleatorización, el algoritmo puede comportarse de manera diferente en cada ejecución, incluso para la misma entrada, lo que garantiza un buen rendimiento de caso promedio para cualquier entrada. Volviendo al ejemplo de la tabla de símbolos de un compilador, encontramos que el programador La elección de identificadores ahora no puede causar un rendimiento hash consistentemente deficiente. Pobre El rendimiento ocurre solo cuando el compilador elige una función hash aleatoria que hace que conjunto de identificadores para realizar un hash deficiente, pero la probabilidad de que ocurra esta situación es pequeña y es lo mismo para cualquier conjunto de identificadores del mismo tamaño.

Sea $\mathcal{H}$ una colección finita de funciones hash que mapean un universo dado U de claves en el rango $\{0, 1, ..., m - 1\}$. Se dice que tal colección es universal si para cada par de claves distintas $k, l \in U$, el número de funciones hash $h \in \mathcal{H}$ para las cuales $h(k) = h(l)$ es como máximo $|\mathcal{H}| / m$. En otras palabras, con una función hash elegida al azar de $\mathcal{H}$, la posibilidad de una colisión entre teclas distintas k y l no hay más que la posibilidad de $1 / m$ de una colisión si $h(k)$ y $h(l)$ fueron elegidos aleatoria e independientemente del conjunto $\{0, 1, ..., m - 1\}$.

El siguiente teorema muestra que una clase universal de funciones hash da un buen caso promedio comportamiento. Recuerde que n_i denota la longitud de la lista $T[i]$.

Teorema 11.3

Suponga que una función hash h se elige de una colección universal de funciones hash y es utilizado para hash n claves en una tabla T de tamaño m, utilizando encadenamiento para resolver colisiones. Si la clave k no es en la tabla, entonces la longitud esperada $E[n_{h(k)}]$ de la lista a la que se aplica la clave k es como máximo α. Si clave k está en la tabla, entonces la longitud esperada $E[n_{h(k)}]$ de la lista que contiene la clave k es como máximo $1 + \alpha$.

Prueba Observamos que las expectativas aquí están sobre la elección de la función hash, y no dependen de las suposiciones sobre la distribución de las claves. Para cada par de k y l de distinta claves, defina la variable aleatoria del indicador $X_{kl} = I\{h(k) = h(l)\}$. Dado que, por definición, un solo par de claves choca con probabilidad como máximo $1 / m$, tenemos $\Pr\{h(k) = h(l)\} \le 1 / m$, y así El lema 5.1 implica que $E[X_{kl}] \le 1 / m$.

A continuación definimos, para cada clave k, la variable aleatoria Y_k que es igual al número de claves de otras que k ese hash en la misma ranura que k, de modo que

Página 203

Así tenemos

El resto de la prueba depende de si la tecla k está en la tabla T .

- Si $k \notin T$, entonces $n_{h(k)} = Y_k$ y $| \{ l : l \in T \text{ y } l \neq k \} | = n$. Por tanto, $E [n_{h(k)}] = E [Y_k] \leq n / m = \alpha$.
- Si $k \in T$, entonces porque la tecla k aparece en la lista $T [h (k)]$ y el recuento Y_k no incluye clave k , tenemos $n_{h(k)} = Y_k + 1$ y $| \{ l : l \in T \text{ y } l \neq k \} | = n - 1$. Por lo tanto, $E [n_{h(k)}] = E [Y_k] + 1 \leq (n - 1) / m + 1 = 1 + \alpha - 1 / m < 1 + \alpha$.

El siguiente corolario dice que el hash universal proporciona la recompensa deseada: ahora es imposible que un adversario elija una secuencia de operaciones que obligue al peor de los casos tiempo de ejecución. Al aleatorizar inteligentemente la elección de la función hash en tiempo de ejecución, garantizamos que cada secuencia de operaciones puede manejarse con un buen tiempo de ejecución esperado.

Corolario 11.4

Utilizando hash universal y resolución de colisiones encadenando en una tabla con m ranuras, se necesita tiempo esperado $\Theta (n)$ para manejar cualquier secuencia de n operaciones INSERT, SEARCH y DELETE que contiene operaciones $O (m)$ INSERT.

Prueba Dado que el número de inserciones es $O (m)$, tenemos $n = O (m)$ y entonces $\alpha = O (1)$. los Las operaciones INSERT y DELETE toman un tiempo constante y, según el teorema 11.3 , el tiempo esperado para cada operación de BÚSQUEDA es $O (1)$. Por linealidad de la expectativa, por lo tanto, el tiempo esperado para toda la secuencia de operaciones es $O (n)$.

Diseñar una clase universal de funciones hash

Es bastante fácil diseñar una clase universal de funciones hash, ya que un poco de teoría de números ayudará nosotros probamos. Es posible que desee consultar el Capítulo 31 primero si no está familiarizado con la teoría de números.

Página 204

Comenzamos eligiendo un número primo p lo suficientemente grande como para que todas las posibles claves k *estén* en el rango 0 *ap* - 1, inclusive. Sea $\mathbf{Z}_p$ el conjunto $\{0, 1, ..., p - 1\}$, y denote el conjunto $\{1, 2, ..., p - 1\}$. Como p es primo, podemos resolver ecuaciones módulo p con los métodos dados en Capítulo 31 . Porque asumimos que el tamaño del universo de claves es mayor que el número de ranuras en la tabla hash, tenemos $p > m$.

Ahora definimos la función hash $h_{a, b}$ para cualquier y cualquier $b \in \mathbf{Z}_p$ usando un lineal transformación seguida de reducciones módulo p*y* luego módulo m :

(11,3)

Por ejemplo, con $p = 17$ *ym* = 6, tenemos $h_{3,4} (8) = 5$. La familia de todos esos hash funciones es

(11,4)

Cada función hash $h_{a, b}$ mapea $\mathbf{Z}_p$ a $\mathbf{Z}_m$. Esta clase de funciones hash tiene la propiedad de que el tamaño m del rango de salida es arbitrario, no necesariamente primo, una característica que utilizar en la Sección 11.5. Puesto que hay p - 1 opciones para *una* y hay p opciones de b , hay $p (p - 1)$ funciones hash en $\mathcal{H}_{p, m}$.

Teorema 11.5

La clase $\mathcal{H}_{p,m}$ de funciones hash definidas por las ecuaciones (11.3) y (11.4) es universal.

Prueba Considere dos claves distintas k y l de $\mathbf{Z}_p$, de modo que $k \neq l$. Para una función hash dada $h_{a,b}$ dejamos

$r = (ak + b) \bmod p$,
$s = (al + b) \bmod p$.

Primero notamos que $r \neq s$. ¿Por qué? Observa eso

$r - s \equiv a (k - l) \pmod p$).

Se deduce que $r \neq s$ porque p es primo y tanto a como $(k - l)$ son módulo p distintos de cero , por lo que su producto también debe ser módulo p distinto de cero según el Teorema 31.6 . Por lo tanto, durante el cómputo de cualquier $h_{a,b}$ en $\mathcal{H}_{p,m}$, entradas distintas k y l mapa para distintos valores r y s modulo p ; todavía no hay colisiones en el " nivel mod p ". Además, cada uno de los posibles $p (p - 1)$ opciones para el par (a , b) con $a \neq 0$ produce un par resultante *diferente* (r , s) con $r \neq s$, ya que puede resolver para *una* y b dado r y s :

$a = ((r - s) ((k - l)^{-1} \bmod p)) \bmod p$,
$b = (r - ak) \bmod p$,

donde $((k - l)^{-1} \bmod p)$ denota el único inverso multiplicativo, módulo p , de $k - l$. Ya que solo hay $p (p - 1)$ pares posibles (r , s) con $r \neq s$, hay una correspondencia uno a uno

Página 205

entre pares (a , b) con $a \neq = 0$ y pares (r , s) con $r \neq s$. Por lo tanto, para cualquier par de entradas dado k y l , si tomamos (un , b) uniformemente al azar de , el par resultante (r , s) es igualmente probablemente sea cualquier par de valores distintos módulo p .

Luego se deduce que la probabilidad de que las distintas claves k y l colisionen es igual a la probabilidad que $r \equiv s \pmod m$) cuando r y s se eligen al azar como valores distintos modulo p . Para una dada valor de r , de los $p - 1$ posibles valores restantes para s , el número de valores s tales que $s \neq r$ y $s \equiv r \pmod m$) es como máximo

$\lceil p / m \rceil - 1 \leq ((p + m - 1) / m) - 1$ (por desigualdad (3.7))
$= (p - 1) / m$.

La probabilidad de que s choque con r cuando el módulo reducido m es como máximo $((p - 1) / m) / (p - 1) = 1 / m$.

Por lo tanto, para cualquier par de valores distintos $k , l \mathbf{Z}_p$,

$\Pr \{ h_{a,b} (k) = h_{a,b} (l)\} \leq 1 / m$,

de modo que $\mathcal{H}_{p,m}$ es de hecho universal.

Ejercicios 11.3-1

Supongamos que deseamos buscar una lista enlazada de longitud n , donde cada elemento contiene una clave k junto con un valor hash $h (k)$. Cada tecla es una cadena de caracteres larga. ¿Cómo podríamos tomar ventaja de los valores hash al buscar en la lista un elemento con una clave determinada?

Ejercicios 11.3-2

Suponga que una cadena de r caracteres se codifica en m ranuras tratándola como un número radix-128

y luego usando el método de división. El número m se representa fácilmente como una computadora de 32 bits palabra, pero la cadena de r caracteres, tratada como un número radix-128, requiere muchas palabras. Cómo ¿Podemos aplicar el método de división para calcular el valor hash de la cadena de caracteres sin utilizando más de un número constante de palabras de almacenamiento fuera de la propia cadena?

Ejercicios 11.3-3

Considere una versión del método de división en la que $h(k) = k \bmod m$, donde $m = 2^p - 1$ y k es una cadena de caracteres interpretada en base 2^p. Demuestre que si la cadena x se puede derivar de la cadena y por

Página 206

permutando sus caracteres, luego x e y hash al mismo valor. Da un ejemplo de aplicación en la que esta propiedad no sería deseable en una función hash.

Ejercicios 11.3-4

Considere una tabla hash de tamaño $m = 1000$ y una función hash correspondiente $h(k) = \lfloor m(kA \bmod 1) \rfloor$ para . Calcule las ubicaciones a las que se encuentran las claves 61, 62, 63, 64 y 65. mapeado.

Ejercicios 11.3-5: ⋆

Defina una familia $\mathcal{H}$ de funciones hash desde un conjunto finito U a un conjunto finito B para que sea -universal si para todos los pares de elementos distintos k y l en U,

$\Pr\{h(k) = h(l)\} \leq$,

donde la probabilidad se toma sobre el dibujo de la función hash h al azar de la familia $\mathcal{H}$. Demuestre que una familia universal de funciones hash debe tener

Ejercicios 11.3-6: ⋆

Sea U el conjunto de n-tuplas de valores extraídos de $\mathbf{Z}_p$, y sea $B = \mathbf{Z}_p$, donde p es primo. Definir la función hash $h_b : U \to B$ para $b \mathbf{Z}_p$ en una entrada n-tupla $a_0, a_1, ..., a_{n-1}$ de U como

y sea $\mathcal{H} = \{h_b : b \mathbf{Z}_p\}$. Argumenta que $\mathcal{H}$ es $((n - 1) / p)$ -universal según la definición de -universal en el ejercicio 11.3-5 . (*Sugerencia:* vea el ejercicio 31.4-4 .)

11.4 Direccionamiento abierto

En el direccionamiento abierto , todos los elementos se almacenan en la propia tabla hash. Es decir, cada entrada de la tabla contiene un elemento del conjunto dinámico o NIL. Al buscar un elemento,
Examine sistemáticamente las ranuras de la mesa hasta que se encuentre el elemento deseado o hasta que esté claro que el elemento no está en la tabla. No hay listas ni elementos almacenados fuera de la tabla, ya que

Página 207

están en encadenamiento. Por lo tanto, en el direccionamiento abierto, la tabla hash se puede "llenar" para que no haya más se pueden hacer inserciones; el factor de carga α nunca puede exceder 1.

Por supuesto, podríamos almacenar las listas vinculadas para encadenarlas dentro de la tabla hash, de lo contrario ranuras de tabla hash no utilizadas (vea el ejercicio 11.2-4), pero la ventaja del direccionamiento abierto es que evita los punteros por completo. En lugar de seguir punteros, *calculamos* la secuencia de ranuras para ser examinado. La memoria extra liberada al no almacenar punteros proporciona a la tabla hash una mayor número de ranuras para la misma cantidad de memoria, lo que potencialmente produce menos colisiones y recuperación más rápida.

Para realizar la inserción mediante el direccionamiento abierto, examinamos o sondeamos sucesivamente el hash mesa hasta encontrar una ranura vacía en la que poner la llave. En lugar de fijarse en el orden 0, 1, ..., m - 1 (que requiere $\Theta(n)$ tiempo de búsqueda), la secuencia de posiciones probadas *depende de la llave que se inserta* . Para determinar qué ranuras sondear, ampliamos la función hash a incluya el número de sonda (comenzando desde 0) como una segunda entrada. Por tanto, la función hash se convierte en

$h : U \times \{0, 1, ..., m - 1\} \rightarrow \{0, 1, ..., m - 1\}$.

Con direccionamiento abierto, requerimos que para cada tecla k , la secuencia de la sonda

$h(k, 0), h(k, 1), ..., h(k, m - 1)$

ser una permutación de 0, 1, ..., m -1, de modo que cada posición de la tabla hash sea eventualmente considerado como un espacio para una nueva llave a medida que la mesa se llena. En el siguiente pseudocódigo, suponga que los elementos de la tabla hash T son claves sin información de satélite; la clave k es idéntico al elemento que contiene la clave k . Cada ranura contiene una llave o NIL (si la ranura es vacío).

```
HASH-INSERT ( T , k )
1 i ← 0
2 repetir j ← h ( k , i )
3                   si T [ j ] = NULO
4                         luego T [ j ] ← k
5                                 volver j
6                         más yo ← yo + 1
7 hasta que yo = m
8 error "desbordamiento de tabla hash"
```

El algoritmo para buscar la clave k sondea la misma secuencia de ranuras que la inserción algoritmo examinado cuando se insertó la clave k . Por lo tanto, la búsqueda puede terminar (sin éxito) cuando encuentra una ranura vacía, ya que k se habría insertado allí y no más adelante en su secuencia de sondeo. (Este argumento asume que las claves no se eliminan del hash tabla.) El procedimiento HASH-SEARCH toma como entrada una tabla hash T y una tecla k , devolviendo j si ranura j se encontró que contenía clave k , o NIL si la clave k no está presente en la tabla T .

```
BÚSQUEDA DE HASH ( T , k )
1 i ← 0
2 repetir j ← h ( k , i )
3                   si T [ j ] = k
4                         luego regresa j
5                   yo ← yo + 1
6 hasta que T [ j ] = NULO o i = m
```

Página 208

```
7 devuelve NIL
```

La eliminación de una tabla hash de direcciones abiertas es difícil. Cuando eliminamos una clave de la ranura i , no puede simplemente marcar esa ranura como vacía almacenando NIL en ella. Hacerlo podría hacerlo imposible para recuperar cualquier llave k durante cuya inserción hayamos sondeado la ranura i y la encontramos ocupada. Uno La solución es marcar la ranura almacenando en ella el valor especial DELETED en lugar de NIL. Nosotros

luego modificaría el procedimiento HASH-INSERT para tratar dicha ranura como si estuviera vacía, por lo que que se puede insertar una nueva llave. No se necesita ninguna modificación de HASH-SEARCH, ya que pasar los valores BORRADOS durante la búsqueda. Cuando usamos el valor especial ELIMINADO, sin embargo, los tiempos de búsqueda ya no dependen del factor de carga α, y por esta razón El encadenamiento se selecciona más comúnmente como una técnica de resolución de colisiones cuando las teclas deben ser eliminado.

En nuestro análisis, asumimos el hash uniforme : asumimos que cada clave es igualmente probable de tener alguno de los m ! permutaciones de 0, 1, ..., m - 1 como su secuencia de sonda. El hash uniforme generaliza la noción de hash uniforme simple definido antes de la Situación en la que la función hash produce no solo un número, sino una sonda completa. secuencia. Sin embargo, el hash uniforme verdadero es difícil de implementar y en la práctica es adecuado se utilizan aproximaciones (como el doble hash, que se define a continuación).

Normalmente se utilizan tres técnicas para calcular las secuencias de sonda necesarias para la apertura direccionamiento: palpado lineal, palpado cuadrático y doble hash. Todas estas técnicas garantizar que $h(k, 0), h(k, 1), ..., h(k, m - 1)$ es una permutación de 0, 1, ..., m - 1 para cada clave k . Sin embargo, ninguna de estas técnicas cumple con el supuesto de hash uniforme, ya que ninguno de ellos es capaz de generar más de m^2 secuencias de sonda diferentes (en lugar del m ! que requiere el hash uniforme). El hash doble tiene la mayor cantidad de sonda secuencias y, como era de esperar, parece dar los mejores resultados.

Palpado lineal

Dada una función hash ordinaria $h' : U \rightarrow \{0, 1, ..., m - 1\}$, a la que nos referimos como auxiliar función hash , el método de sondeo lineal utiliza la función hash

$$h(k, i) = (h'(k) + i) \bmod m$$

para $i = 0, 1, ..., m - 1$. Dada la clave k , la primera ranura probada es $T[h'(k)]$, es decir, la ranura dada por el función hash auxiliar. A continuación, sondeamos la ranura $T[h'(k) + 1]$, y así sucesivamente hasta la ranura $T[m - 1]$. Luego pasamos por las ranuras $T[0]$, $T[1]$, ..., hasta que finalmente sondeamos la ranura $T[h'(k) - 1]$. Porque el La sonda inicial determina la secuencia completa de la sonda, solo hay m secuencias de sonda distintas.

El palpado lineal es fácil de implementar, pero adolece de un problema conocido como primario. agrupamiento . Se acumulan largas tiradas de espacios ocupados, lo que aumenta el tiempo medio de búsqueda. Clusters surgen porque un espacio vacío precedido por i espacios completos se llena a continuación con probabilidad $(i + 1) / m$. Las tiradas largas de espacios ocupados tienden a alargarse y el tiempo medio de búsqueda aumenta.

Sondeo cuadrático

El sondeo cuadrático utiliza una función hash de la forma

(11,5)

Página 209

donde h' es una función hash auxiliar, c_1 y $c_2 \neq 0$ son constantes auxiliares, e $i = 0, 1, ..., m - 1$. La posición inicial probada es $T[h'(k)]$; posiciones posteriores investigadas se compensan con cantidades que dependen de forma cuadrática del número de sonda i . Este método funciona mucho mejor que sondeo lineal, sino para hacer pleno uso de la tabla hash, los valores de c_1 , c_2 , y m son constreñido. El problema 11-3 muestra una forma de seleccionar estos parámetros. Además, si dos llaves tienen la misma posición inicial de la sonda, entonces sus secuencias de sonda son las mismas, ya que $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$. Esta propiedad conduce a una forma más leve de agrupación, llamada agrupación secundaria . Como en el palpado lineal, el palpador inicial determina toda la secuencia, de modo que sólo se utilizan m secuencias de sonda distintas.

Hash doble

El doble hash es uno de los mejores métodos disponibles para el direccionamiento abierto porque las permutaciones producidas tienen muchas de las características de las permutaciones elegidas al azar. El hash doble usa una función hash de la forma

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m ,$$

donde h_1 y h_2 son funciones hash auxiliares. La sonda inicial es la posición $T[h_1(k)]$; las posiciones sucesivas de la sonda se compensan de las posiciones anteriores por la cantidad $h_2(k)$, módulo m . Por tanto, a diferencia del caso del sondeo lineal o cuadrático, la secuencia de la sonda aquí depende de dos

formas en la tecla k , ya que la posición inicial de la sonda, el desplazamiento, o ambos, pueden variar. Figura 11.5 da un ejemplo de inserción mediante doble hash.

Figura 11.5: Inserción mediante doble hash. Aquí tenemos una tabla hash de tamaño 13 con $h_1(k) = k \bmod 13$ y $h_2(k) = 1 + (k \bmod 11)$. Dado que $14 \equiv 1 \pmod{13}$ y $14 \equiv 3 \pmod{11}$, la clave 14 es insertado en la ranura vacía 9, después de que las ranuras 1 y 5 se examinen y se descubra que están ocupadas.

El valor $h_2(k)$ debe ser relativamente primo para el tamaño de la tabla hash m para que toda la tabla hash sea buscado. (Ver ejercicio 11,4-3 .) Una forma conveniente para asegurar esta condición es dejar que m sea una potencia de 2 y diseñar h_2 para que siempre produzca un número impar. Otra forma es dejar m

Página 210

ser primo y diseñar h_2 de modo que siempre devuelva un entero positivo menor que m . Por ejemplo, podríamos elegir m prime y dejar

$h_1(k) = k \bmod m$,
$h_2(k) = 1 + (k \bmod m')$,

donde m' se elige para que sea ligeramente menor que m (digamos, $m - 1$). Por ejemplo, si $k = 123456$, $m = 701$, y $m' = 700$, tenemos $h_1(k) = 80$ y $h_2(k) = 257$, por lo que la primera sonda está en la posición 80, y luego se examina cada 257a ranura (módulo m) hasta que se encuentra la llave o se examinan todas las ranuras.

El hash doble mejora con respecto al sondeo lineal o cuadrático en el sentido de que las secuencias de sonda $\Theta(m^2)$ son utilizado, en lugar de $\Theta(m)$, ya que cada posible par $(h_1(k), h_2(k))$ produce una secuencia de sonda distinta. Como resultado, el rendimiento del doble hash parece estar muy cerca del rendimiento de el esquema "ideal" de hash uniforme.

Análisis de hash de direcciones abiertas

Nuestro análisis del direccionamiento abierto, como nuestro análisis del encadenamiento, se expresa en términos de factor de carga $\alpha = n / m$ de la tabla hash, como n y m Ir al infinito. Por supuesto, con abierto direccionamiento, tenemos como máximo un elemento por ranura, y por lo tanto $n \le m$, lo que implica $\alpha \le 1$.

Suponemos que se utiliza hash uniforme. En este esquema idealizado, la secuencia de la sonda $h(k, 0), h(k, 1), \ldots, h(k, m - 1)$ usado para insertar o buscar cada clave k es igualmente probable que sea cualquier permutación de $0, 1, \ldots, m - 1$. Por supuesto, una clave determinada tiene una secuencia de sonda fija única asociado a ello; lo que se quiere decir aquí es que, considerando la distribución de probabilidad en el espacio de claves y el funcionamiento de la función hash en las claves, cada posible sonda La secuencia es igualmente probable.

Ahora analizamos el número esperado de sondas para hash con direccionamiento abierto bajo el suposición de hash uniforme, comenzando con un análisis del número de sondas realizadas en una búsqueda fallida.

Teorema 11.6

Dada una tabla hash de dirección abierta con factor de carga $\alpha = n / m < 1$, el número esperado de sondas en una búsqueda sin éxito es como máximo $1 / (1-\alpha)$, asumiendo un hash uniforme.

Prueba En una búsqueda infructuosa, todas las sondas excepto la última acceden a un espacio ocupado que no no contiene la clave deseada y la última ranura probada está vacía. Definamos el azar variable X para ser el número de sondas realizadas en una búsqueda fallida, y definamos también el evento A_i, para $i = 1, 2, ...$, para ser el evento de que hay una i- ésima sonda y es un ocupado espacio. Entonces el evento $\{ X \geq i \}$ es la intersección de los eventos $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. Vamos a atar $\Pr \{ X \geq i \}$ limitando $\Pr \{ A_1 \cap A_2 \cap \cdots \cap A_{i-1} \}$. Mediante el ejercicio C.2-6 ,

$$\Pr \{ A_1 \cap A_2 \cap \cdots \cap A_{i-1} \} = \Pr \{ A_1 \} \cdot \Pr \{ A_2 \mid A_1 \} \cdot \Pr \{ A_3 \mid A_1 \cap A_2 \} \Pr \{ A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2} \}.$$

Página 211

Como hay n elementos y m ranuras, $\Pr \{ A_1 \} = n / m$. Para $j > 1$, la probabilidad de que haya un j- ésima sonda y es a una ranura ocupada, dado que las primeras j - 1 sondas fueron a ranuras ocupadas, es $(n - j + 1) / (m - j + 1)$. Esta probabilidad sigue porque estaríamos encontrando uno de los $(n - (j - 1))$ elementos restantes en uno de los $(m - (j - 1))$ espacios no examinados, y por Supuesto de hash uniforme, la probabilidad es la razón de estas cantidades. Observando eso $n < m$ implica que $(n - j) / (m - j) \leq n / m$ para todo j tal que $0 \leq j < m$, tenemos para todo i tal que $1 \leq yo \leq m$,

Ahora usamos la ecuación (C.24) para acotar el número esperado de sondas:

El límite anterior de $1 + \alpha + \alpha_2 + \alpha_3 + \cdots$ tiene una interpretación intuitiva. Siempre se hace una sonda. Con probabilidad de aproximadamente α, la primera sonda encuentra una ranura ocupada de modo que una segunda sonda es necesario. Con una probabilidad aproximada de α_2 , las dos primeras ranuras están ocupadas de modo que una tercera sonda es necesaria, y así sucesivamente.

Si α es una constante, el teorema 11.6 predice que una búsqueda infructuosa se ejecuta en el tiempo O (1). por Por ejemplo, si la tabla hash está medio llena, el número promedio de sondas en una búsqueda sin éxito es como máximo 1 / (1 - .5) = 2. Si está lleno al 90 por ciento, el número promedio de sondas es como máximo 1 / (1 - .9) = 10.

El teorema 11.6 nos da el rendimiento del procedimiento HASH-INSERT casi inmediatamente.

Corolario 11.7

Insertar un elemento en una tabla hash de dirección abierta con factor de carga α requiere como máximo 1 / (1 - α) sondas en promedio, asumiendo un hash uniforme.

Prueba Un elemento se inserta solo si hay espacio en la mesa y, por lo tanto, α <1. Inserción de una llave requiere una búsqueda fallida seguida de la colocación de la llave en la primera ranura vacía encontró. Por tanto, el número esperado de sondas es como máximo 1 / (1 - α).

Página 212

Calcular el número esperado de sondas para una búsqueda exitosa requiere un poco más de trabajo.

Teorema 11.8

Dada una tabla hash de dirección abierta con factor de carga $\alpha < 1$, el número esperado de sondas en un la búsqueda exitosa es como máximo

asumiendo hash uniforme y asumiendo que cada clave en la tabla es igualmente probable que sea buscado para.

Prueba La búsqueda de una clave k sigue la misma secuencia de sondas que se siguió cuando Se insertó un elemento con la clave k . Por el Corolario 11.7 , si k fuera la $(i + 1)$ a clave insertada en el tabla hash, el número esperado de sondas realizadas en una búsqueda de k es como máximo $1 / (1 - i / m) = m / (m - i)$. El promedio de todas las n claves en la tabla hash nos da el número promedio de sondas en un búsqueda exitosa:

dónde es el i- ésimo número armónico (como se define en la ecuación (A.7)). Utilizando la técnica de delimitar una suma por una integral, como se describe en la Sección A.2 , obtenemos

para un límite en el número esperado de sondas en una búsqueda exitosa.

Si la tabla hash está medio llena, el número esperado de sondas en una búsqueda exitosa es menor que 1.387. Si la tabla hash está llena en un 90 por ciento, el número esperado de sondas es menor a 2.559.

Ejercicios 11.4-1

Página 213

Considere insertar las claves 10, 22, 31, 4, 15, 28, 17, 88, 59 en una tabla hash de longitud $m = 11$ utilizando direccionamiento abierto con la función hash primaria $h'(k) = k \bmod m$. Ilustre el resultado de

insertando estas llaves usando palpación lineal, usando palpación cuadrática con $c_1 = 1$ y $c_2 = 3$, y usando doble hash con $h_2(k) = 1 + (k \bmod (m - 1))$.

Ejercicios 11.4-2

Escriba el pseudocódigo para HASH-DELETE como se describe en el texto y modifique HASH-INSERT para manejar el valor especial ELIMINADO.

Ejercicios 11.4-3: ⋆

Suponga que usamos doble hash para resolver colisiones; es decir, usamos la función hash $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Demostrar que si m y $h_2(k)$ tienen el máximo común divisor $d \geq 1$ para alguna clave k, luego una búsqueda fallida de la clave k examina $(1/d)$ th de la tabla hash antes de volver a la ranura $h_1(k)$. Así, cuando $d = 1$, de modo que m y $h_2(k)$ son relativamente primos, la la búsqueda puede examinar toda la tabla hash. (*Sugerencia:* consulte el capítulo 31).

Ejercicios 11.4-4

Considere una tabla hash de dirección abierta con hash uniforme. Dar límites superiores en el número esperado de sondas en una búsqueda sin éxito y en el número esperado de sondas en una búsqueda exitosa cuando el factor de carga es 3/4 y cuando es 7/8.

Ejercicios 11.4-5: ⋆

Considere una tabla hash de dirección abierta con un factor de carga α. Encuentre el valor α distinto de cero para el cual el número esperado de sondas en una búsqueda sin éxito es igual al doble del número esperado de sondas en una búsqueda exitosa. Utilice los límites superiores dados por los teoremas 11.6 y 11.8 para estos números esperados de sondas.

11.5 Hash perfecto

Aunque el hash se usa con mayor frecuencia por su excelente rendimiento esperado, el hash puede ser utilizado para obtener un excelente rendimiento en el *peor de los casos* cuando el conjunto de claves es estático: una vez que las claves

Página 214

se almacenan en la tabla, el conjunto de claves nunca cambia. Algunas aplicaciones tienen naturalmente conjuntos de claves: considere el conjunto de palabras reservadas en un lenguaje de programación, o el conjunto de archivos nombres en un CD-ROM. Llamamos hash perfecto a una técnica de hash si el número del peor de los casos de los accesos a la memoria necesarios para realizar una búsqueda es $O(1)$.

La idea básica para crear un esquema de hash perfecto es simple. Usamos un hash de dos niveles esquema con hash universal en cada nivel. La figura 11.6 ilustra el enfoque.

Figura 11.6: Uso de hash perfecto para almacenar el conjunto $K = \{10, 22, 37, 40, 60, 70, 75\}$. El exterior la función hash es $h(k) = ((ak + b) \bmod p) \bmod m$, donde $a = 3$, $b = 42$, $p = 101$ y $m = 9$. Para

ejemplo, $h(75) = 2$, por lo clave 75 hashes a la ranura 2 de la tabla T. Una tabla hash secundaria S_j almacena todo hash de las teclas en la ranura j. El tamaño de la tabla hash S_j es m_j, y la función hash asociada es $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Dado que $h_2(75) = 1$, la clave 75 se almacena en la ranura 1 del hash secundario tabla S_2. No hay colisiones en ninguna de las tablas hash secundarias, por lo que la búsqueda requiere tiempo constante en el peor de los casos.

El primer nivel es esencialmente el mismo que para el hash con encadenamiento: las n claves se hash en m ranuras usando una función hash h cuidadosamente seleccionada de una familia de funciones hash universales.

Sin embargo, en lugar de hacer una lista de las claves hash en la ranura j, usamos un pequeño hash secundario tabla S_j con una función hash asociada h_j. Al elegircuidadosamente lasfunciones hash h_j, puede garantizar que no haya colisiones a nivel secundario.

Sin embargo, para garantizar que no haya colisiones en el nivel secundario, necesitaremos para permitir que el tamaño m_j de la tabla hash S_j sea el cuadrado del número n_j de claves hash en la ranura j. Mientras tener tal dependencia cuadrática de m_j en n_j puede parecer probable que cause el almacenamiento general los requisitos sean excesivos, mostraremos que eligiendo la función hash de primer nivel bueno, la cantidad total esperada de espacio utilizado sigue siendo $O(n)$.

Usamos funciones hash elegidas de las clases universales de funciones hash de la Sección 11.3.3.

La función hash de primer nivel se elige de la clase $\mathcal{H}_{p,m}$, donde, como en la Sección 11.3.3, p es un número primo mayor que cualquier valor clave. Esas claves hash en la ranura j se vuelven a convertir en un tabla hash secundaria S_j de tamaño m_j usando una función hash h_j elegida de la clase . [1]

Procederemos en dos pasos. Primero, determinaremos cómo asegurarnos de que el secundario las tablas no tienen colisiones. En segundo lugar, mostraremos que la cantidad esperada de memoria utilizada en general, para la tabla hash primaria y todas las tablas hash secundarias, es $O(n)$.

Teorema 11.9

Página 215

Si almacenamos n claves en una tabla hash de tamaño $m = n^2$ usando una función hash h elegida al azar una clase universal de funciones hash, entonces la probabilidad de que haya colisiones es menor de 1/2.

Prueba Hay pares de llaves que pueden colisionar; cada par choca con probabilidad $1/m$ si h es elegido al azar de una familia universal $\mathcal{H}$ de funciones hash. Sea X una variable aleatoria que cuenta el número de colisiones. Cuando $m = n^2$, el número esperado de colisiones es

(Tenga en cuenta que este análisis es similar al análisis de la paradoja del cumpleaños en la Sección 5.4.1.) Aplicando la desigualdad de Markov (C.29), $\Pr\{X \geq t\} \leq E[X]/t$, con $t = 1$ completa la demostración.

En la situación descrita en el teorema 11.9, donde $m = n^2$, se deduce que una función hash h elegidos al azar de $\mathcal{H}$ es más probable que no tenga *ningún* colisiones. Dado el conjunto K de n claves para ser hash (recuerde que K es estático), por lo tanto, es fácil encontrar un hash sin colisiones función h con algunos ensayos aleatorios.

Sin embargo, cuando n es grande, una tabla hash de tamaño $m = n^2$ es excesiva. Por lo tanto, adoptamos la enfoque hash de dos niveles, y usamos el enfoque del teorema 11.9 solo para hash el entradas dentro de cada ranura. Un exterior, o de primer nivel, función hash h se utiliza para hash de las claves en la $m = n$ ranuras. Entonces, si n_j claves hash en la ranura j, una tabla hash secundaria S_j de tamaño $m_j = n_j^2$ se usa para proporciona una búsqueda de tiempo constante sin colisiones.

Pasamos ahora a la cuestión de garantizar que la memoria total utilizada sea $O(n)$. Dado que el tamaño m_j de la j-ésima tabla hash secundaria crece cuadráticamente con el número n_j de claves almacenadas, hay

el riesgo de que la cantidad total de almacenamiento sea excesiva.

Si el tamaño de la tabla de primer nivel es $m = n$, entonces la cantidad de memoria utilizada es $O(n)$ para la tabla hash, para el almacenamiento de los tamaños m_j de las tablas hash secundarias, y para el almacenamiento de los parámetros a_j y b_j que definen las funciones hash secundarias h_j extraídas de la clase de Sección 11.3.3 (excepto cuando $n_j = 1$ y usamos $a = b = 0$). El siguiente teorema y un El corolario proporciona un límite en los tamaños combinados esperados de todas las tablas hash secundarias. UNA El segundo corolario limita la probabilidad de que el tamaño combinado de todo el hash secundario tablas es superlineal.

Teorema 11.10

Si almacenamos n claves en una tabla hash de tamaño $m = n$ usando una función hash h elegida al azar una clase universal de funciones hash, entonces

Página 216

donde n_j es el número de claves hash en la ranura j .

Prueba Comenzamos con la siguiente identidad, que es válida para cualquier número entero no negativo a :

(11,6)

Tenemos

Para evaluar la suma , observamos que es solo el número total de colisiones. Según las propiedades del hash universal, el valor esperado de esta suma es como máximo

ya que $m = n$. Así,

Corolario 11.11

Si almacenamos n claves en una tabla hash de tamaño $m = n$ usando una función hash h elegida al azar una clase universal de funciones hash y establecemos el tamaño de cada tabla hash secundaria en para $j = 0, 1, ..., m - 1$, entonces la cantidad esperada de almacenamiento requerida para todo el hash secundario tablas en un esquema de hash perfecto es menor que $2n$.

Prueba desde para $j = 0, 1, ..., m - 1$, el teorema 11.10 da

Página 217

(11,7)

que completa la prueba.

Corolario 11.12

Si almacenamos n claves en una tabla hash de tamaño $m = n$ usando una función hash h elegida al azar una clase universal de funciones hash y establecemos el tamaño de cada tabla hash secundaria en para $j = 0, 1, ..., m - 1$, entonces la probabilidad de que el almacenamiento total utilizado para las tablas hash secundarias excede $4n$ es menor que 1/2.

Prueba Nuevamente aplicamos la desigualdad de Markov (C.29) , $\Pr \{ X \geq t \} \leq E [X] / t$, esta vez a la desigualdad (11,7) , con y $t = 4n$:

En el Corolario 11.12 , vemos que probar algunas funciones hash elegidas al azar de la La familia universal producirá rápidamente una que utilice una cantidad razonable de almacenamiento.

Ejercicios 11.5-1: ⋆

Suponga que insertamos n claves en una tabla hash de tamaño m usando direccionamiento abierto y uniforme hash. Sea $p (n , m)$ la probabilidad de que no ocurran colisiones. Muestre que $p (n , m) \leq e^{-n(n-1)/2m}$. (*Sugerencia:* vea la ecuación (3.11) .) Argumente que cuando n excede, la probabilidad de evitar colisiones pasa rápidamente a cero.

Problemas 11-1: enlace de la sonda más larga para hash

Se utiliza una tabla hash de tamaño m para almacenar n elementos, con $n \leq m / 2$. El direccionamiento abierto se utiliza para resolución de colisiones.

a. Suponiendo hash uniforme, demuestre que para $i = 1, 2, ..., n$, la probabilidad de que el i- ésimo la inserción requiere estrictamente más de k sondas es como máximo 2^{-k} .
si. Demuestre que para $i = 1, 2, ..., n$, la probabilidad de que la i- ésima inserción requiera más de $2 \lg n$ sondas es como máximo $1 / n^2$.

Página 218

Deje que la variable aleatoria X_i denote el número de sondas requeridas por la i- ésima inserción. Tú han demostrado en la parte (b) que $\Pr \{ X_i > 2 \lg n \} \leq 1 / n^2$. Sea la variable aleatoria $X = \max_{1 \leq i \leq n} X_i$

denota el número máximo de sondas requeridas por cualquiera de las n inserciones.

C. Muestre que Pr $\{ X > 2\lg n \} \leq 1 / n$.
re. Muestre que la longitud esperada E $[X]$ de la secuencia de sonda más larga es $O (\lg n)$.

Problemas 11-2: límite de tamaño de ranura para encadenamiento

Suponga que tenemos una tabla hash con n ranuras, con colisiones resueltas por encadenamiento, y supongamos que se insertan n claves en la tabla. Es igualmente probable que cada clave tenga un hash para cada espacio. Sea M el número máximo de llaves en cualquier ranura después de que se hayan insertado todas las llaves. Su misión es demostrar un $O (\lg n / \lg \lg n)$ límite superior de E $[M]$, el valor esperado de M .

a. Argumenta que la probabilidad $Q_{k de}$ que exactamente k claves se apunten a una ranura en particular viene dada por

si. Sea P_k la probabilidad de que $M = k$, es decir, la probabilidad de que la ranura que contiene el la mayoría de las claves contienen k claves. Demuestre que $P_k \leq nQ_k$.
C. Utilice la aproximación de Stirling, ecuación (3.17) , para demostrar que $Q_k < e_k / k_k$.
re. Demuestre que existe una constante $c > 1$ tal que para $k_0 = c \lg n / \lg \lg n$.
Concluya que $P_k < 1 / n^2$ para $k \geq k_0 = c \lg n / \lg \lg n$.
mi. Argumenta eso

Concluya que E $[M] = O (\lg n / \lg \lg n)$.

Problemas 11-3: Sondeo cuadrático

Supongamos que se nos da una clave k para buscar en una tabla hash con posiciones 0, 1, ..., m - 1, y supongamos que tenemos una función hash h mapeando el espacio clave en el conjunto {0, 1, ..., m - 1}. El esquema de búsqueda es el siguiente.

1. Calcule el valor $i \leftarrow h (k)$ y establezca $j \leftarrow 0$.
2. Sonda en la posición i para la llave deseada k . Si lo encuentra, o si esta posición está vacía, terminar la búsqueda.
3. Configure $j \leftarrow (j + 1) \bmod m$ e $i \leftarrow (i + j) \bmod m$, y regrese al paso 2.

Suponga que m es una potencia de 2.

Página 219

a. Demuestre que este esquema es un ejemplo del esquema general de "prueba cuadrática" por exhibiendo las constantes apropiadas c_1 y c_2 para la ecuación (11.5) .
si. Demuestre que este algoritmo examina todas las posiciones de la tabla en el peor de los casos.

Problemas 11-4: autenticación y hash k-universal

Sea $\mathcal{H} = \{ h \}$ una clase de funciones hash en las que cada h asigna el universo U de claves a {0, 1, ..., m - 1}. Decimos que $\mathcal{H}$ es k-universal si, para cada secuencia fija de k claves distintas $x^{(1)}, x^{(2)}, ..., x^{(k)}$ y para cualquier h elegida al azar de $\mathcal{H}$, la secuencia $h (x^{(1)}), h (x^{(2)}), ...,$ $h (x^{(k)})$ es igualmente probable que sea cualquiera de las m_k secuencias de longitud k con elementos extraídos de {0, 1, ..., m - 1}.

a. Demuestre que si $\mathcal{H}$ es 2-universal, entonces es universal.

si. Sea U el conjunto de n -tuplas de valores extraídos de $\mathbf{Z}_p$, y sea $B = \mathbf{Z}_p$, donde p es primo. Para cualquier n- tupla $a = a_0, a_1, ..., a_{n-1}$ de valores de $\mathbf{Z}_p$ y para cualquier b $\mathbf{Z}_p$, defina el función hash $h_{a,b} : U \rightarrow B$ en una entrada n -tupla $x = x_0, x_1, ..., x_{n-1}$ de U como

y sea $\mathcal{H} = \{ h_{a,b} \}$. Argumenta que $\mathcal{H}$ es 2-universal.

C. Suponga que Alice y Bob acuerdan secretamente una función hash $h_{a,b}$ de un 2-universal familia $\mathcal{H}$ de funciones hash. Más tarde, Alice envía un mensaje m a Bob a través de Internet, donde m U . Ella autentica este mensaje a Bob enviando también un etiqueta de autenticación $t = h_{a,b}(m)$, y Bob comprueba que el par (m, t) que recibe satisface $t = h_{a,b}(m)$. Suponga que un adversario intercepta (m, t) en el camino y trata de engañar a Bob reemplazando el par con un par diferente (m', t'). Argumenta que la probabilidad de que el El adversario logra engañar a Bob para que acepte (m', t') es como máximo $1 / p$, sin importar cuán mucho poder de computación tiene el adversario.

[1] Cuando $n_j = m_j = 1$, realmente no necesitamos una función hash para la ranura j ; cuando elegimos un hash función $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$ para tal ranura, solo usamos $a = b = 0$.

Notas del capítulo

Knuth [185] y Gonnet [126] son excelentes referencias para el análisis de algoritmos hash. Knuth le da crédito a HP Luhn (1953) por inventar tablas hash, junto con el método de encadenamiento para resolución de colisiones. Aproximadamente al mismo tiempo, GM Amdahl originó la idea de abrir direccionamiento.

Página 220

Carter y Wegman introdujeron la noción de clases universales de funciones hash en 1979 [52] .

Fredman, Komlós y Szemerédi [96] desarrollaron el esquema de hash perfecto para conjuntos estáticos presentado en la Sección 11.5. Una extensión de su método a conjuntos dinámicos, manejando inserciones y eliminaciones en el tiempo esperado amortizado $O(1)$, ha sido proporcionado por Dietzfelbinger et al. [73] .

Capítulo 12: Árboles de búsqueda binaria

Visión general

Los árboles de búsqueda son estructuras de datos que admiten muchas operaciones de conjuntos dinámicos, incluidas BÚSQUEDA, MÍNIMO, MÁXIMO, PREDECESOR, SUCESOR, INSERTAR y ELIMINAR. Por tanto, un árbol de búsqueda se puede utilizar como diccionario y como cola de prioridad.

Las operaciones básicas en un árbol de búsqueda binaria toman un tiempo proporcional a la altura del árbol. Para árbol binario completo con n nodos, tales operaciones se ejecutan en $\Theta(\lg n)$ en el peor de los casos. Si el arbol es una cadena lineal de n nodos, sin embargo, las mismas operaciones toman $\Theta(n)$ tiempo en el peor de los casos. Deberíamos vea en la Sección 12.4 que la altura esperada de un árbol de búsqueda binario construido al azar es $O(\lg n)$, de modo que las operaciones básicas de conjuntos dinámicos en dicho árbol toman $\Theta(\lg n)$ tiempo en promedio.

En la práctica, no siempre podemos garantizar que los árboles de búsqueda binarios se creen aleatoriamente, pero no son variaciones de árboles de búsqueda binaria cuyo desempeño en el peor de los casos en operaciones básicas puede ser garantizado para ser bueno. El capítulo 13 presenta una de esas variaciones, árboles rojo-negro, que tienen altura $O(\lg n)$. El Capítulo 18 presenta los árboles B, que son particularmente buenos para mantener bases de datos en almacenamiento secundario (en disco) de acceso aleatorio.

Después de presentar las propiedades básicas de los árboles de búsqueda binarios, las siguientes secciones muestran cómo recorrer un árbol de búsqueda binaria para imprimir sus valores en orden ordenado, cómo buscar un valor en un árbol de búsqueda binaria, cómo encontrar el elemento mínimo o máximo, cómo encontrar el predecesor o sucesor de un elemento, y cómo insertarlo o eliminarlo de una búsqueda binaria

árbol. Las propiedades matemáticas básicas de los árboles aparecen en el Apéndice B .

12.1 ¿Qué es un árbol de búsqueda binario?

Un árbol de búsqueda binaria está organizado, como su nombre indica, en un árbol binario, como se muestra en la Figura 12.1 . Dicho árbol puede representarse mediante una estructura de datos enlazados en la que cada nodo es un objeto. Además de una *clave de* datos de campo y de satélite, cada nodo contiene campos *izquierda* , *derecha* , y *p* que apunte a los nodos correspondientes a su hijo izquierdo, su hijo derecho y su padre, respectivamente. Si falta un hijo o el padre, el campo correspondiente contiene el valor NIL. El nodo raíz es el único nodo en el árbol cuyo campo padre es NIL.

Página 221

Figura 12.1: Árboles de búsqueda binarios. Para cualquier nodo x , las claves en el subárbol izquierdo de x son como máximo *clave* [x], y las claves en el subárbol derecho de x son al menos *clave* [x]. Diferentes árboles de búsqueda binaria puede representar el mismo conjunto de valores. El peor tiempo de ejecución para la mayoría de los árboles de búsqueda operaciones es proporcional a la altura del árbol. *(a)* Un árbol de búsqueda binario en 6 nodos con altura 2. *(b)* Un árbol de búsqueda binaria menos eficiente con altura 4 que contiene las mismas claves.

Las claves en un árbol de búsqueda binaria siempre se almacenan de tal manera que satisfagan los valores binarios. propiedad del árbol de búsqueda :

- Sea x un nodo en un árbol de búsqueda binario. Si y es un nodo en el subárbol izquierdo de x , entonces *clave* [y] ≤ *tecla* [x]. Si y es un nodo en el subárbol derecho de x , entonces *clave* [x] ≤ *clave* [y].

Por lo tanto, en la figura 12.1 (a) , la clave de la raíz es 5, las claves 2, 3 y 5 en su subárbol izquierdo no son mayor que 5, y las claves 7 y 8 en su subárbol derecho no son menores que 5. El mismo La propiedad se mantiene para cada nodo del árbol. Por ejemplo, la clave 3 en la Figura 12.1 (a) es no más pequeño que la clave 2 en su subárbol izquierdo y no más grande que la clave 5 en su subárbol derecho.

La propiedad binary-search-tree nos permite imprimir todas las claves en un árbol de búsqueda binaria en orden ordenado por un algoritmo recursivo simple, llamado un paseo de árbol en orden . Este algoritmo es tan nombrado porque la clave de la raíz de un subárbol se imprime entre los valores en su subárbol izquierdo y los de su subárbol derecho. (De manera similar, un árbol de preordenar imprime la raíz antes de los valores en cualquiera de los subárboles, y un árbol de postorder imprime la raíz después de los valores en sus subárboles). use el siguiente procedimiento para imprimir todos los elementos en un árbol de búsqueda binario T , llamamos INORDER-TREE-WALK *(raíz* [T]).

```
PASEO-ÁRBOL-INTERIOR ( x )
1 si x ≠ NULO
2 luego INORDER-TREE-WALK ( izquierda [ x ])
3                 imprimir clave [ x ]
4                 INORDER-TREE-WALK ( derecha [ x ])
```

A modo de ejemplo, el árbol inorder walk imprime las claves en cada uno de los dos árboles de búsqueda binaria de la Figura 12.1 en el orden 2, 3, 5, 5, 7, 8. La exactitud del algoritmo sigue por inducción directamente desde la propiedad binary-search-tree.

Se necesita Θ (n) tiempo para recorrer un árbol de búsqueda binaria de n nodos, ya que después de la llamada inicial, El procedimiento se llama de forma recursiva exactamente dos veces para cada nodo del árbol, una vez para su hijo izquierdo y una vez por su hijo adecuado. El siguiente teorema da una prueba más formal de que se necesita tiempo lineal para realizar una caminata de árbol en orden.

Teorema 12.1

Página 222

Si x es la raíz de un subárbol de n nodos, entonces la llamada INORDER-TREE-WALK (x) toma Θ (n) hora.

Prueba Sea T (n) el tiempo que tarda INORDER-TREE-WALK cuando es llamado en el raíz de un subárbol de n nodos. INORDER-TREE-WALK lleva una pequeña cantidad de tiempo constante en un subárbol vacío (para la prueba $x \neq$ NIL), por lo que $T(0) = c$ para alguna constante positiva c .

Para $n > 0$, suponga que INORDER-TREE-WALK se llama en un nodo x cuyo subárbol izquierdo tiene k nodos y cuyo subárbol derecho tiene $n - k - 1$ nodos. El momento de realizar INORDER-TREE-WALK (x) es $T(n) = T(k) + T(n - k - 1) + d$ para alguna constante positiva d que refleja el tiempo hasta ejecutar INORDER-TREE-WALK (x), sin incluir el tiempo empleado en llamadas recursivas.

Usamos el método de sustitución para mostrar que $T(n) = \Theta(n)$ probando que $T(n) = (c + d)n + c$. Para $n = 0$, tenemos $(c + d) \cdot 0 + c = c = T(0)$. Para $n > 0$, tenemos

$$\begin{aligned} T(norte) &= T(k) + T(norte - k - 1) + d \\ &= ((c + re)k + c) + ((c + re)(norte - k - 1) + c) + re \\ &= (c + d)norte + c - (c + d) + c + d \\ &= (c + d)n + c , \end{aligned}$$

que completa la prueba.

Ejercicios 12.1-1

Para el conjunto de claves {1, 4, 5, 10, 16, 17, 21}, dibuje árboles de búsqueda binarios de altura 2, 3, 4, 5 y 6.

Ejercicios 12.1-2

¿Cuál es la diferencia entre la propiedad binary-search-tree y la propiedad min-heap (ver página 129)? ¿Se puede usar la propiedad min-heap para imprimir las claves de un árbol de n nodos en orden orden en $O(n)$ tiempo? Explique cómo o por qué no.

Ejercicios 12.1-3

Proporcione un algoritmo no recursivo que realice una caminata de árbol en orden. (*Pista:* hay una solución que utiliza una pila como estructura de datos auxiliar y una más complicada pero elegante solución que no usa pila, pero asume que se pueden probar dos punteros para verificar la igualdad).

Página 223

Ejercicios 12.1-4

Proporcione algoritmos recursivos que realicen caminatas de árbol de preorden y postorder en $\Theta(n)$ tiempo en un árbol de n nodos.

Ejercicios 12.1-5

Argumenta que, dado que ordenar n elementos toma $\Omega(n \lg n)$ tiempo en el peor de los casos en la comparación modelo, cualquier algoritmo basado en comparación para construir un árbol de búsqueda binaria a partir de un una lista arbitraria de n elementos toma $\Omega(n \lg n)$ tiempo en el peor de los casos.

12.2 Consultar un árbol de búsqueda binario

Una operación común que se realiza en un árbol de búsqueda binaria es buscar una clave almacenada en el árbol. Además de la operación de BÚSQUEDA, los árboles de búsqueda binarios pueden admitir consultas como MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR. En esta sección,
Examine estas operaciones y demuestre que cada una puede ser soportada en el tiempo $O(h)$ en una búsqueda binaria. árbol de altura h .

buscando

Usamos el siguiente procedimiento para buscar un nodo con una clave dada en un árbol de búsqueda binaria. Dado un puntero a la raíz del árbol y una clave k , TREE-SEARCH devuelve un puntero a un nodo con la tecla k si existe; de lo contrario, devuelve NIL.

```
BÚSQUEDA DE ÁRBOL ( x , k )
1 si x = NIL o k = clave [ x ]
2 luego devuelve x
3 si k < tecla [ x ]
4 luego regrese BÚSQUEDA DE ÁRBOL ( izquierda [ x ], k )
5 si no, devuelve BÚSQUEDA DE ÁRBOL ( derecha [ x ], k )
```

El procedimiento comienza su búsqueda en la raíz y traza un camino hacia abajo en el árbol, como se muestra. en la Figura 12.2 . Para cada nodo x que encuentra, compara la clave k con la *clave* [x]. Si los dos las claves son iguales, la búsqueda termina. Si k es menor que la *tecla* [x], la búsqueda continúa en el subárbol izquierdo de x , ya que la propiedad binary-search-tree implica que k no podría almacenarse en el subárbol derecho. Simétricamente, si k es mayor que la *clave* [x], la búsqueda continúa a la derecha subárbol. Los nodos encontrados durante la recursividad forman un camino hacia abajo desde la raíz de el árbol, y por lo tanto el tiempo de ejecución de TREE-SEARCH es $O(h)$, donde h es la altura de la árbol.

Página 224

Figura 12.2: Consultas en un árbol de búsqueda binaria. Para buscar la clave 13 en el árbol, seguimos la ruta 15 → 6 → 7 → 13 desde la raíz. La clave mínima en el árbol es 2, que se puede encontrado siguiendo los punteros *izquierdos* desde la raíz. La clave máxima 20 se encuentra siguiendo punteros *correctos* desde la raíz. El sucesor del nodo con clave 15 es el nodo con clave 17, ya que es la clave mínima en el subárbol derecho de 15. El nodo con clave 13 no tiene derecho subárbol, y por lo tanto su sucesor es su antepasado más bajo cuyo hijo izquierdo es también un antepasado. En esto caso, el nodo con clave 15 es su sucesor.

El mismo procedimiento se puede escribir de forma iterativa por "desenrollar" la recursión en un **mientras** bucle.
En la mayoría de las computadoras, esta versión es más eficiente.

```
BÚSQUEDA-ÁRBOL-ITERATIVO ( x , k )
1 mientras que x ≠ NIL yk ≠ tecla [ x ]
2 hacer si k < tecla [ x ]
3                    luego x ← izquierda [ x ]
4                    else x ← derecha [ x ]
5 devuelve x
```

Mínimo y máximo

Un elemento en un árbol de búsqueda binario cuya clave es un mínimo siempre se puede encontrar siguiendo punteros secundarios *izquierdos* desde la raíz hasta que se encuentra un NIL, como se muestra en la Figura 12.2 . los
El siguiente procedimiento devuelve un puntero al elemento mínimo en el subárbol enraizado en un determinado nodo x .

```
ÁRBOL-MÍNIMO ( x )
1 mientras queda [ x ] ≠ NIL
2 hacer x ← izquierda [ x ]
3 volver x
```

La propiedad binary-search-tree garantiza que TREE-MINIMUM es correcto. Si un nodo x tiene sin subárbol izquierdo, entonces dado que cada clave en el subárbol derecho de x es al menos tan grande como *clave* [x], el la clave mínima en el subárbol con raíz en x es la *clave* [x]. Si el nodo x tiene un subárbol izquierdo, entonces como no la clave del subárbol derecho es más pequeña que la *clave* [x] y todas las claves del subárbol izquierdo no son más grandes que la *clave* [x], la clave mínima en el subárbol enraizado en x se puede encontrar en el subárbol enraizado en *izquierda* [x].

El pseudocódigo de TREE-MAXIMUM es simétrico.

```
ÁRBOL-MÁXIMO ( x )
1 mientras que a la derecha [ x ] ≠ NIL
2 hacer x ← derecha [ x ]
3 volver x
```

Página 225

Ambos procedimientos se ejecutan en tiempo O (h) en un árbol de altura h ya que, como en TREE-SEARCH, la secuencia de nodos encontrados forma un camino hacia abajo desde la raíz.

Sucesor y predecesor

Dado un nodo en un árbol de búsqueda binario, a veces es importante poder encontrar su sucesor en el orden determinado por una caminata de árbol en orden. Si todas las claves son distintas, el sucesor de un nodo x es el nodo con la clave más pequeña mayor que la *clave* [x]. La estructura de una búsqueda binaria tree nos permite determinar el sucesor de un nodo sin siquiera comparar claves. los
El siguiente procedimiento devuelve el sucesor de un nodo x en un árbol de búsqueda binaria si existe, y NULO si x tiene la clave más grande del árbol.

```
ÁRBOL-SUCESOR ( x )
1 si es correcto [ x ] ≠ NIL
2 luego regresa TREE-MINIMUM ( derecha [ x ])
3 y ← p [ x ]
4 mientras y ≠ NIL y x = derecha [ y ]
5 hacer x ← y
6                y ← p [ y ]
7 volver y
```

El código de TREE-SUCCESSOR se divide en dos casos. Si el subárbol derecho del nodo x es no vacío, entonces el sucesor de x es solo el nodo más a la izquierda en el subárbol derecho, que se encuentra en la línea 2 llamando a TREE-MINIMUM (*derecha* [x]). Por ejemplo, el sucesor del nodo con La clave 15 en la Figura 12.2 es el nodo con la clave 17.

Por otro lado, como el ejercicio 12.2-6 le pide que muestre, si el subárbol derecho del nodo x está vacío y x tiene un sucesor y , entonces y es el antepasado más bajo de x cuyo hijo izquierdo también es un antepasado de x . En la figura 12.2 , el sucesor del nodo con clave 13 es el nodo con clave 15. Para encontrar y , simplemente subimos por el árbol desde x hasta que encontramos un nodo que es el hijo izquierdo de su padre; esto se logra con las líneas 3 a 7 de TREE-SUCCESSOR.

El tiempo de ejecución de TREE-SUCCESSOR en un árbol de altura h es $O(h)$, ya que seguimos un camino hasta el árbol o seguir un camino hacia abajo del árbol. El procedimiento ARBOL PREDECESOR, que es simétrico a TREE-SUCCESSOR, también corre en el tiempo $O(h)$.

Incluso si las claves no son distintas, definimos el sucesor y el predecesor de cualquier nodo x como el nodo devuelto por llamadas realizadas a TREE-SUCCESSOR (x) y TREE-PREDECESSOR (x), respectivamente.

En resumen, hemos probado el siguiente teorema.

Teorema 12.2

Las operaciones de conjuntos dinámicos SEARCH, MINIMUM, MAXIMUM, SUCCESSOR y
Se puede hacer que el PREDECESOR se ejecute en el tiempo $O(h)$ en un árbol de búsqueda binario de altura h .

Ejercicios 12.2-1

Página 226

Suponga que tenemos números entre 1 y 1000 en un árbol de búsqueda binaria y queremos buscar para el número 363. ¿Cuál de las siguientes secuencias *no* podría ser la secuencia de nodos? ¿examinado?

a. 2, 252, 401, 398, 330, 344, 397, 363.
si. 924, 220, 911, 244, 898, 258, 362, 363.
C. 925, 202, 911, 240, 912, 245, 363.
re. 2, 399, 387, 219, 266, 382, 381, 278, 363.
mi. 935, 278, 347, 621, 299, 392, 358, 363.

Ejercicios 12.2-2

Escriba versiones recursivas de los procedimientos TREE-MINIMUM y TREE-MAXIMUM.

Ejercicios 12.2-3

Escriba el procedimiento PREDECESOR DE ÁRBOL.

Ejercicios 12.2-4

El profesor Bunyan cree haber descubierto una propiedad notable de los árboles de búsqueda binarios. Suponga que la búsqueda de la clave k en un árbol de búsqueda binaria termina en una hoja. Considere tres conjuntos: A , las teclas a la izquierda de la ruta de búsqueda; B , las claves de la ruta de búsqueda; y C , las claves del a la derecha de la ruta de búsqueda. Reivindicaciones Profesor Bunyan que cualquiera de las tres teclas de *un* A , b B , y C C debe satisfacer $a \le b \le c$. Dé un pequeño contraejemplo posible a la afirmación del profesor.

Ejercicios 12.2-5

Demuestre que si un nodo en un árbol de búsqueda binario tiene dos hijos, entonces su sucesor no tiene izquierda hijo y su predecesor no tiene hijo derecho.

Ejercicios 12.2-6

Página 227

Considere un árbol de búsqueda binario T cuyas claves son distintas. Demuestre que si el subárbol derecho de un el nodo x en T está vacío y x tiene un sucesor y , entonces y es el antepasado más bajo de x cuyo hijo izquierdo también es un antepasado de x . (Recuerde que cada nodo es su propio antepasado).

Ejercicios 12.2-7

Se puede implementar un recorrido de árbol en orden de un árbol de búsqueda binaria de n nodos elemento mínimo en el árbol con TREE-MINIMUM y luego hacer n -1 llamadas a TREE-SUCESOR. Demuestre que este algoritmo se ejecuta en $\Theta (n)$ tiempo.

Ejercicios 12.2-8

Demostrar que no importa qué nodo empezamos en una altura- h árbol de búsqueda binaria, k llamadas sucesivas a TREE-SUCCESSOR toma $O (k + h)$ tiempo.

Ejercicios 12.2-9

Sea T un árbol de búsqueda binario cuyas claves son distintas, sea x un nodo hoja y sea y su padre. Muestre que la *clave* [y] es la clave más pequeña en T mayor que la *clave* [x] o la clave más grande en T menor que la *tecla* [x].

12.3 Inserción y eliminación

Las operaciones de inserción y eliminación provocan el conjunto dinámico representado por una búsqueda binaria árbol para cambiar. La estructura de datos debe modificarse para reflejar este cambio, pero de tal manera que la propiedad binary-search-tree continúa siendo válida. Como veremos, modificando el árbol para insertar un nuevo elemento es relativamente sencillo, pero manejar la eliminación es algo más intrincado.

Inserción

Para insertar un nuevo valor v en un árbol de búsqueda binario T , usamos el procedimiento TREE-INSERT. Al procedimiento se le pasa un nodo z para el cual *key* [z] = v , *left* [z] = NIL y *right* [z] = NIL. Eso modifica T y algunos de los campos de z de tal manera que z se inserta en un posición en el árbol.

```
INSERCIÓN DE ÁRBOL ( T , z )
 1 y ← NIL
 2 x ← raíz [ T ]
 3 mientras x ≠ NIL
 4 hacer y ← x
 5                    si tecla [ z ] < tecla [ x ]
```

Página 228

```
 6                    luego x ← izquierda [ x ]
 7                    else x ← derecha [ x ]
 8 p [ z ] ← y
 9 si y = NULO
10 luego root [ T ] ← z                          ⊹ El árbol T estaba vacío
11 else if tecla [ z ] < tecla [ y ]
12                    luego a la izquierda [ y ] ← z
13                    si no a la derecha [ y ] ← z
```

La figura 12.3 muestra cómo funciona TREE-INSERT. Al igual que los procedimientos TREE-SEARCH y ITERATIVE-TREE-SEARCH, TREE-INSERT comienza en la raíz del árbol y traza un camino hacia abajo. El puntero x traza la ruta y el puntero y se mantiene como padre de x . Después de la inicialización, el **tiempo** de bucle de las líneas 3-7 hace que estos dos punteros para bajar el árbol, yendo hacia la izquierda o hacia la derecha dependiendo de la comparación de la *tecla* [z] con la *tecla* [x], hasta que x se establezca en NULO. Este NIL ocupa la posición donde deseamos colocar el elemento de entrada z . Conjunto de líneas 8-13 los punteros que hacen que se inserte z .

Figura 12.3: Insertar un elemento con la clave 13 en un árbol de búsqueda binaria. Nodos ligeramente sombreados indique la ruta desde la raíz hasta la posición donde se inserta el elemento. El rayado La línea indica el enlace en el árbol que se agrega para insertar el elemento.

Como las otras operaciones primitivas en árboles de búsqueda, el procedimiento TREE-INSERT se ejecuta en $O (h)$ tiempo en un árbol de altura h .

Supresión

El procedimiento para eliminar un nodo dado z de un árbol de búsqueda binaria toma como argumento a puntero a z . El procedimiento considera los tres casos que se muestran en la Figura 12.4 . Si z no tiene children, modificamos su padre p [z] para reemplazar z con NIL como hijo. Si el nodo tiene solo un hijo único, "empalmamos" z haciendo un nuevo vínculo entre su hijo y su padre. Finalmente, si el nodo tiene dos hijos, empalmamos la sucesión *y de z* , que no tiene hijo izquierdo (ver Ejercicio 12,2-5) y reemplazar z 's de datos clave y vía satélite con Y ' s de los datos clave y de satélite.

Figura 12.4: Eliminación de un nodo z de un árbol de búsqueda binaria. Qué nodo se elimina realmente depende de cuántos hijos tenga z ; este nodo se muestra ligeramente sombreado. *(a)* Si z no tiene niños, simplemente lo quitamos. *(b)* Si z solo tiene un hijo, empalmamos z . *(c)* Si z tiene dos hijos, separamos su sucesor y , que tiene como máximo un hijo, y luego reemplazamos la tecla z y datos vía satélite con Y 's de datos clave y de satélite.

El código de TREE-DELETE organiza estos tres casos de forma un poco diferente.

```
BORRAR ÁRBOL ( T , z )
  1 si izquierda [ z ] = NIL o derecha [ z ] = NIL
  2 luego y ← z
  3 else y ← ÁRBOL-SUCESOR ( z )
  4 si deja [ y ] ≠ NIL
  5 luego x ← izquierda [ y ]
  6 más x ← derecha [ y ]
  7 si x ≠ NULO
  8 luego p [ x ] ← p [ y ]
  9 si p [ y ] = NULO
10 luego raíz [ T ] ← x
11 más si y = izquierda [ p [ y ]]
12                          luego a la izquierda [ p [ y ]] ← x
13                          si no a la derecha [ p [ y ]] ← x
14 si y ≠ z
15 luego tecla [ z ] ← tecla [ y ]
dieciséis                copiar Y los datos de satélite 's en z
17 regreso y
```

En las líneas 1 a 3, el algoritmo determina un nodo y para empalmar. El nodo y es la entrada nodo z (si z tiene como máximo 1 hijo) o el sucesor de z (si z tiene dos hijos). Luego, en las líneas 4–6, x se establece en el hijo no NIL de y , o en NIL si y no tiene hijos. El nodo y se empalma en líneas 7-13 modificando punteros en p [y] y x . Empalmar y es algo complicado por el necesidad de un manejo adecuado de las condiciones de contorno, que ocurren cuando x = NIL o cuando y es la raíz. Finalmente, en líneas 14-16, si el sucesor de z era el nodo empalmado a cabo, y clave 's y los datos de satélite se mueven a z , sobrescribiendo la clave anterior y los datos de satélite. El nodo y es devuelto en la línea 17 para que el procedimiento de llamada pueda reciclarlo a través de la lista libre. El procedimiento corre en tiempo O (h) en un árbol de altura h .

En resumen, hemos probado el siguiente teorema.

Página 230

Teorema 12.3

Las operaciones de conjunto dinámico INSERT y DELETE se pueden hacer para que se ejecuten en tiempo O (h) en un árbol de búsqueda binaria de altura h .

Ejercicios 12.3-1

Proporcione una versión recursiva del procedimiento TREE-INSERT.

Ejercicios 12.3-2

Suponga que un árbol de búsqueda binario se construye insertando repetidamente valores distintos en el árbol. Argumenta que el número de nodos examinados al buscar un valor en el árbol es uno más el número de nodos examinados cuando el valor se insertó por primera vez en el árbol.

Ejercicios 12.3-3

Podemos ordenar un conjunto dado de n números construyendo primero un árbol de búsqueda binario que contenga estos números (usando TREE-INSERT repetidamente para insertar los números uno por uno) y luego imprimiendo los números por un árbol en orden. ¿Cuáles son el peor y el mejor caso en ejecución? tiempos para este algoritmo de clasificación?

Ejercicios 12.3-4

Suponga que otra estructura de datos contiene un puntero a un nodo y en un árbol de búsqueda binaria, y Supongamos que Y 's predecesor z se elimina del árbol por el árbol-BORRAR procedimiento. ¿Qué problema puede surgir? ¿Cómo se puede reescribir TREE-DELETE para resolver este problema?

Ejercicios 12.3-5

¿Es la operación de eliminación "conmutativa" en el sentido de que eliminar x y luego y de un El árbol de búsqueda binaria deja el mismo árbol que eliminar y y luego x ? Discute por qué es o da un contraejemplo.

Página 231

Ejercicios 12.3-6

Cuando el nodo z en TREE-DELETE tiene dos hijos, podríamos empalmar a su predecesor en lugar de que su sucesor. Algunos han argumentado que una estrategia justa, dando igual prioridad al predecesor y sucesor, produce un mejor desempeño empírico. ¿Cómo se puede cambiar TREE-DELETE? para implementar una estrategia tan justa?

12.4 Árboles de búsqueda binarios construidos aleatoriamente

Hemos demostrado que todas las operaciones básicas en un árbol de búsqueda binaria se ejecutan en el tiempo $O(h)$, donde h es la altura del árbol. Sin embargo, la altura de un árbol de búsqueda binaria varía a medida que se insertado y eliminado. Si, por ejemplo, los elementos se insertan en un orden estrictamente creciente, el árbol será una cadena con altura n - 1. Por otro lado, el ejercicio B.5-4 muestra que $h \geq \lfloor \lg n \rfloor$. Como con quicksort, podemos mostrar que el comportamiento del caso promedio está mucho más cerca del mejor caso que el peor de los casos.

Desafortunadamente, poco se sabe acerca de la altura promedio de un árbol de búsqueda binaria cuando ambos la inserción y la eliminación se utilizan para crearlo. Cuando el árbol se crea solo por inserción, el el análisis se vuelve más manejable. Por lo tanto, definamos un *árbol de búsqueda binario construido aleatoriamente* en n claves como una que surge de insertar las claves en orden aleatorio en un inicialmente vacío árbol, donde cada uno de los n ! Las permutaciones de las claves de entrada son igualmente probables. (Ejercicio 12.4-3 le pide que demuestre que esta noción es diferente de suponer que cada árbol de búsqueda binaria en n claves es igualmente probable.) En esta sección, mostraremos que la altura esperada de un El árbol de búsqueda binario construido en n claves es $O(\lg n)$. Suponemos que todas las claves son distintas.

Comenzamos por definir tres variables aleatorias que ayudan a medir la altura de un edificio construido al azar. árbol de búsqueda binaria. Denotamos la altura de una búsqueda binaria construida aleatoriamente en n claves por X_n, y definimos la *altura exponencial* . Cuando construimos un árbol de búsqueda binario en n claves, elegimos una clave como la de la raíz, y dejamos que R_n denote la variable aleatoria que contiene esta rango de la clave dentro del conjunto de n claves. Es igualmente probable que el valor de R_n sea cualquier elemento de la establecer $\{1, 2, ..., n\}$. Si $R_n = i$, entonces el subárbol izquierdo de la raíz es un árbol de búsqueda binario construido al azar en las teclas i - 1, y el subárbol derecho es un árbol de búsqueda binario construido aleatoriamente en las teclas n - i. Porque la altura de un árbol binario es uno más que la mayor de las alturas de los dos subárboles del raíz, la altura exponencial de un árbol binario es dos veces mayor de las alturas exponenciales de los dos subárboles de la raíz. Si sabemos que $R_n = i$, entonces tenemos que

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}).$$

Como casos base, tenemos $Y_1 = 1$, porque la altura exponencial de un árbol con 1 nodo es $2^0 = 1$ y, por conveniencia, definimos $Y_0 = 0$.

A continuación, definimos las variables aleatorias del indicador $Z_{n,1}, Z_{n,2}, ..., Z_{n,n}$, donde

$Z_{n,i} = I\{R_n = i\}$.

Página 232

Dado que es igualmente probable que R_n sea cualquier elemento de $\{1, 2, ..., n\}$, tenemos que $\Pr\{R_n = i\} = 1/n$ para $i = 1,2, ..., n$, y por tanto, según el Lema 5.1 ,

(12,1)

para $i = 1, 2, ..., n$. Dado que exactamente un valor de $Z_{n,i}$ es 1 y todos los demás son 0, también tenemos

Demostraremos que $E[Y_n]$ es polinomio en n, lo que finalmente implicará que $E[X_n] = O(\lg n)$.

La variable aleatoria indicadora $Z_{n,i} = I\{R_n = i\}$ es independiente de los valores de Y_{i-1} e Y_{n-i}. Habiendo elegido $R_n = i$, el subárbol izquierdo, cuya altura exponencial es Y_{i-1}, se construye aleatoriamente sobre las teclas $i - 1$ cuyos rangos son menores que i. Este subárbol es como cualquier otro construido aleatoriamente árbol de búsqueda binaria en las teclas $i - 1$. Aparte del número de claves que contiene, este subárbol la estructura no se ve afectada en absoluto por la elección de $R_n = i$; de ahí las variables aleatorias Y_{i-1} y $Z_{n,i}$ son independientes. Asimismo, el subárbol derecho, cuya altura exponencial es Y_{n-i}, es aleatoriamente construido sobre las claves $n - i$ cuyos rangos son mayores que i. Su estructura es independiente del valor de R_n, por lo que las variables aleatorias Y_{n-i} y $Z_{n,i}$ son independientes. Por lo tanto,

Cada término $E[Y_0], E[Y_1], ..., E[Y_{n-1}]$ aparece dos veces en la última suma, una vez como $E[Y_{i-1}]$ y una vez como $E[Y_{n-i}]$, por lo que tenemos la recurrencia

(12,2)

Usando el método de sustitución, mostraremos que para todos los enteros positivos n, la recurrencia (12.2) tiene la solución

Al hacerlo, usaremos la identidad

Página 233

(12,3)

(El ejercicio 12.4-1 le pide que pruebe esta identidad).

Para el caso base, verificamos que el límite

sostiene. Para la sustitución, tenemos que

Hemos acotado E [Y_n], pero nuestro objetivo final es limitar E [X_n]. Como le pide el ejercicio 12.4-4 para mostrar, la función $f(x) = 2^x$ es convexa (consulte la página 1109). Por lo tanto, podemos aplicar Jensen's desigualdad (C.25), que dice que

para derivar eso

Tomando logaritmos de ambos lados da E [X_n] = O (lg n). Por lo tanto, hemos probado lo siguiente:

Teorema 12.4

La altura esperada de un árbol de búsqueda binario construido aleatoriamente en n claves es O (lg n).

Página 234

Ejercicios 12.4-1

Demuestre la ecuación (12.3) .

Ejercicios 12.4-2

Describa un árbol de búsqueda binaria en n nodos de manera que la profundidad promedio de un nodo en el árbol sea $\Theta(\lg n)$ pero la altura del árbol es $\omega(\lg n)$. Dar un límite superior asintótico en la altura de un árbol de búsqueda binaria de n nodos en el que la profundidad media de un nodo es $\Theta(\lg n)$.

Ejercicios 12.4-3

Muestre que la noción de un árbol de búsqueda binario elegido al azar en n claves, donde cada binario Es igualmente probable que se elija un árbol de búsqueda de n claves, es diferente de la noción de una árbol de búsqueda binario construido que se proporciona en esta sección. (*Sugerencia:* enumere las posibilidades cuando $n = 3$.)

Ejercicios 12.4-4

Demuestre que la función $f(x) = 2^x$ es convexa.

Ejercicios 12.4-5:

Considere RANDOMIZED-QUICKSORT operando en una secuencia de n números de entrada. Probar que para cualquier constante $k > 0$, todos menos $O(1 / n^k)$ del $n!$ las permutaciones de entrada producen una $O(n \lg n)$ tiempo de ejecución.

Problemas 12-1: árboles de búsqueda binaria con claves iguales

Las claves iguales plantean un problema para la implementación de árboles de búsqueda binarios.

a. ¿Cuál es el rendimiento asintótico de TREE-INSERT cuando se usa para insertar n elementos? con claves idénticas en un árbol de búsqueda binario inicialmente vacío?

Página 235

Proponemos mejorar TREE-INSERT probando antes de la línea 5 si *clave* [z] = *clave* [x] y probando antes de la línea 11 si *clave* [z] = *clave* [y]. Si se cumple la igualdad, implementamos una de las siguientes estrategias. Para cada estrategia, encuentre el rendimiento asintótico de insertar n elementos con claves idénticas en un árbol de búsqueda binario inicialmente vacío. (Las estrategias se describen por la línea 5, en el que comparamos las llaves de z y x . Sustituya y por x para llegar en las estrategias de la línea 11.)

si. Mantenga una bandera booleana b [x] en el nodo x , y establezca x a la *izquierda* [x] o a la *derecha* [x] según el valor de b [x], que alterna entre FALSO y VERDADERO cada vez que se visita x durante la inserción de un nodo con la misma clave que x .

C. Mantenga una lista de nodos con claves iguales en x e inserte z en la lista.

re. Establezca x aleatoriamente a la *izquierda* [x] o a la *derecha* [x]. (Dar el peor de los casos

y derivar informalmente el desempeño promedio de casos).

Problemas 12-2: Árboles de radix

Dadas dos cadenas $a = a_0 a_1 \ldots a_p$ y $b = b_0 b_1 \ldots b_q$, donde cada a_i y cada b_j está en algún orden conjunto de caracteres, decimos que la cadena a es lexicográficamente menor que la cadena b si

1. existe un número entero j , donde $0 \le j \le \min(p, q)$, tal que $a_i = b_i$ para todo $i = 0, 1, \ldots, j - 1$ y $a_j < b_j$, o

2. $p < q$ y $a_i = b_i$ para todo $i = 0, 1, ..., p$.

Por ejemplo, si *un* y *b* son cadenas de bits, entonces 10100 <10.110 por la regla 1 (dejando que $j = 3$) y 10 100 <101000 según la regla 2. Esto es similar al orden utilizado en los diccionarios de idioma inglés.

La estructura de datos del árbol de la base que se muestra en la Figura 12.5 almacena las cadenas de bits 1011, 10, 011, 100, y 0. Al buscar una clave $a = a_0 a_1 ... a_p$, vamos a la izquierda en un nodo de profundidad i si $a_i = 0$ y a la derecha si $a_i = 1$. Sea S un conjunto de cadenas binarias distintas cuyas longitudes suman n . Muestre cómo usar un árbol de raíz para ordenar S lexicográficamente en $\Theta(n)$ tiempo. Para el ejemplo de la Figura 12.5 , la salida del tipo debe ser la secuencia 0, 011, 10, 100, 1011.

Figura 12.5: Un árbol de base que almacena las cadenas de bits 1011, 10, 011, 100 y 0. La clave de cada nodo puede determinarse atravesando la ruta desde la raíz hasta ese nodo. Por tanto, no es necesario almacenar las claves en los nodos; las claves se muestran aquí solo con fines ilustrativos. Los nodos son muy sombreado si las claves que les corresponden no están en el árbol; tales nodos están presentes solo para establecer un camino a otros nodos.

Página 236

Problemas 12-3: profundidad de nodo promedio en un árbol de búsqueda binario construido al azar

En este problema, probamos que la profundidad promedio de un nodo en una búsqueda binaria construida al azar árbol con n nodos es $O(\lg n)$. Aunque este resultado es más débil que el del teorema 12.4 , el técnica que usaremos revela una sorprendente similitud entre la construcción de una búsqueda binaria tree y la ejecución de RANDOMIZED-QUICKSORT de la Sección 7.3 .

Definimos la longitud total del camino $P(T)$ de un árbol binario T como la suma, sobre todos los nodos x en T , de la profundidad del nodo x , que denotamos por $d(x, T)$.

a. Argumenta que la profundidad promedio de un nodo en T es

Por lo tanto, deseamos mostrar que el valor esperado de $P(T)$ es $O(n \lg n)$.

si. Sean T_L y T_R los subárboles izquierdo y derecho del árbol T , respectivamente. Argumenta que si T tiene n nodos, entonces

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

C. Sea $P(n)$ la longitud de ruta total promedio de un árbol de búsqueda binario construido aleatoriamente con n nodos. Muestra esa

re. Muestre que $P(n)$ se puede reescribir como

mi. Recordando el análisis alternativo de la versión aleatoria de clasificación rápida que se proporciona en Problema 7-2 , concluya que $P(n) = O(n \lg n)$.

En cada invocación recursiva de quicksort, elegimos un elemento pivote aleatorio para dividir el

conjunto de elementos que se ordenan. Cada nodo de un árbol de búsqueda binario divide el conjunto de elementos que caen en el subárbol enraizado en ese nodo.

F. Describir una implementación de ordenación rápida en la que las comparaciones para ordenar un conjunto de Los elementos son exactamente iguales a las comparaciones para insertar los elementos en un binario. árbol de búsqueda. (El orden en el que se hacen las comparaciones puede diferir, pero el mismo deben hacerse comparaciones.)

Página 237

Problemas 12-4: Número de árboles binarios diferentes

Sea b_n el número de árboles binarios diferentes con n nodos. En este problema, encontrará una fórmula para b_n, así como una estimación asintótica.

a. Demuestre que $b_0 = 1$ y que, para $n \geq 1$,

si. Con referencia al problema 4-5 para la definición de una función generadora, sea $B(x)$ función generadora

Demuestre que $B(x) = xB(x)^2 + 1$, y por lo tanto, una forma de expresar $B(x)$ en forma cerrada es

La expansión de Taylor de $f(x)$ alrededor del punto $x = a$ está dada por

donde $f^{(k)}(x)$ es la k- ésima derivada de f evaluada en x.

C. Muestra esa

(el n- ésimo número catalán) utilizando la expansión de Taylor de alrededor de $x = 0$. (Si que desee, en lugar de utilizar la expansión de Taylor, puede utilizar la generalización de la expansión binomial (C.4) a exponentes no integrales n, donde para cualquier número real n y para cualquier entero k, interpretamos que es $n(n-1)(n-k+1)/k!$ si $k \geq 0$, y 0 en caso contrario.)

re. Muestra esa

Notas del capítulo

Knuth [185] contiene una buena discusión de árboles de búsqueda binarios simples, así como muchos variaciones. Los árboles de búsqueda binaria parecen haber sido descubiertos de forma independiente por varios

Página 238

gente a finales de la década de 1950. Los árboles de radix a menudo se denominan intentos, que provienen del medio letras en la recuperación de palabras. Knuth también las analiza [185] .

La sección 15.5 mostrará cómo construir un árbol de búsqueda binario óptimo cuando se busca Las frecuencias se conocen antes de construir el árbol. Es decir, dadas las frecuencias de búsqueda de cada clave y las frecuencias de búsqueda de valores que caen entre claves en el árbol, construimos un árbol de búsqueda binario para el cual un conjunto de búsquedas que sigue a estos Frecuencias examina el número mínimo de nodos.

La prueba de la Sección 12.4 que limita la altura esperada de una búsqueda binaria construida al azar árbol se debe a Aslam [23] . Mart nez y Roura [211] proporcionan algoritmos aleatorios para la inserción y supresión de árboles de búsqueda binarios en los que el resultado de cualquiera de las operaciones es un árbol de búsqueda binaria. Su definición de un árbol de búsqueda binario aleatorio difiere ligeramente de la de un árbol de búsqueda binario construido aleatoriamente en este capítulo, sin embargo.

Capítulo 13: Árboles Rojo-Negro

El capítulo 12 mostró que un árbol de búsqueda binario de altura h puede implementar cualquiera de los operaciones de conjuntos dinámicos, como BÚSQUEDA, PREDECESOR, SUCESOR, MÍNIMO, MÁXIMO, INSERTAR y ELIMINAR — en el tiempo $O(h)$. Por tanto, las operaciones de conjunto son rápidas si el la altura del árbol de búsqueda es pequeña; pero si su altura es grande, su rendimiento puede no ser mejor que con una lista vinculada. Los árboles rojo-negro son uno de los muchos esquemas de árbol de búsqueda que son "equilibrado" para garantizar que las operaciones básicas del conjunto dinámico toman $O(\lg n)$ tiempo en el peor de los casos.

13.1 Propiedades de los árboles rojo-negros

Un **árbol rojo-negro** es un árbol de búsqueda binario con un bit extra de almacenamiento por nodo: su **color** , que puede ser ROJO o NEGRO. Restringiendo la forma en que los nodos se pueden colorear en cualquier ruta de la raíz a la hoja, los árboles rojo-negros aseguran que ningún camino sea más del doble de largo que cualquier otro, para que el árbol esté aproximadamente **equilibrado** .

Cada nodo del árbol ahora contiene los campos *color* , *clave* , *izquierda* , *derecha* y *p* . Si un niño o el padre de un nodo no existe, el campo de puntero correspondiente del nodo contiene el valor NULO. Consideraremos estos NIL como punteros a nodos externos (hojas) del binario árbol de búsqueda y los nodos portadores de claves normales como nodos internos del árbol.

Un árbol de búsqueda binaria es un árbol rojo-negro si satisface las siguientes **propiedades rojo-negro** :

1. Cada nodo es rojo o negro.
2. La raíz es negra.
3. Cada hoja (NIL) es negra.
4. Si un nodo es rojo, entonces sus dos hijos son negros.
5. Para cada nodo, todas las rutas desde el nodo hasta las hojas descendientes contienen el mismo número de nodos negros.

La figura 13.1 (a) muestra un ejemplo de un árbol rojo-negro.

Página 239

Figura 13.1: Un árbol rojo-negro con nodos negros oscurecidos y nodos rojos sombreados. Cada nodo en un árbol rojo-negro es rojo o negro, los hijos de un nodo rojo son ambos negros, y cada La ruta simple de un nodo a una hoja descendiente contiene el mismo número de nodos negros. *(una)* Cada hoja, mostrada como NIL, es negra. Cada nodo que no es NIL está marcado con su altura en negro; NIL tiene negro de altura 0. *(b)* El mismo árbol rojo-negro pero con cada NIL reemplazado por el centinela único *nulo* [*T*], que siempre es negro, y se omiten las alturas negras. El padre de la raíz también es el centinela. *(c)* El mismo árbol rojo-negro pero con hojas y el padre de la raíz omitido enteramente. Usaremos este estilo de dibujo en el resto de este capítulo.

Como cuestión de conveniencia al tratar con las condiciones de contorno en el código de árbol rojo-negro, use un solo centinela para representar NIL (vea la página 206). Para un árbol rojo-negro T , el centinela *nulo* [T] es un objeto con los mismos campos que un nodo ordinario del árbol. Su campo de *color* es NEGRO, y sus otros campos , p , *izquierda* , *derecha* y *clave, se* pueden establecer en valores arbitrarios. Como figura 13.1 (b) muestra, todos los punteros a NIL se reemplazan por punteros al centinela *nil* [T].

Usamos el centinela para poder tratar a un hijo NIL de un nodo x como un nodo ordinario cuyo padre es x . Aunque en su lugar podríamos agregar un nodo centinela distinto para cada NIL en el árbol, entonces que el padre de cada NIL está bien definido, ese enfoque desperdiciaría espacio. En su lugar, usamos el centinela *nil* [T] para representar todos los NIL: todas las hojas y el padre de la raíz. Los valores de los campos p , *izquierda* , *derecha* y *clave* del centinela son inmateriales, aunque podemos establecerlos durante el transcurso de un procedimiento para nuestra conveniencia.

Generalmente, limitamos nuestro interés a los nodos internos de un árbol rojo-negro, ya que contienen el valores clave. En el resto de este capítulo, omitimos las hojas cuando dibujamos árboles rojo-negro, como se muestra en la Figura 13.1 (c) .

Llamamos al número de nodos negros en cualquier camino desde, pero sin incluir, un nodo x hasta un leaf la altura negra del nodo, denotado bh (x). Por propiedad 5, la noción de altura del negro es bien definido, ya que todos los caminos descendentes del nodo tienen el mismo número de nodos negros. Definimos la altura negra de un árbol rojo-negro como la altura negra de su raíz.

Página 240

El siguiente lema muestra por qué los árboles rojo-negro son buenos árboles de búsqueda.

Lema 13.1

Un árbol rojo-negro con n nodos internos tiene una altura máxima de 2 lg (n + 1).

Prueba Comenzamos mostrando que el subárbol enraizado en cualquier nodo x contiene al menos $2^{bh(x)} - 1$ nodos internos. Demostramos esta afirmación por inducción sobre la altura de x . Si la altura de x es 0, entonces x debe ser una hoja (*nil* [T]), y el subárbol enraizado en x contiene al menos $2^{bh(x)} - 1 = 2^0 - 1 =$ 0 nodos internos. Para el paso inductivo, considere un nodo x que tiene altura positiva y es un nodo interno con dos hijos. Cada niño tiene una altura de negro de bh (x) o bh (x) - 1, dependiendo de si su color es rojo o negro, respectivamente. Dado que la altura de un hijo de x es menor que la altura de x misma, podemos aplicar la hipótesis inductiva para concluir que cada el niño tiene al menos $2^{bh(x)-1} - 1$ nodos internos. Por lo tanto, el subárbol enraizado en x contiene al menos $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodos internos, lo que prueba la afirmación.

Para completar la demostración del lema, sea h la altura del árbol. Según la propiedad 4, en al menos la mitad de los nodos en cualquier camino simple desde la raíz a una hoja, sin incluir la raíz, deben ser negro. En consecuencia, la altura de negro de la raíz debe ser de al menos $h / 2$; así,

$n \geq 2^{h/2} - 1$.

Moviendo el 1 al lado izquierdo y tomando logaritmos en ambos lados, se obtiene lg (n + 1) ≥ h / 2, o h ≤ 2 lg (n + 1).

Una consecuencia inmediata de este lema es que las operaciones de conjuntos dinámicos SEARCH, MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR se pueden implementar en O (lg n) tiempo en árboles rojo-negro, ya que se puede hacer que se ejecuten en O (h) tiempo en un árbol de búsqueda de altura h (como se muestra en el Capítulo 12) y cualquier árbol rojo-negro en n nodos es un árbol de búsqueda con altura O (lg n). (Por supuesto, las referencias a NIL en los algoritmos del Capítulo 12 deberían ser reemplazado por *nil* [T].) Aunque los algoritmos TREE-INSERT y TREE-DELETE de El capítulo 12 se ejecuta en tiempo O (lg n) cuando se le da un árbol rojo-negro como entrada, no admiten las operaciones de conjunto dinámico INSERT y DELETE, ya que no garantizan que el árbol de búsqueda binaria modificado será un árbol rojo-negro. Veremos en las Secciones 13.3 y 13.4 , sin embargo, que estas dos operaciones pueden ser soportadas en tiempo O (lg n).

Ejercicios 13.1-1

En el estilo de la Figura 13.1 (a) , dibuje el árbol de búsqueda binario completo de altura 3 en las teclas {1, 2, ..., 15}. Agregue las hojas NIL y coloree los nudos de tres maneras diferentes de modo que el negro-las alturas de los árboles rojo-negros resultantes son 2, 3 y 4.

Ejercicios 13.1-2

Página 241

Dibuje el árbol rojo-negro que resulta después de llamar a TREE-INSERT en el árbol de la Figura 13.1 con la tecla 36. Si el nodo insertado es de color rojo, ¿es el árbol resultante un árbol rojo-negro? Que si es de color negro?

Ejercicios 13.1-3

Definamos un árbol rojo-negro relajado como un árbol de búsqueda binario que satisface el rojo-negro propiedades 1, 3, 4 y 5. En otras palabras, la raíz puede ser roja o negra. Considere un árbol relajado rojo-negro T cuya raíz es roja. Si coloreamos la raíz de T *de* negro pero no hacemos otra cambia a T , ¿es el árbol resultante un árbol rojo-negro?

Ejercicios 13.1-4

Supongamos que "absorbemos" cada nodo rojo en un árbol rojo-negro en su padre negro, de modo que el los hijos del nodo rojo se convierten en hijos del padre negro. (Ignore lo que le sucede al claves.) ¿Cuáles son los grados posibles de un nodo negro después de que se absorben todos sus hijos rojos? ¿Qué puedes decir sobre la profundidad de las hojas del árbol resultante?

Ejercicios 13.1-5

Demuestre que el camino simple más largo desde un nodo x en un árbol rojo-negro hasta una hoja descendiente tiene la longitud como máximo el doble que la ruta simple más corta desde el nodo x hasta una hoja descendiente.

Ejercicios 13.1-6

¿Cuál es el mayor número posible de nodos internos en un árbol rojo-negro con altura negra k ?
¿Cuál es el número más pequeño posible?

Ejercicios 13.1-7

Describe un árbol rojo-negro en n teclas que se da cuenta de la mayor proporción posible de nodos internos rojos a los nodos internos negros. ¿Cuál es esta relación? ¿Qué árbol tiene la proporción más pequeña posible y qué es la razón?

Página 242

13.2 Rotaciones

Las operaciones del árbol de búsqueda TREE-INSERT y TREE-DELETE, cuando se ejecutan en un árbol rojo-negro con n teclas, tome O (lg n) tiempo. Debido a que modifican el árbol, el resultado puede violar el rojo. propiedades negras enumeradas en la Sección 13.1 . Para restaurar estas propiedades, debemos cambiar el colores de algunos de los nodos del árbol y también cambian la estructura del puntero.

Cambiamos la estructura del puntero mediante la rotación , que es una operación local en un árbol de búsqueda. que conserva la propiedad binary-search-tree. La figura 13.2 muestra los dos tipos de rotaciones: rotaciones a la izquierda y rotaciones a la derecha. Cuando hacemos una rotación a la izquierda en un nodo x , asumimos que su niño derecho y no es *nulo* [T]; x puede ser cualquier nodo en el árbol cuyo hijo derecho no sea *nil* [T]. los Giro a izquierdas "pivotes" en torno a la relación de X a Y . Hace y la nueva raíz del subárbol, con x como Y 's hijo izquierdo y y s hijo izquierdo como' x hijo derecho 's.

Figura 13.2: Las operaciones de rotación en un árbol de búsqueda binaria. La operación IZQUIERDA-ROTATE (T, x) transforma la configuración de los dos nodos de la izquierda en configuración a la derecha cambiando un número constante de punteros. La configuración en la derecha se puede transformar en la configuración de la izquierda mediante la operación inversa GIRAR A LA DERECHA (T, y). Las letras α, β y γ representan subárboles arbitrarios. Una rotacion La operación conserva la propiedad binary-search-tree: las claves en α preceden a la *clave* [x], que precede a las claves en β, que preceden a la *clave* [y], que precede a las claves en γ.

El pseudocódigo para ROTAR IZQUIERDA asume que *derecha* [x] ≠ *nil* [T] y que el padre de la raíz es *nulo* [T].

```
ROTACIÓN IZQUIERDA ( T , x )
 1 y ← derecha [ x ]                    ▸ Configure y .
 2 derecho [ x ] ← izquierda [ y ]      ▸ Turn Y 's subárbol izquierdo en x ' subárbol derecho s.
 3 p [ izquierda [ y ]] ← x
 4 p [ y ] ← p [ x ]                    ▸ Vincular el padre de x con y .
 5 si p [ x ] = cero [ T ]
 6 luego root [ T ] ← y
 7 más si x = izquierda [ p [ x ]]
 8                  luego a la izquierda [ p [ x ]] ← y
 9                  si no a la derecha [ p [ x ]] ← y
10 izquierda [ y ] ← x                  ▸ Put x en y S hacia la izquierda'.
11 p [ x ] ← y
```

La figura 13.3 muestra cómo funciona ROTACIÓN IZQUIERDA. El código para ROTAR A LA DERECHA es simétrico. Tanto ROTAR IZQUIERDA como ROTAR DERECHA se ejecutan en tiempo O (1). Solo los punteros son cambiado por una rotación; todos los demás campos de un nodo siguen siendo los mismos.

Página 243

Figura 13.3: Un ejemplo de cómo el procedimiento IZQUIERDA-ROTAR (T , x) modifica un binario árbol de búsqueda. Los recorridos por árbol en orden del árbol de entrada y el árbol modificado producen el mismo listado de valores clave.

Ejercicios 13.2-1

Escriba un pseudocódigo para RIGHT-ROTATE.

Ejercicios 13.2-2

Argumenta que en cada árbol de búsqueda binaria de n nodos, hay exactamente n - 1 rotaciones posibles.

Ejercicios 13.2-3

Deje un , b , y c ser nodos arbitrarios en subárboles alpha, β, y γ, respectivamente, en el árbol de la izquierda de la figura 13.2 . ¿Cómo hacer las profundidades de un , b , y c cambio cuando una rotación a la izquierda se lleva a cabo en el nodo x en ¿la figura?

Ejercicios 13.2-4

Demuestre que cualquier árbol de búsqueda binario de n nodos arbitrario se puede transformar en cualquier otro árbol de búsqueda binaria de n- nodos usando O (n) rotaciones. (*Sugerencia:* primero muestre que como máximo n - 1 a la derecha las rotaciones son suficientes para transformar el árbol en una cadena que va hacia la derecha).

Ejercicios 13.2-5: ⋆

Página 244

Decimos que un árbol de búsqueda binario T_1 se puede convertir a la derecha en un árbol de búsqueda binario T_2 si es posible obtener T_2 de T_1 a través de una serie de llamadas a ROTAR DERECHA. Dar un ejemplo de dos árboles T_1 y T_2 de modo que T_1 no se pueda convertir a la derecha en T_2 . Luego demuestre que si un árbol T_1 se puede convertir a la derecha a T_2 , se puede convertir a la derecha usando llamadas O (n_2) a RIGHT-ROTATE.

13.3 Inserción

La inserción de un nodo en un árbol rojo-negro de n nodos se puede lograr en O (lg n) tiempo. Usamos una versión ligeramente modificada del procedimiento TREE-INSERT (Sección 12.3) para insertar el nodo z en el árbol T como si fuera un árbol de búsqueda binario ordinario, y luego coloreamos z rojo. A garantizar que se conservan las propiedades rojo-negro, luego llamamos a un procedimiento auxiliar RB-INSERT-FIXUP para cambiar el color de los nodos y realizar rotaciones. La llamada RB-INSERT (T , z) inserta el nodo z , cuyo campo *clave* se supone que ya ha sido completado, en el rojo-negro árbol T .

```
INSERCIÓN RB ( T , z )
 1 y ← nil [ T ]
 2 x ← raíz [ T ]
 3 mientras que x ≠ cero [ T ]
 4 hacer y ← x
 5                 si tecla [ z ] < tecla [ x ]
 6                       luego x ← izquierda [ x ]
 7                       else x ← derecha [ x ]
 8 p [ z ] ← y
 9 si y = cero [ T ]
10 luego root [ T ] ← z
11 else if tecla [ z ] < tecla [ y ]
12                        luego a la izquierda [ y ] ← z
13                        si no a la derecha [ y ] ← z
14 izquierda [ z ] ← cero [ T ]
15 derecha [ z ] ← cero [ T ]
16 colores [ z ] ← ROJO
17 RB-INSERT-FIXUP ( T , z )
```

Hay cuatro diferencias entre los procedimientos TREE-INSERT y RB-INSERT. Primeramente las instancias de NIL en TREE-INSERT se reemplazan por *nil* [T]. En segundo lugar, establecemos *izquierda* [z] y *derecha* [z] a *cero* [T] en las líneas 14-15 de RB-INSERT, para mantener la estructura de árbol adecuada. Tercero, coloreamos z rojo en la línea 16. Cuarto, porque colorear z rojo puede causar una violación de uno de los propiedades rojo-negro, llamamos RB-INSERT-FIXUP (T , z) en la línea 17 de RB-INSERT para restaurar las propiedades rojo-negro.

```
RB-INSERT-FIXUP ( T , z )
 1 mientras que el color [ p [ z ]] = ROJO
 2 hacer si p [ z ] = izquierda [ p [ p [ z ]]]
 3                 luego y ← derecha [ p [ p [ z ]]]
 4                       si color [ y ] = ROJO
 5                             luego color [ p [ z ]] ← NEGRO                ▸ Caso 1
 6                                   color [ y ] ← NEGRO                     ▸ Caso 1
 7                                   color [ p [ p [ z ]]] ← ROJO            ▸ Caso 1
 8                                   z ← p [ p [ z ]]                        ▸ Caso 1
 9                             si no, si z = derecha [ p [ z ]]
10                                         luego z ← p [ z ]                 ▸ Caso 2
11                                               ROTACIÓN IZQUIERDA ( T , z ) ▸ Caso 2
```

Página 245

```
12                                         color [ p [ z ]] ← NEGRO          ▸ Caso 3
13                                         color [ p [ p [ z ]]] ← ROJO       ▸ Caso 3
14                                         GIRAR A LA DERECHA ( T , p [ p [ z ]]) ▸ Caso 3
15                 else (igual que la cláusula then
                              con "derecha" e "izquierda" intercambiados)
16 colores [ raíz [ T ]] ← NEGRO
```

Para entender cómo funciona RB-INSERT-FIXUP, romperemos nuestro examen del código en tres pasos principales. Primero, determinaremos qué violaciones de las propiedades rojo-negro son introducido en RB-INSERT cuando el nodo z se inserta y se colorea en rojo. Segundo, haremos examinar el objetivo general del **tiempo** de bucle en las líneas 1-15. Finalmente, exploraremos cada uno de los tres casos [1] en los **que** se rompe el ciclo while y ver cómo logran el objetivo.
La figura 13.4 muestra cómo funciona RB-INSERT-FIXUP en un árbol rojo-negro de muestra.

Figura 13.4: El funcionamiento de RB-INSERT-FIXUP. *(a)* Un nodo z después de la inserción. Dado que z y su padre p [z] son ambos rojos, se produce una violación de la propiedad 4. Dado que el tío *y de z* es rojo, el caso 1 en el código se puede aplicar. Los nodos se vuelven a colorear y el puntero z se mueve hacia arriba en el árbol, lo que resulta en el árbol que se muestra en *(b)* . Una vez más, z y su padre son ambos rojos, pero el tío *y de z* es negro. Dado que z es el hijo derecho de p [z], se puede aplicar el caso 2. Se realiza una rotación a la izquierda y el árbol resultante se muestra en *(c)* . Ahora z es el hijo izquierdo de su padre y se puede aplicar el caso 3. Una rotación a la derecha produce el árbol en *(d)* , que es un árbol rojo-negro legal.

¿Cuál de las propiedades rojo-negro se puede violar al llamar a RB-INSERT-FIXUP?
La propiedad 1 ciertamente sigue siendo válida, al igual que la propiedad 3, ya que ambos hijos del recién nacido Los ganglios rojos insertados son el centinela *nulo* [T]. Propiedad 5, que dice que el número de negros nodos es el mismo en todas las rutas desde un nodo dado, también se satisface, porque el nodo z reemplaza al centinela (negro), y el nodo z es rojo con los niños centinela. Así, el único Las propiedades que pueden violarse son la propiedad 2, que requiere que la raíz sea negra, y

Página 246

propiedad 4, que dice que un nodo rojo no puede tener un hijo rojo. Ambas posibles violaciones son debido a que z es de color rojo. La propiedad 2 se viola si z es la raíz, y la propiedad 4 se viola si El padre de z es rojo. La figura 13.4 (a) muestra una violación de la propiedad 4 después de que el nodo z ha sido insertado.

El **tiempo** de bucle en líneas 1-15 mantiene la siguiente de tres partes invariantes:

Al comienzo de cada iteración del ciclo,

a. El nodo z es rojo.
si. Si p [z] es la raíz, entonces p [z] es negro.
C. Si hay una infracción de las propiedades rojo-negro, hay como máximo una infracción, y es de la propiedad 2 o de la propiedad 4. Si hay una violación de la propiedad 2, ocurre porque z es la raíz y es rojo. Si hay una infracción de la propiedad 4, se produce porque tanto z como p [z] son rojos.

La parte (c), que trata de las violaciones de las propiedades rojo-negro, es más central para mostrar que RB-INSERT-FIXUP restaura las propiedades rojo-negro que las partes (a) y (b), que usamos en el camino para comprender situaciones en el código. Debido a que nos centraremos en el nodo z y nodos cercanos a él en el árbol, es útil saber del inciso a) que z es rojo. Usaremos la parte (b) para mostrar que el nodo p [p [z]] existe cuando lo referenciamos en las líneas 2, 3, 7, 8, 13 y 14.

Recuerde que necesitamos mostrar que un invariante de ciclo es verdadero antes de la primera iteración del ciclo, que cada iteración mantiene el ciclo invariante, y que el ciclo invariante nos da un útil propiedad en la terminación del bucle.

Comenzamos con los argumentos de inicialización y terminación. Luego, mientras examinamos cómo el cuerpo del ciclo funciona con más detalle, argumentaremos que el ciclo mantiene la invariante en cada iteración. En el camino, también demostraremos que hay dos posibles resultados de cada iteración del bucle: el puntero z *se* mueve hacia arriba en el árbol, o se realizan algunas rotaciones y el bucle termina.

- **Inicialización:** antes de la primera iteración del ciclo, comenzamos con un árbol rojo-negro sin violaciones, y agregamos un nodo rojo z . Mostramos que cada parte de la invariante se mantiene en el momento en que se llama a RB-INSERT-FIXUP:
    a. Cuando se llama a RB-INSERT-FIXUP, z es el nodo rojo que se agregó.

si. Si $p[z]$ es la raíz, entonces $p[z]$ comenzó en negro y no cambió antes de la llamada de RB-INSERT-FIXUP.

C. Ya hemos visto que las propiedades 1, 3 y 5 se mantienen cuando RB-INSERT- Se llama FIXUP.

Si hay una violación de la propiedad 2, entonces la raíz roja debe ser la recién agregada. nodo z , que es el único nodo interno en el árbol. Porque el padre y ambos los hijos de z son el centinela, que es negro, tampoco hay una violación de propiedad 4. Por lo tanto, esta violación de la propiedad 2 es la única violación de rojo-negro propiedades en todo el árbol.

Si hay una violación de la propiedad 4, entonces debido a que los hijos del nodo z son centinelas negros y el árbol no tuvieron otras violaciones antes de que se agregara z , el

Página 247

La violación debe ser porque tanto z como $p[z]$ son rojos. Además, no hay otros violaciones de las propiedades rojo-negro.

- **Terminación:** cuando el bucle termina, lo hace porque $p[z]$ es negro. (Si z es el raíz, entonces $p[z]$ es el centinela $nil[T]$, que es negro.) Por lo tanto, no hay violación de propiedad 4 en la terminación del bucle. Por el ciclo invariante, la única propiedad que puede fallar mantener es propiedad 2. La línea 16 también restaura esta propiedad, de modo que cuando RB-INSERT-FIXUP termina, todas las propiedades rojo-negro se mantienen.
- **Mantenimiento:** En realidad, hay seis casos a considerar en el **tiempo** de bucle, pero tres de ellos son simétricos a los otros tres, dependiendo de si z 's padre $p[z]$ es una izquierda hijo o hijo derecho del abuelo de z $p[p[z]]$, que se determina en la línea 2. Tenemos dado el código sólo para la situación en la que $p[z]$ es un hijo izquierdo. El nodo $p[p[z]]$ existe, ya que en la parte (b) del ciclo invariante, si $p[z]$ es la raíz, entonces $p[z]$ es negro. Como entramos en una iteración de bucle solo si $p[z]$ es rojo, sabemos que $p[z]$ no puede ser la raíz. Por tanto, existe $p[p[z]]$.

  El caso 1 se distingue de los casos 2 y 3 por el color del hermano del padre de z , o "tío." La línea 3 hace que y apunte a la *derecha del* tío de z $[p[p[z]]]$, y se realiza una prueba en la línea 4. Si y es rojo, entonces se ejecuta el caso 1. De lo contrario, el control pasa a los casos 2 y 3. En los tres casos, z 's abuelo $p[p[z]]$ es negro, ya que su padre $P[z]$ es rojo, y es propiedad de 4 violado sólo entre z y $p[z]$.

Caso 1: el tío y de z es rojo

La figura 13.5 muestra la situación para el caso 1 (líneas 5-8). El caso 1 se ejecuta cuando tanto $p[z]$ como y son rojos. Como $p[p[z]]$ es negro, podemos colorear tanto $p[z]$ como y negro, solucionando así el problema de Z y $P[z]$ tanto siendo rojo, y el color $p[p[z]]$ Rojo, manteniendo de esta manera la propiedad 5. A continuación, repetir el **mientras** bucle con $p[p[z]]$ como el nuevo nodo z . El puntero z sube dos niveles en el árbol.

Figura 13.5: Caso 1 del procedimiento RB-INSERT. La propiedad 4 se viola, ya que z y su padre $p[z]$ son ambos rojos. Se toma la misma acción si *(a)* z es un hijo derecho o *(b)* z es un hijo izquierdo niño. Cada uno de los subárboles α, β, γ, δ y ε tiene una raíz negra, y cada uno tiene la misma raíz negra. altura. El código para el caso 1 cambia los colores de algunos nodos, conservando la propiedad 5: todos Los caminos descendentes de un nodo a una hoja tienen el mismo número de negros. El *tiempo* de bucle continúa con el abuelo $p[p[z]]$ del nodo z como el nuevo z . Cualquier violación de la propiedad 4 ahora puede ocurren solo entre la nueva z , que es roja, y su padre, si también es roja.

Ahora mostramos que el caso 1 mantiene el ciclo invariante al comienzo de la siguiente iteración. Usamos z para denotar el nodo z en la iteración actual, y z' $p[p[z]]$ para denotar el nodo z en la prueba en línea 1 en la siguiente iteración.

Página 248

a. Debido a que esta iteración colorea *p* [*p* [*z*]] rojo, el nodo *z ′* es rojo al comienzo de la siguiente iteración.
si. El nodo *p* [*z ′*] es *p* [*p* [*p* [*z*]]] en esta iteración, y el color de este nodo no cambia.
Si este nodo es la raíz, era negro antes de esta iteración y permanece negro en el inicio de la siguiente iteración.
C. Ya hemos argumentado que el caso 1 mantiene la propiedad 5, y claramente no introducir una violación de las propiedades 1 o 3.

Si el nodo *z ′* es la raíz al comienzo de la siguiente iteración, entonces el caso 1 corrigió el único violación de la propiedad 4 en esta iteración. Dado que *z ′* es rojo y es la raíz, propiedad 2 se convierte en el único que se viola, y esta violación se debe a *z ′*.

Si el nodo *z ′* no es la raíz al comienzo de la siguiente iteración, entonces el caso 1 no ha creado una violación de la propiedad 2. El caso 1 corrigió la única violación de la propiedad 4 que existía en el inicio de esta iteración. Luego hizo *z ′* rojo y dejó *p* [*z ′*] solo. Si *p* [*z ′*] era negro, no hay violación de la propiedad 4. Si *p* [*z ′*] era rojo, colorear *z ′* rojo creaba uno violación de la propiedad 4 entre *z ′ yp* [*z ′*].

Caso 2: el tío y de z es negro y z es un niño adecuado

Caso 3: el tío y de z es negro y z es un hijo izquierdo

En los casos 2 y 3, el color del tío *y* de *z* es negro. Los dos casos se distinguen por si *z* es un hijo derecho o izquierdo de *p* [*z*]. Las líneas 10-11 constituyen el caso 2, que se muestra en la figura 13.6 junto con el caso 3. En el caso 2, el nodo *z* es un hijo derecho de su padre. Inmediatamente usamos una izquierda rotación para transformar la situación en el caso 3 (líneas 12-14), en el que el nodo *z* es un hijo izquierdo. Debido a que tanto *z* como *p* [*z*] son rojos, la rotación no afecta ni la altura del negro de los nodos ni propiedad 5. Ya sea que ingresemos al caso 3 directamente oa través del caso 2, el tío *y de z* es negro, ya que de lo contrario, habríamos ejecutado el caso 1. Además, el nodo *p* [*p* [*z*]] existe, ya que tenemos argumentó que este nodo existía en el momento en que se ejecutaron las líneas 2 y 3, y después de mover *z* arriba un nivel en la línea 10 y luego un nivel abajo en la línea 11, la identidad de *p* [*p* [*z*]] permanece sin alterar. En el caso 3, ejecutamos algunos cambios de color y una rotación a la derecha, que conservan propiedad 5, y luego, dado que ya no tenemos dos nodos rojos seguidos, hemos terminado. El cuerpo del **tiempo** de bucle no se ejecuta otra vez, ya que *p* [*z*] es ahora negro.

Figura 13.6: Casos 2 y 3 del procedimiento RB-INSERT. Como en el caso 1, se viola la propiedad 4 en el caso 2 o en el caso 3 porque *z* y su padre *p* [*z*] son ambos rojos. Cada uno de los subárboles α, β, γ, y δ tiene una raíz negra (α, β y γ de la propiedad 4, y δ porque de lo contrario estaríamos en caso 1), y cada uno tiene la misma altura de negro. El caso 2 se transforma en el caso 3 por una izquierda rotación, que conserva la propiedad 5: todos los caminos descendentes de un nodo a una hoja tienen el mismo número de negros. El caso 3 provoca algunos cambios de color y una rotación a la derecha, que también preservan propiedad 5. El *mientras* bucle termina entonces, porque la propiedad 4 se satisface: ya no son dos nodos rojos seguidos.

Ahora mostramos que los casos 2 y 3 mantienen el ciclo invariante. (Como acabamos de argumentar, *p* [*z*] será negro en la próxima prueba en la línea 1, y el cuerpo del bucle no se ejecutará nuevamente).

Página 249

a. El caso 2 hace que *z* apunte *ap* [*z*], que es rojo. No se produce ningún cambio adicional en *zo* su color en casos 2 y 3.
si. El caso 3 hace que *p* [*z*] sea negro, de modo que si *p* [*z*] es la raíz al comienzo de la siguiente iteración, es negro.

C. Como en el caso 1, las propiedades 1, 3 y 5 se mantienen en los casos 2 y 3.

Dado que el nodo z no es la raíz en los casos 2 y 3, sabemos que no hay violación de propiedad 2. Los casos 2 y 3 no introducen una violación de la propiedad 2, ya que el único El nodo que se hace rojo se convierte en hijo de un nodo negro por la rotación en el caso 3.

Los casos 2 y 3 corrigen la única violación de la propiedad 4, y no introducen otra violación.

Habiendo demostrado que cada iteración del ciclo mantiene el invariante, hemos demostrado que RB-INSERT-FIXUP restaura correctamente las propiedades rojo-negro.

Análisis

¿Cuál es el tiempo de ejecución de RB-INSERT? Dado que la altura de un árbol rojo-negro en n nodos es O (lg n), las líneas 1 a 16 de RB-INSERT toman O (lg n) tiempo. En RB-Insert- Fixup, el **tiempo** de bucle se repite solo si se ejecuta el caso 1, y luego el puntero z mueve dos niveles hacia arriba en el árbol. los Por tanto, el número total de veces **que** se puede ejecutar el ciclo while es O (lg n). Por lo tanto, RB-INSERT toma un total de O (lg n) tiempo. Curiosamente, nunca realiza más de dos rotaciones, ya que el **El** ciclo while termina si se ejecuta el caso 2 o el caso 3.

Ejercicios 13.3-1

En la línea 16 de RB-INSERT, establecemos el color del nodo z recién insertado en rojo. Note que si habíamos elegido establecer el color de z en negro, entonces la propiedad 4 de un árbol rojo-negro no sería violado. ¿Por qué no elegimos establecer el color de z en negro?

Ejercicios 13.3-2

Muestra los árboles rojo-negro que resultan tras insertar sucesivamente las teclas 41, 38, 31, 12, 19, 8 en un árbol rojo-negro inicialmente vacío.

Ejercicios 13.3-3

Suponga que la altura de negro de cada uno de los subárboles α, β, γ, δ , ε en las Figuras 13.5 y 13.6 es k . Etiquete cada nodo en cada figura con su altura en negro para verificar que la propiedad 5 se conserva mediante la transformación indicada.

Página 250

Ejercicios 13.3-4

Al profesor Teach le preocupa que RB-INSERT-FIXUP pueda establecer el *color* [*nil* [T]] en ROJO, en cuyo caso la prueba en la línea 1 no haría que el ciclo terminara cuando z es la raíz. mostrar que la preocupación del profesor es infundada al argumentar que RB-INSERT-FIXUP nunca establece *color* [*nulo* [T]] a ROJO.

Ejercicios 13.3-5

Considere un árbol rojo-negro formado insertando n nodos con RB-INSERT. Argumenta que si $n > 1$, el árbol tiene al menos un nodo rojo.

Ejercicios 13.3-6

Sugerir cómo implementar RB-INSERT de manera eficiente si la representación de árboles rojo-negro no incluye espacio de almacenamiento para los indicadores de los padres.

[1] El caso 2 cae dentro del caso 3, por lo que estos dos casos no son mutuamente excluyentes.

13.4 Eliminación

Al igual que las otras operaciones básicas en un n -node rojo-negro árbol, la supresión de un nodo de toma tiempo O (lg n). Eliminar un nodo de un árbol rojo-negro es solo un poco más complicado que insertar un nodo.

El procedimiento RB-DELETE es una modificación menor del procedimiento TREE-DELETE (Sección 12.3). Después de empalmar un nodo, llama a un procedimiento auxiliar RB-DELETE-FIXUP que cambia de color y realiza rotaciones para restaurar las propiedades rojo-negro.

```
BORRAR RB ( T , z )
 1 si izquierda [ z ] = cero [ T ] o derecha [ z ] = cero [ T ]
 2 luego y ← z
 3 else y ← ÁRBOL-SUCESOR ( z )
 4 si se deja [ y ] ≠ cero [ T ]
 5 luego x ← izquierda [ y ]
 6 más x ← derecha [ y ]
 7 p [ x ] ← p [ y ]
 8 si p [ y ] = cero [ T ]
 9 luego raíz [ T ] ← x
10 más si y = izquierda [ p [ y ]]
11                    luego a la izquierda [ p [ y ]] ← x
```

Página 251

```
12                    si no a la derecha [ p [ y ]] ← x
13 si y 3 ≠ z
14 luego tecla [ z ] ← tecla [ y ]
15                 copiar Y los datos de satélite 's en z
16 si color [ y ] = NEGRO
17 luego RB-DELETE-FIXUP ( T , x )
18 regreso y
```

Hay tres diferencias entre los procedimientos TREE-DELETE y RB-DELETE. Primero, todas las referencias a NIL en TREE-DELETE se reemplazan por referencias al centinela *nil* [T] en RB-DELETE. En segundo lugar, se elimina la prueba de si x es NIL en la línea 7 de TREE-DELETE, y la asignación p [x] ← p [y] se realiza incondicionalmente en la línea 7 de RB-DELETE. Así, si x es el centinela *nulo* [T], su puntero padre apunta al padre del nodo empalmado y . En tercer lugar, se realiza una llamada a RB-DELETE-FIXUP en las líneas 16-17 si y es negro. Si y es rojo, el rojo propiedades negro todavía mantienen cuando y se empalma a cabo, por las siguientes razones:

- no han cambiado las alturas negras del árbol,
- no se han hecho nodos rojos adyacentes, y
- dado que y no podría haber sido la raíz si fuera roja, la raíz permanece negra.

El nodo x pasa a RB-DELETE-fixup es uno de los dos nodos: o bien el nodo que era Y 's único hijo antes y fue empalmado a cabo si y tenían un hijo que no era el centinela *nula* [T], o, si y tenían sin hijos, x es el centinela *nulo* [T]. En este último caso, la cesión incondicional en la línea 7 garantiza que x 's de los padres es ahora el nodo que estaba previamente y ' s padres, si x es un número- que lleva el nódulo interno o el centinela *nulo* [T].

Ahora podemos examinar cómo el procedimiento RB-DELETE-FIXUP restaura el rojo-negro propiedades al árbol de búsqueda.

```
RB-BORRAR-FIXUP ( T , x )
 1 mientras x ≠ raíz [ T ] y color [ x ] = NEGRO
 2 hacer si x = izquierda [ p [ x ]]
 3                    luego w ← derecha [ p [ x ]]
 4                          si color [ w ] = ROJO
 5                                luego color [ w ] ← NEGRO                ▸ Caso 1
                                                                           ▸ Caso 1
```

```
6                    color [ p [ x ]] ← ROJO
7                    ROTACIÓN IZQUIERDA ( T , p [ x ])                ▸ Caso 1
8                    w ← derecha [ p [ x ]]                           ▸ Caso 1
9                if color [ izquierda [ w ]] = NEGRO y color [ derecha [ w ]] = NEGRO
10                   luego colorea [ w ] ← ROJO                       ▸ Caso 2
11                   xp [ x ]                                         ▸ Caso 2
12               de lo contrario, si color [ right [ w ]] = NEGRO
13                       luego color [ izquierda [ w ]] ← NEGRO       ▸ Caso 3
14                           color [ w ] ← ROJO                       ▸ Caso 3
15                           GIRAR A LA DERECHA ( T , w )             ▸ Caso 3
dieciséis                    w ← derecha [ p [ x ]]                   ▸ Caso 3
17               color [ w ] ← color [ p [ x ]]                       ▸ Caso 4
18               color [ p [ x ]] ← NEGRO                             ▸ Caso 4
19               color [ derecha [ w ]] ← NEGRO                       ▸ Caso 4
20               ROTACIÓN IZQUIERDA ( T , p [ x ])                    ▸ Caso 4
21               x ← raíz [ T ]                                       ▸ Caso 4
```

Página 252

```
22           else (lo mismo que la cláusula then con "derecha" e "izquierda" intercambiadas)
23 colores [ x ] ← NEGRO
```

Si el nodo *y* empalmado en RB-DELETE es negro, pueden surgir tres problemas. En primer lugar, si *y* tenía sido la raíz y un hijo rojo de *y se* convierte en la nueva raíz, hemos violado la propiedad 2. En segundo lugar, si tanto *x* como *p* [*y*] (que ahora también es *p* [*x*]) fueran rojos, entonces hemos violado la propiedad 4. En tercer lugar, *y* eliminación 's hace que cualquier camino que previamente contenida *y* tener uno menos nodo negro. Así, la propiedad 5 ahora es violada por cualquier ancestro de *y* en el árbol. Podemos corregir este problema diciendo que el nodo *x* tiene un negro "extra". Es decir, si sumamos 1 al recuento de nodos negros en cualquier ruta que contiene *x* , entonces, bajo esta interpretación, se cumple la propiedad 5. Cuando empalmamos el nodo negro *y* , "empujamos" su negrura sobre su hijo. El problema es que ahora el nodo *x* no es ni rojo ni negro, violando así la propiedad 1. En cambio, el nodo *x* es "doblemente negro" o "rojo- y negro ", y contribuye con 2 o 1, respectivamente, al recuento de nodos negros en las rutas que contiene *x* . El atributo de *color* de *x* seguirá siendo ROJO (si *x* es rojo y negro) o NEGRO (si *x* es doblemente negro). En otras palabras, el negro extra en un nodo se refleja en *x* 's apuntando al nodo en lugar de al atributo de *color* .

El procedimiento RB-DELETE-FIXUP restaura las propiedades 1, 2 y 4. Ejercicios 13.4-1 y 13.4-2 le piden que demuestre que el procedimiento restaura las propiedades 2 y 4, y así en el resto de esta sección, nos centraremos en la propiedad 1. El objetivo del **tiempo** de bucle en las líneas 1-22 consiste en Mueva el negro adicional arriba del árbol hasta

1. *x* apunta a un nodo rojo y negro, en cuyo caso coloreamos *x* (individualmente) negro en la línea 23,
2. *x* apunta a la raíz, en cuyo caso el negro adicional puede simplemente "eliminarse", o
3. Se pueden realizar rotaciones y cambios de color adecuados.

Dentro del **tiempo** de bucle, *x* siempre apunta a un nonroot doblemente nodo negro. Determinamos en linea 2 si *x* es un hijo izquierdo o un hijo derecho de su padre *p* [*x*]. (Hemos dado el código para el situación en la que *x* es un hijo izquierdo; la situación en la que *x* es un hijo correcto (línea 22) es simétrico.) Mantenemos un puntero *w* al hermano de *x* . Dado que el nodo *x* es doblemente negro, el nodo *w* no puede ser *nulo* [*T*]; de lo contrario, el número de negros en el camino desde *p* [*x*] hasta el (solo negro) la hoja *w* sería menor que el número en la ruta de *p* [*x*] a *x* .

Los cuatro casos [2] del código se ilustran en la Figura 13.7 . Antes de examinar cada caso en detalle, veamos de manera más general cómo podemos verificar que la transformación en cada uno de los casos conserva la propiedad 5. La idea clave es que en cada caso el número de nodos negros (incluyendo el negro extra de *x*) desde (e incluyendo) la raíz del subárbol que se muestra en cada uno de los subárboles α, β, ..., ζ se conserva mediante la transformación. Por tanto, si la propiedad 5 se mantiene antes de la transformación, sigue manteniéndose después. Por ejemplo, en la Figura 13.7 (a) , que ilustra el caso 1, el número de nodos negros desde la raíz hasta el subárbol α o β es 3, ambos antes y después de la transformación. (Nuevamente, recuerde que el nodo *x* agrega un negro adicional). De manera similar, el número de nodos negros desde la raíz hasta cualquiera de γ, δ, ε, y es ζ, tanto antes como después de la transformación. En la Figura 13.7 (b) , el conteo debe involucrar el valor *c* del *color* atributo de la raíz del subárbol mostrado, que puede ser ROJO o NEGRO. Si definimos count (RED) = 0 y count (BLACK) = 1, entonces el número de nodos negros desde la raíz hasta α es 2 + cuenta (*c*), tanto antes como después de la transformación. En este caso, después de la transformación, el nuevo nodo *x* tiene el atributo de *color c* , pero este nodo es realmente rojo y negro (si *c* = RED) o doblemente negro (si *c* = NEGRO). Los otros casos se pueden verificar de manera similar (ver Ejercicio 13.4-5).

Página 253

Figura 13.7: Los casos en el *tiempo* de bucle del procedimiento RB-DELETE-fixup. Oscurecido los nodos tienen atributos de *color* NEGRO, los nodos muy sombreados tienen atributos de *color* ROJO y nodos ligeramente sombreadas tienen *de color* atributos representados por c y c', que puede ser o bien RED o NEGRO. Las letras α, β, ..., ζ representan subárboles arbitrarios. En cada caso, la configuración a la izquierda se transforma en la configuración de la derecha cambiando algunos colores y / o realizando una rotación. Cualquier nodo señalado por x tiene un negro adicional y es doblemente negro o rojo y negro. El único caso que hace que el bucle se repita es el caso 2. *(a) El* caso 1 es transformado al caso 2, 3 o 4 intercambiando los colores de los nodos B y D y realizando una rotación a la izquierda. *(b)* En el caso 2, el negro adicional representado por el puntero x se mueve hacia arriba en el árbol por colorear nodo D rojo y el establecimiento de x a punto al nodo B. Si ingresamos del caso 2 al caso 1, el *mientras* termina de bucle porque el nuevo nodo x es de color rojo y negro, y por lo tanto el valor c de su *El* atributo de *color* es ROJO. *(c) El* caso 3 se transforma en el caso 4 intercambiando los colores de los nodos C y D y realizando una rotación a la derecha. *(d)* En el caso 4, el negro extra representado por x puede ser eliminado cambiando algunos colores y realizando una rotación a la izquierda (sin violar el rojo-propiedades negras), y el bucle termina.

Caso 1: el hermano w de x es rojo

El caso 1 (líneas 5-8 de RB-DELETE-FIXUP y Figura 13.7 (a)) ocurre cuando el nodo w, el hermano del nodo x, es rojo. Como w debe tener hijos negros, podemos cambiar los colores de w y $p[x]$ y luego realice una rotación a la izquierda en $p[x]$ sin violar ninguna de las propiedades rojo-negro. El nuevo hermano de x, que es uno de los hijos de w antes de la rotación, ahora es negro y, por lo tanto, hemos convertido el caso 1 en el caso 2, 3 o 4.

Los casos 2, 3 y 4 ocurren cuando el nodo w es negro; que se distinguen por los colores de W's niños.

Caso 2: el hermano w de x es negro y los dos hijos de w son negros

En el caso 2 (líneas 10-11 de RB-DELETE-FIXUP y Figura 13.7 (b)), los dos hijos de w son negro. Como w también es negro, quitamos un negro tanto de x como de w, dejando x con solo un negro y dejando w rojo. Para compensar la eliminación de un negro de x y w, nos gustaría añadir un negro adicional $ap[x]$, que originalmente era rojo o negro. Lo hacemos repitiendo el

Página 254

while bucle con $p[x]$ como el nuevo nodo x. Observe que si ingresamos del caso 2 al caso 1, el el nuevo nodo x es rojo y negro, ya que el $p[x]$ original era rojo. Por tanto, el valor c del *color*

El atributo del nuevo nodo *x* es ROJO, y el bucle termina cuando prueba la condición del bucle.
El nuevo nodo *x* se colorea (individualmente) de negro en la línea 23.

Caso 3: el hermano w de x es negro, el hijo izquierdo de w es rojo y el hijo derecho de w es negro

El caso 3 (líneas 13-16 y Figura 13.7 (c)) ocurre cuando *w* es negro, su hijo izquierdo es rojo y su el niño correcto es negro. Podemos cambiar los colores de *w* y su hijo *izquierdo dejó* [*w*] y luego realizar un rotación derecha en *w* sin violar ninguna de las propiedades rojo-negro. El nuevo hermano *w* de *x* es ahora un nodo negro con un hijo derecho rojo, y así hemos transformado el caso 3 en el caso 4.

Caso 4: el hermano w de x es negro y el hijo derecho de w es rojo

El caso 4 (líneas 17-21 y figura 13.7 (d)) ocurre cuando el hermano *w del* nodo *x* es negro y *w* el derecho el niño es rojo. Al hacer algunos cambios de color y realizar una rotación a la izquierda en *p* [*x*], podemos quitar el negro extra en *x* , haciéndolo solo negro, sin violar ninguno de los rojo-negro propiedades. Ajuste *x* sea la raíz hace que el **tiempo** de bucle de interrumpir cuando se pone a prueba el bucle condición.

Análisis

¿Cuál es el tiempo de ejecución de RB-DELETE? Dado que la altura de un árbol rojo-negro de *n* nodos es *O* (lg *n*), el costo total del procedimiento sin la llamada a RB-DELETE-FIXUP toma *O* (lg *n*) hora. Dentro de RB-DELETE-FIXUP, los casos 1, 3 y 4 terminan cada uno después de realizar una número constante de cambios de color y como máximo tres rotaciones. El caso 2 es el único caso en el que el **tiempo** de bucle se puede repetir, y luego el puntero *x* se mueve hacia arriba el árbol en la mayoría de *O* (lg *n* veces) y no se realizan rotaciones. Por lo tanto, el procedimiento RB-DELETE-FIXUP toma *O* (lg *n*) tiempo y realiza como máximo tres rotaciones, por lo que el tiempo total para RB-DELETE también es *O* (lg *n*).

Ejercicios 13.4-1

Argumenta que después de ejecutar RB-DELETE-FIXUP, la raíz del árbol debe ser negra.

Ejercicios 13.4-2

Argumenta que si en RB-DELETE tanto *x* como *p* [*y*] son rojos, entonces la propiedad 4 es restaurada por la llamada RB-BORRAR-FIXUP (*T* , *x*).

Ejercicios 13.4-3

Página 255

En el ejercicio 13.3-2 , encontró el árbol rojo-negro que resulta de insertar sucesivamente el claves 41, 38, 31, 12, 19, 8 en un árbol inicialmente vacío. Ahora muestre los árboles rojo-negros que resultan a partir del borrado sucesivo de las claves en el orden 8, 12, 19, 31, 38, 41.

Ejercicios 13.4-4

¿En qué líneas del código para RB-DELETE-FIXUP podríamos examinar o modificar el centinela? *nada* [*T*]?

Ejercicios 13.4-5

En cada uno de los casos de la figura 13.7 , proporcione el recuento de nodos negros de la raíz del subárbol mostrado a cada uno de los subárboles α, β , ..., ζ, y verifique que cada recuento permanece igual después de la transformación. Cuando un nodo tiene un atributo de *color* c o c', use la notación count (c) o cuente (c') simbólicamente en su cuenta.

Ejercicios 13.4-6

A los profesores Skelton y Baron les preocupa que al comienzo del caso 1 de RB-DELETE-FIXUP, el nodo p [x] podría no ser negro. Si los profesores están en lo correcto, entonces las líneas 5-6 son incorrecto. Demuestre que p [x] debe ser negro al comienzo del caso 1, para que los profesores no tengan nada preocuparse de.

Ejercicios 13.4-7

Suponga que un nodo x se inserta en un árbol rojo-negro con RB-INSERT y luego inmediatamente eliminado con RB-DELETE. ¿Es el árbol rojo-negro resultante el mismo que el rojo-negro inicial? ¿árbol? Justifica tu respuesta.

Problemas 13-1: Conjuntos dinámicos persistentes

Durante el curso de un algoritmo, a veces encontramos que necesitamos mantener versiones pasadas de un conjunto dinámico a medida que se actualiza. Tal conjunto se llama **persistente** . Una forma de implementar un conjunto persistente es copiar el conjunto completo siempre que se modifica, pero este enfoque puede ralentizar un programa y también consumen mucho espacio. A veces, podemos hacerlo mucho mejor.

Página 256

Considere un conjunto persistente S con las operaciones INSERT, DELETE y SEARCH, que implementar utilizando árboles de búsqueda binarios como se muestra en la Figura 13.8 (a) . Mantenemos una raíz separada para cada versión del set. Para insertar la clave 5 en el conjunto, creamos un nuevo nodo con clave 5. Este nodo se convierte en el hijo izquierdo de un nuevo nodo con clave 7, ya que no podemos modificar el nodo existente con clave 7. De manera similar, el nuevo nodo con clave 7 se convierte en el hijo izquierdo de un nuevo nodo con clave 8 cuyo hijo derecho es el nodo existente con clave 10. El nuevo nodo con clave 8 se convierte, a su vez, en el hijo derecho de una nueva raíz r' con la clave 4 cuyo hijo izquierdo es el existente nodo con la clave 3. Por lo tanto, copiamos solo una parte del árbol y compartimos algunos de los nodos con el árbol original, como se muestra en la Figura 13.8 (b) .

Figura 13.8: *(a)* Un árbol de búsqueda binario con claves 2, 3, 4, 7, 8, 10. *(b)* El binario persistente árbol de búsqueda que resulta de la inserción de la clave 5. La versión más reciente del conjunto consiste de los nodos accesibles desde la raíz r' , y la versión anterior consta de los nodos accesible desde r . Los nodos muy sombreados se agregan cuando se inserta la clave 5.

Suponga que cada nodo del árbol tiene la *clave de* campos , *izquierda* y *derecha,* pero ningún campo principal. (Ver también Ejercicio 13.3-6 .)

a. Para un árbol de búsqueda binario persistente general, identifique los nodos que deben cambiarse para insertar una clave k o eliminar un nodo y .
si. Escriba un procedimiento PERSISTENT-TREE-INSERT que, dado un árbol persistente T y un clave k a insertar, devuelve un nuevo árbol persistente T 'que es el resultado de la inserción de k en T .
C. Si la altura del árbol de búsqueda binaria persistente T es h , ¿cuáles son el tiempo y el espacio?

requisitos de su implementación de PERSISTENT-TREE-INSERT? (El espacio
El requisito es proporcional al número de nuevos nodos asignados).

re. Supongamos que hemos incluido el campo padre en cada nodo. En este caso,
PERSISTENT-TREE-INSERT necesitaría realizar copias adicionales. Pruebalo
PERSISTENT-TREE-INSERT requeriría entonces $\Omega(n)$ tiempo y espacio, donde n es el
número de nodos en el árbol.

mi. Muestre cómo usar árboles rojo-negro para garantizar que el tiempo de ejecución y el
los espacios son $O(\lg n)$ por inserción o eliminación.

Problemas 13-2: operación de unión en árboles rojo-negro

La operación de **unión** toma dos conjuntos dinámicos S_1 y S_2 y un elemento x tal que para cualquier x_1 S_1 y x_2 S_2, tenemos la $tecla[x_1] \le tecla[x] \le tecla[x_2]$. Devuelve un conjunto $S = S_1 \cup \{x\} \cup S_2$. En esto problema, investigamos cómo implementar la operación de unión en árboles rojo-negro.

Página 257

a. Dado un árbol T rojo-negro , almacenamos su altura negra como el campo $bh[T]$. Argumenta que esto
El campo puede ser mantenido por RB-INSERT y RB-DELETE sin requerir extra
almacenamiento en los nodos del árbol y sin incrementar los tiempos de ejecución asintóticos.
Muestre que al descender por T, podemos determinar la altura de negro de cada
nodo que visitamos en $O(1)$ tiempo por nodo visitado.

Deseamos implementar la operación RB-JOIN (T_1, x, T_2), que destruye T_1 y T_2 y devuelve un árbol rojo-negro $T = T_1 \cup \{x\} \cup T_2$. Sea n el número total de nodos en T_1 y T_2.

si. Suponga que $bh[T_1] \ge bh[T_2]$. Describe un algoritmo de tiempo $O(\lg n)$ que encuentra un
nodo y en T_1 con la clave más grande de entre los nodos cuya altura de negro es
$bh[T_2]$.

C. Sea T_y el subárbol enraizado en y. Describe cómo $T_y \cup \{X\} \cup T_2$ puede sustituir a T_y en $O(1)$
tiempo sin destruir la propiedad binary-search-tree.

re. ¿De qué color deberíamos hacer x para que se mantengan las propiedades rojo-negro 1, 3 y 5?
Describe cómo las propiedades 2 y 4 se pueden aplicar en el tiempo $O(\lg n)$.

mi. Argumenta que no se pierde ninguna generalidad al hacer el supuesto del inciso b). Describe el
situación simétrica que surge cuando $bh[T_1] = bh[T_2]$.

F. Argumenta que el tiempo de ejecución de RB-JOIN es $O(\lg n)$.

Problemas 13-3: árboles AVL

Un **árbol AVL** es un árbol de búsqueda binario con **equilibrio de altura**: para cada nodo x, las alturas del los subárboles izquierdo y derecho de x difieren como máximo en 1. Para implementar un árbol AVL, mantenemos un campo adicional en cada nodo: $h[x]$ es la altura del nodo x. Como para cualquier otro árbol de búsqueda binario T, suponga que la $raíz[T]$ apunta al nodo raíz.

a. Demuestre que un árbol AVL con n nodos tiene una altura $O(\lg n)$. (*Sugerencia:* demuestre que en un AVL
árbol de altura h, hay al menos F_h nodos, donde F_h es el h- ésimo número de Fibonacci).

si. Para insertarlo en un árbol AVL, primero se coloca un nodo en el lugar apropiado en binario
orden de árbol de búsqueda. Después de esta inserción, es posible que el árbol ya no esté equilibrado en altura.
Específicamente, las alturas de los hijos izquierdo y derecho de algún nodo pueden diferir en 2.
Describe un procedimiento BALANCE (x), que toma un subárbol enraizado en x cuya izquierda y
los niños de la derecha tienen una altura equilibrada y tienen alturas que difieren como máximo en 2, es decir,
$|h[derecha[x]] - h[izquierda[x]]| \le 2$, y altera el subárbol enraizado en x para que esté equilibrado en altura.
(*Sugerencia:* use rotaciones).

C. Utilizando el inciso b), describa un procedimiento recursivo AVL-INSERT (x, z), que toma un nodo
x dentro de un árbol AVL y un nodo z recién creado (cuya clave ya se ha llenado
in), y agrega z al subárbol enraizado en x, manteniendo la propiedad de que x es la raíz de
un árbol AVL. Como en TREE-INSERT de la Sección 12.3 , suponga que la $tecla[z]$ ya
completado y que a la $izquierda[z]$ = NIL y a la $derecha[z]$ = NIL; también suponga que $h[z] = 0$.
Así, para insertar el nodo z en el árbol AVL T, llamamos AVL-INSERT ($root[T]$, z).

re. Dé un ejemplo de un árbol AVL de n nodos en el que una operación AVL-INSERT provoca

Ω (lg n) rotaciones a realizar.

Página 258

Problemas 13-4: Treaps

Si insertamos un conjunto de n elementos en un árbol de búsqueda binario, el árbol resultante puede ser horriblemente desequilibrado, lo que lleva a tiempos de búsqueda prolongados. Como vimos en la Sección 12.4 , sin embargo, construidos al azar Los árboles de búsqueda binaria tienden a estar equilibrados. Por tanto, una estrategia que, en promedio, construye árbol equilibrado para un conjunto fijo de elementos es permutar aleatoriamente los elementos y luego insertarlos en ese orden en el árbol.

¿Qué pasa si no tenemos todos los artículos a la vez? Si recibimos los artículos uno a la vez, ¿podemos todavía construye aleatoriamente un árbol de búsqueda binario a partir de ellos?

Examinaremos una estructura de datos que responde afirmativamente a esta pregunta. Un **treap** es un árbol de búsqueda binaria con una forma modificada de ordenar los nodos. La figura 13.9 muestra un ejemplo. Como es habitual, cada nodo x en el árbol tiene una clave de valor *clave* [x]. Además, asignamos *prioridad* [x], que es un número aleatorio elegido independientemente para cada nodo. Asumimos que todas las prioridades son distintas y también que todas las claves son distintas. Los nodos de la treap están ordenados de modo que el las claves obedecen a la propiedad binary-search-tree y las prioridades obedecen a la propiedad de orden min-heap:

- Si v es un hijo izquierdo de u , entonces la *tecla* [v] < *tecla* [u].
- Si v es un hijo derecho de u , entonces *tecla* [v]> *tecla* [u].
- Si v es un hijo de u , entonces *prioridad* [v]> *prioridad* [u].

Figura 13.9: Una golosina. Cada nodo x está etiquetado con la *tecla* [x]: *Prioridad* [x]. Por ejemplo, la raíz tiene clave G y prioridad 4.

(Esta combinación de propiedades es la razón por la que el árbol se llama "treap"; tiene características de un árbol de búsqueda binaria y un montón).

Es útil pensar en las golosinas de la siguiente manera. Supongamos que insertamos nodos x_1 , x_2 , ..., x_n , con claves asociadas, en un treap. Entonces el treap resultante es el árbol que se habría formado si los nodos se hubieran insertado en un árbol de búsqueda binario normal en el orden dado por su prioridades (elegidas al azar), es decir, *prioridad* [x_i] < *prioridad* [x_j] significa que x_i se insertó antes x_j .

a. Demuestre que dado un conjunto de nodos x_1 , x_2 , ..., x_n , con claves y prioridades asociadas (todas distinto), hay un tratamiento único asociado con estos nodos.
si. Demuestre que la altura esperada de un treap es Θ (lg n) y, por lo tanto, el tiempo para buscar un el valor en el treap es Θ (lg n).

Página 259

Veamos cómo insertar un nuevo nodo en un treap existente. Lo primero que hacemos es asignar a el nuevo nodo una prioridad aleatoria. Luego llamamos al algoritmo de inserción, al que llamamos TREAP-INSERT, cuyo funcionamiento se ilustra en la Figura 13.10 .

Figura 13.10: El funcionamiento de TREAP-INSERT. *(a)* El tratamiento original, antes de la inserción. *(si)* El tratamiento después de insertar un nodo con clave C y prioridad 25. *(c)* - *(d)* Etapas intermedias cuando insertar un nodo con clave D y prioridad 9. *(e)* El tratamiento después de la inserción de las partes (c) y (d) está hecho. *(f)* El treap después de insertar un nodo con clave F y prioridad 2.

C. Explica cómo funciona TREAP-INSERT. Explica la idea en inglés y da pseudocódigo. (*Sugerencia:* ejecute el procedimiento habitual de inserción de árbol de búsqueda binaria y luego realizar rotaciones para restaurar la propiedad de orden de pila mínima).

re. Muestre que el tiempo de ejecución esperado de TREAP-INSERT es $\Theta(\lg n)$.

TREAP-INSERT realiza una búsqueda y luego una secuencia de rotaciones. Aunque estos dos las operaciones tienen el mismo tiempo de ejecución esperado, tienen diferentes costos en la práctica. UNA La búsqueda lee la información del Treap sin modificarla. Por el contrario, una rotación cambia punteros para padres e hijos dentro del treap. En la mayoría de las computadoras, las operaciones de lectura son mucho más rápido que las operaciones de escritura. Por lo tanto, nos gustaría que TREAP-INSERT realizara pocas rotaciones. Demostraremos que el número esperado de rotaciones realizadas está acotado por una constante.

Para hacerlo, necesitaremos algunas definiciones, que se ilustran en la Figura 13.11 . La **izquierda columna vertebral** de un árbol de búsqueda binaria T es la ruta desde la raíz hasta el nodo con la clave más pequeña. En En otras palabras, la columna izquierda es el camino desde la raíz que consta solo de bordes izquierdos. Simétricamente, la **columna derecha** de T es el camino desde la raíz que consta solo de bordes rectos. La **longitud** de una columna es el número de nodos que contiene.

Figura 13.11: Espinas de un árbol de búsqueda binaria. La columna izquierda está sombreada en *(a)* y la columna derecha está sombreado en *(b)* .

mi. Considere el Treap T inmediatamente después de insertar x usando TREAP-INSERT. Deje que C sea la longitud de la columna derecha del subárbol izquierdo de x . Sea D la longitud de la columna izquierda del subárbol derecho de x . Demuestre que el número total de rotaciones que se realizaron durante la inserción de x es igual a $C + D$.

Ahora vamos a calcular los valores esperados de C y D . Sin pérdida de generalidad, asumimos

que las claves son 1, 2, ..., n , ya que las estamos comparando solo entre sí.

Para los nodos x y y , donde $y \neq x$, vamos $k = clave [x]$ y $i = tecla [Y]$. Definimos indicador aleatorio variables

$X_{i,k} = I \{ y$ está en la columna derecha del subárbol izquierdo de x (en $T)\}$.

F. Muestre que $X_{i,k} = 1$ si y solo si $prioridad [y] > prioridad [x]$, $tecla [y] < tecla [x]$, y, para cada z tal que $tecla [y] < tecla [z] < tecla [x]$, tenemos $prioridad [y] < prioridad [z]$.

gramo. Muestra esa

h. Muestra esa

yo. Utilice un argumento de simetría para demostrar que

j. Concluya que el número esperado de rotaciones realizadas al insertar un nodo en una pastilla es menos de 2.

[2] Como en RB-INSERT-FIXUP, los casos en RB-DELETE-FIXUP no son mutuamente excluyentes.

Notas del capítulo

Página 261

La idea de equilibrar un árbol de búsqueda se debe a Adel'son-Vel'ski i y Landis [2] , quienes introdujo una clase de árboles de búsqueda equilibrados llamados "árboles AVL" en 1962, que se describen en Problema 13-3 . JE Hopcroft introdujo otra clase de árboles de búsqueda, denominados "2-3 árboles". (inédito) en 1970. El equilibrio se mantiene en un árbol 2-3 manipulando los grados de nodos en el árbol. Una generalización de 2-3 árboles introducida por Bayer y McCreight [32] , llamados árboles B, es el tema del Capítulo 18 .

Los árboles rojo-negros fueron inventados por Bayer [31] bajo el nombre de "árboles B binarios simétricos". Guibas y Sedgewick [135] estudiaron sus propiedades en profundidad e introdujeron el rojo / negro convención de color. Andersson [15] da una variante más simple de codificar de árboles rojo-negro. Weiss [311] llama a esta variante árboles AA. Un árbol AA es similar a un árbol rojo-negro excepto que a la izquierda es posible que los niños nunca sean rojos.

Seidel y Aragon propusieron trampas [271] . Son la implementación predeterminada de un diccionario en LEDA, que es una colección bien implementada de estructuras de datos y algoritmos.

Hay muchas otras variaciones en árboles binarios equilibrados, incluidos árboles con equilibrio de peso [230] , k -árboles vecinos [213] y árboles de chivo expiatorio [108] . Quizás los más intrigantes son los "árboles de esparcimiento" introducidos por Sleator y Tarjan [281] , que son "autoajustables". (Un bien Tarjan [292] da una descripción de los árboles de extensión). Los árboles de extensión mantienen el equilibrio sin condición de equilibrio explícita como el color. En su lugar, "operaciones de extensión" (que implican rotaciones) se realizan dentro del árbol cada vez que se realiza un acceso. El costo amortizado (ver Capítulo 17) de cada operación en un árbol de n nodos es O (lg n).

Las listas de omisión [251] son una alternativa a los árboles binarios equilibrados. Una lista de omisión es una lista enlazada que se aumentado con una serie de punteros adicionales. Cada operación de diccionario se ejecuta en el esperado time O (lg n) en una lista de salto de n elementos.

Capítulo 14: Aumento de las estructuras de datos

Hay algunas situaciones de ingeniería que no requieren más que una estructura de datos de "libro de texto": como una lista doblemente enlazada, una tabla hash o un árbol de búsqueda binaria, pero muchos otros requieren una pizca de creatividad. Solo en raras situaciones necesitará crear un tipo de datos completamente nuevo estructura, sin embargo. Más a menudo, será suficiente aumentar la estructura de datos de un libro de texto almacenando información adicional en él. A continuación, puede programar nuevas operaciones para que la estructura de datos apoyar la aplicación deseada. Aumentar una estructura de datos no siempre es sencillo, sin embargo, dado que la información agregada debe ser actualizada y mantenida por el operaciones en la estructura de datos.

Este capítulo analiza dos estructuras de datos que se construyen aumentando árboles rojo-negro. La seccin 14.1 describe una estructura de datos que soporta operaciones estadsticas de orden general en un conjunto dinámico. Entonces podemos encontrar rápidamente el *i*- ésimo número más pequeño de un conjunto o el rango de un determinado elemento en el orden total del conjunto. La sección 14.2 abstrae el proceso de aumentar un estructura de datos y proporciona un teorema que puede simplificar el aumento de árboles rojo-negro. La sección 14.3 usa este teorema para ayudar a diseñar una estructura de datos para mantener un conjunto dinámico de intervalos, como intervalos de tiempo. Dado un intervalo de consulta, podemos encontrar rápidamente un intervalo en el conjunto que lo superpone.

Página 262

14.1 Estadísticas de pedidos dinámicos

El capítulo 9 introdujo la noción de una estadística de orden. Específicamente, la estadística de *i*- ésimo orden de un conjunto de *n* elementos, donde *i* {1, 2, ..., *n* }, es simplemente el elemento del conjunto con la *i*- ésima clave más pequeña. Vimos que cualquier estadística de orden se puede recuperar en *O* (*n*) tiempo de un conjunto desordenado. En esto sección, veremos cómo los árboles rojo-negro pueden modificarse para que cualquier estadística de orden pueda ser determinado en tiempo *O* (lg *n*). También veremos cómo el rango de un elemento, su posición en el orden lineal del conjunto: también se puede determinar en tiempo *O* (lg *n*).

En la figura 14.1 se muestra una estructura de datos que puede soportar operaciones rápidas de estadística de pedidos . Un El árbol de orden-estadística T es simplemente un árbol rojo-negro con información adicional almacenada en cada nodo. Además de la *clave de* campos de árbol rojo-negro habitual [*x*], *color* [*x*], *p* [*x*], *izquierda* [*x*] y *derecha* [*x*] en una nodo *x* , tenemos otro *tamaño de* campo [*x*]. Este campo contiene el número de nodos (internos) en el subárbol enraizado en *x* (incluido el propio *x*), es decir, el tamaño del subárbol. Si definimos el el tamaño del centinela en 0, es decir, establecemos el *tamaño* [*nil* [*T*]] en 0, luego tenemos la identidad

tamaño [*x*] = *tamaño* [*izquierda* [*x*]] + *tamaño* [*derecha* [*x*]] + 1.

Figura 14.1: Un árbol de estadísticas de orden, que es un árbol rojo-negro aumentado. Los nodos sombreados son los nodos rojos y oscuros son negros. Además de sus campos habituales, cada nodo *x* tiene un campo *tamaño* [*x*], que es el número de nodos en el subárbol con raíz en *x* .

No requerimos que las claves sean distintas en un árbol de estadísticas de orden. (Por ejemplo, el árbol de la Figura 14.1 tiene dos claves con valor 14 y dos claves con valor 21.) En presencia de claves iguales, la noción anterior de rango no está bien definida. Eliminamos esta ambigüedad para una estadística de orden árbol definiendo el rango de un elemento como la posición en la que se imprimiría en un en orden a pie del árbol. En la Figura 14.1 , por ejemplo, la clave 14 almacenada en un nodo negro tiene rango 5, y la clave 14 almacenada en un nodo rojo tiene rango 6.

Recuperar un elemento con un rango determinado

Antes de mostrar cómo mantener esta información de tamaño durante la inserción y eliminación, permítanos Examine la implementación de dos consultas de estadísticas de pedidos que utilizan esta información adicional. Comenzamos con una operación que recupera un elemento con un rango determinado. El procedimiento OS-SELECT (*x* , *i*) devuelve un puntero al nodo que contiene la *i*- ésima clave más pequeña en el subárbol arraigado en *x* . Para encontrar la *i*- ésima clave más pequeña en un árbol de orden-estadística *T* , llamamos OS-SELECCIONAR (*raíz* [*T*], *i*).

OS-SELECT (*x* , *i*)

```
1 r ← tamaño [ izquierda [ x ]] + 1
2 si yo = r
3 luego regresa x
4 elseif i < r
5 luego regrese OS-SELECT ( izquierda [ x ], i )
6 de lo contrario, devuelva OS-SELECT ( derecha [ x ], i - r )
```

Página 263

La idea detrás de OS-SELECT es similar a la de los algoritmos de selección del Capítulo 9 . los
El valor de *tamaño* [*izquierda* [*x*]] es el número de nodos que vienen antes de *x* en un recorrido de árbol en orden del subárbol enraizado en *x* . Por lo tanto, *size* [*left* [*x*]] + 1 es el rango de *x* dentro del subárbol enraizado en *x* .

En la línea 1 de OS-SELECT, calculamos *r* , el rango del nodo *x* dentro del subárbol enraizado en *x* . Si *yo* = *r* , entonces el nodo *x* es el *i*- ésimo elemento más pequeño, por lo que devolvemos *x* en la línea 3. Si *i* < *r* , entonces el *i*- ésimo el elemento más pequeño está en el subárbol izquierdo de *x* , así que recurrimos a la *izquierda* [*x*] en la línea 5. Si *i* > *r* , entonces el *i*- ésimo el elemento más pequeño está en el subárbol derecho de *x* . Dado que hay *r* elementos en el subárbol enraizados en *x* que vienen antes del subárbol derecho de *x* en una caminata de árbol en orden, el *i*- ésimo elemento más pequeño en el subárbol enraizado en *x* es el (*i* - *r*) ésimo elemento más pequeño en el subárbol enraizado en la *derecha* [*x*]. Esta El elemento se determina de forma recursiva en la línea 6.

Para ver cómo funciona OS-SELECT, considere una búsqueda del decimoséptimo elemento más pequeño en el árbol de orden-estadística de la Figura 14.1 . Comenzamos con *x* como raíz, cuya clave es 26, y con *i* = 17. Dado que el tamaño del subárbol izquierdo de 26 es 12, su rango es 13. Por lo tanto, sabemos que el nodo con el rango 17 es el 17 - 13 = cuarto elemento más pequeño en el subárbol derecho de 26. Después de la llamada recursiva, *x* es el nodo con la clave 41, *ei* = 4. Dado que el tamaño del subárbol izquierdo de 41 es 5, su rango dentro de su subárbol es 6. Por lo tanto, sabemos que el nodo con rango 4 es el cuarto elemento más pequeño en 41 a la izquierda subárbol. Después de la llamada recursiva, *x* es el nodo con la clave 30 y su rango dentro de su subárbol es 2. Por lo tanto, recurrimos una vez más para encontrar el 4 × 2 = segundo elemento más pequeño en el subárbol enraizado en el nodo con clave 38. Ahora encontramos que su subárbol izquierdo tiene tamaño 1, lo que significa que es el segundo elemento más pequeño. Por tanto, el procedimiento devuelve un puntero al nodo con la clave 38.

Debido a que cada llamada recursiva desciende un nivel en el árbol de estadísticas de pedidos, el tiempo total para OS-SELECT es, en el peor de los casos, proporcional a la altura del árbol. Dado que el árbol es rojo-negro árbol, su altura es *O* (lg *n*), donde *n* es el número de nodos. Por lo tanto, el tiempo de ejecución de OS-SELECT es *O* (lg *n*) para un conjunto dinámico de *n* elementos.

Determinando el rango de un elemento

Dado un puntero a un nodo *x* en un árbol de orden-estadística *T* , el procedimiento OS-RANK devuelve el la posición de *x* en el orden lineal determinada por un recorrido del árbol de finde *T* .

```
RANGO DE SO ( T , x )
1 r ← tamaño [ izquierda [ x ]] + 1
2 y ← x
3 mientras que y ≠ raíz [ T ]
4 hacer si y = correcto [ p [ y ]]
5 luego r ← r + tamaño [ izquierda [ p [ y ]]] + 1
6 y ← p [ y ]
7 devuelve r
```

El procedimiento funciona de la siguiente manera. El rango de *x* puede verse como el número de nodos precede a *x* en un árbol en orden, más 1 para *x en* sí. OS-RANK mantiene lo siguiente invariante de bucle:

- Al comienzo de cada iteración del **tiempo** de bucle de las líneas 3-6, *r* es el rango de *clave* [*x*] en el subárbol enraizado en el nodo *y* .

Usamos este ciclo invariante para mostrar que OS-RANK funciona correctamente de la siguiente manera:

Página 264

- **Inicialización:** antes de la primera iteración, la línea 1 establece que *r* sea el rango de la *clave* [*x*] dentro de el subárbol enraizado en *x* . Establecer *y* ← *x* en la línea 2 hace que el invariante sea verdadero la primera vez se ejecuta la prueba en la línea 3.
- **Mantenimiento:** Al final de cada iteración del **tiempo** de bucle, que establece *y* ← *p* [*y*]. Así debemos mostrar que si *r* es el rango de la *clave* [*x*] en el subárbol enraizado en *y* al comienzo de la cuerpo del bucle, entonces *r* es el rango de la *clave* [*x*] en el subárbol enraizado en *p* [*y*] al final del cuerpo de bucle. En cada iteración del **tiempo** de bucle, consideramos el subárbol con raíces en *p* [*y*]. Ya hemos contado el número de nodos en el subárbol enraizados en el nodo *y* que preceden *x* en un paseo a finde, por lo que hay que añadir los nodos en el subárbol con raíz en *Y* 's hermano que precede a *x* en una caminata en orden, más 1 para *p* [*y*] si también precede a *x* . Si *y* es un hijo izquierdo, entonces ni *p* [*y*] ni ningún nodo en el subárbol derecho de *p* [*y*] precede a *x* , así que dejamos *r* solo. De lo contrario, *y* es un hijo derecho y todos los nodos en el subárbol izquierdo de *p* [*y*] preceden a *x* , al igual que *p* [*y*] mismo. Por lo tanto, en la línea 5, agregamos *size* [*left* [*p* [*y*]]] + 1 al valor actual de *r* .
- **Terminación:** el bucle termina cuando *y* = *raíz* [*T*], de modo que el subárbol con raíz en *y* es todo el árbol. Por tanto, el valor de *r* es el rango de la *clave* [*x*] en todo el árbol.

Como ejemplo, cuando ejecutamos OS-RANK en el árbol de orden-estadística de la Figura 14.1 para encontrar el rango del nodo con clave 38, obtenemos la siguiente secuencia de valores de *clave* [*y*] *yr* en el parte superior de la **mientras** bucle:

tecla de **iteración**	y	r
1	38	2
2	30	4
3	41	4
4	26	17

Se devuelve el rango 17.

Puesto que cada iteración de la **mientras** bucle toma *O* (1) el tiempo, y *y* sube un nivel en el árbol con cada iteración, el tiempo de ejecución de OS-RANK es, en el peor de los casos, proporcional a la altura del árbol: *O* (lg *n*) en un árbol de estadísticas de orden de *n* nodos.

Mantener los tamaños de los subárboles

Dado el campo de *tamaño* en cada nodo, OS-SELECT y OS-RANK pueden calcular rápidamente el orden información estadística. Pero a menos que estos campos puedan ser mantenidos eficientemente por los modificando operaciones en árboles rojo-negros, nuestro trabajo habrá sido en vano. Lo haremos ahora muestran que los tamaños de los subárboles se pueden mantener tanto para la inserción como para la eliminación sin afectar la tiempo de ejecución asintótico de cualquier operación.

En la sección 13.3 observamos que la inserción en un árbol rojo-negro consta de dos fases. El primero fase baja por el árbol desde la raíz, insertando el nuevo nodo como hijo de un nodo existente. La segunda fase sube por el árbol, cambia de color y finalmente realiza rotaciones para mantener las propiedades rojo-negro.

Para mantener los tamaños de los subárboles en la primera fase, simplemente incrementamos el *tamaño* [*x*] para cada nodo *x* en el camino atravesado desde la raíz hasta las hojas. El nuevo nodo agregado adquiere un *tamaño* de

Página 265

1. Dado que hay *O* (lg *n*) nodos en la ruta atravesada, el costo adicional de mantener la *los* campos de *tamaño* es *O* (lg *n*).

En la segunda fase, los únicos cambios estructurales en el árbol rojo-negro subyacente son causados por rotaciones, de las cuales hay como máximo dos. Además, una rotación es una operación local: solo dos los nodos tienen sus campos de *tamaño* invalidados. El vínculo alrededor del cual se realiza la rotación es incidente en estos dos nodos. Refiriéndose al código para ROTACIÓN IZQUIERDA (*T* , *x*) en la Sección 13.2 , agregamos las siguientes líneas:

12 *talla* [*y*] ← *talla* [*x*]

13 *tamaño* [*x*] ← *tamaño* [*izquierda* [*x*]] + *tamaño* [*derecha* [*x*]] + 1

La figura 14.2 ilustra cómo se actualizan los campos. El cambio a ROTAR DERECHA es simétrico.

Figura 14.2: Actualización de los tamaños de los subárboles durante las rotaciones. El vínculo alrededor del cual se realiza la rotación realizado es incidente en los dos nodos cuyos campos de *tamaño* deben actualizarse. Las actualizaciones son local, requiriendo solo la información de *tamaño* almacenada en *x, y* , y las raíces de los subárboles mostrados como triangulos.

Dado que se realizan como máximo dos rotaciones durante la inserción en un árbol rojo-negro, solo $O(1)$ se dedica más tiempo a actualizar los campos de *tamaño* en la segunda fase. Por tanto, el tiempo total para La inserción en un árbol estadístico de orden de n- nodos es $O(\lg n)$, que es asintóticamente igual que para un árbol rojo-negro ordinario.

La supresión de un árbol rojo-negro también consta de dos fases: la primera opera en el subyacente árbol de búsqueda, y el segundo causa como máximo tres rotaciones y de lo contrario no realiza ninguna estructura cambios. (Consulte la Sección 13.4.) La primera fase empalma un nodo y . Para actualizar el subárbol tamaños, simplemente recorremos una ruta desde el nodo y hasta la raíz, disminuyendo el campo de *tamaño* de cada nodo en la ruta. Dado que esta ruta tiene una longitud $O(\lg n)$ en un árbol rojo-negro de n nodos, el el tiempo empleado en mantener los campos de *tamaño* en la primera fase es $O(\lg n)$. Las rotaciones $O(1)$ en el La segunda fase de eliminación puede manejarse de la misma manera que para la inserción. Por tanto, tanto la inserción y eliminación, incluido el mantenimiento de los campos de *tamaño* , toma $O(\lg n)$ tiempo para un n - árbol de estadística de orden de nodo.

Ejercicios 14.1-1

Muestre cómo opera OS-SELECT (T , 10) en el árbol rojo-negro T de la figura 14.1 .

Ejercicios 14.1-2

Página 266

Muestre cómo OS-RANK (T , x) opera en el árbol rojo-negro T de la figura 14.1 y el nodo x con la *tecla* [x] = 35.

Ejercicios 14.1-3

Escriba una versión no recursiva de OS-SELECT.

Ejercicios 14.1-4

Escribe un procedimiento recursivo OS-KEY-RANK (T , k) que toma como entrada un árbol de orden-estadística T y una clave k y devuelve el rango de k en el conjunto dinámico representados por T . Suponga que el las claves de T son distintas.

Ejercicios 14.1-5

Dado un elemento x en un árbol estadístico de orden de n nodos y un número natural i , ¿cómo puede el i- ésimo sucesor de x en el orden lineal del árbol se determinará en $O(\lg n)$ tiempo?

Ejercicios 14.1-6

Observe que siempre que se hace referencia al campo de *tamaño* de un nodo en OS-SELECT u OS-RANK, se usa solo para calcular el rango del nodo en el subárbol enraizado en ese nodo. En consecuencia, suponga que almacenamos en cada nodo su rango en el subárbol del cual es la raíz. Muestre cómo se puede mantener esta información durante la inserción y eliminación. (Recuerda eso estas dos operaciones pueden provocar rotaciones.)

Ejercicios 14.1-7

Muestre cómo usar un árbol estadístico de orden para contar el número de inversiones (vea el problema 2-4) en una matriz de tamaño n en el tiempo $O(n \lg n)$.

Ejercicios 14.1-8: ⋆

Página 267

Considere n acordes en un círculo, cada uno definido por sus extremos. Describe un tiempo $O(n \lg n)$ algoritmo para determinar el número de pares de cuerdas que se cruzan dentro del círculo. (Por Por ejemplo, si los n acordes son todos diámetros que se encuentran en el centro, entonces la respuesta correcta es .) Suponga que no hay dos acordes que compartan un punto final.

14.2 Cómo aumentar una estructura de datos

Se produce el proceso de aumentar una estructura de datos básica para admitir funciones adicionales. con bastante frecuencia en el diseño de algoritmos. Se utilizará de nuevo en la siguiente sección para diseñar un estructura que soporta operaciones en intervalos. En esta sección, examinaremos los pasos involucrados en tal aumento. También probaremos un teorema que nos permite aumentar el rojo. árboles negros fácilmente en muchos casos.

El aumento de una estructura de datos se puede dividir en cuatro pasos:

1. elegir una estructura de datos subyacente,
2. determinar la información adicional que debe mantenerse en la estructura de datos subyacente,
3. Verificar que la información adicional pueda mantenerse para la modificación básica. operaciones en la estructura de datos subyacente, y
4. desarrollo de nuevas operaciones.

Al igual que con cualquier método de diseño prescriptivo, no debe seguir ciegamente los pasos en el orden dado. La mayoría del trabajo de diseño contiene un elemento de prueba y error, y el progreso en todos los pasos. generalmente procede en paralelo. No tiene sentido, por ejemplo, determinar información y desarrollo de nuevas operaciones (pasos 2 y 4) si no seremos capaces de mantener la información adicional de manera eficiente. Sin embargo, este método de cuatro pasos proporciona una buena concentrarse en sus esfuerzos para aumentar una estructura de datos, y también es una buena manera de organizar documentación de una estructura de datos aumentada.

Seguimos estos pasos en la Sección 14.1 para diseñar nuestros árboles de estadísticas de pedidos. Para el paso 1, eligió árboles rojo-negro como estructura de datos subyacente. Una pista sobre la idoneidad del rojo-negro árboles proviene de su apoyo eficiente de otras operaciones de conjuntos dinámicos en un orden total, como como MÍNIMO, MÁXIMO, SUCESOR y PREDECESOR.

Para el paso 2, proporcionamos el campo de *tamaño* , en el que cada nodo x almacena el tamaño del subárbol arraigado en x . Generalmente, la información adicional hace que las operaciones sean más eficientes. por ejemplo, podríamos haber implementado OS-SELECT y OS-RANK usando solo las claves almacenadas en el árbol, pero no habrían corrido en el tiempo $O(\lg n)$. A veces, el adicional la información es información de puntero en lugar de datos, como en el ejercicio 14.2-1 .

Para el paso 3, nos aseguramos de que la inserción y eliminación pudieran mantener los campos de *tamaño* mientras aún

corriendo en tiempo $O(\lg n)$. Idealmente, solo algunos elementos de la estructura de datos deberían necesitar ser actualizado para mantener la información adicional. Por ejemplo, si simplemente almacenamos en cada nodo su rango en el árbol, los procedimientos OS-SELECT y OS-RANK se ejecutarían rápidamente, pero insertar un nuevo elemento mínimo causaría un cambio en esta información en cada nodo del árbol. Cuando almacenamos tamaños de subárbol en su lugar, la inserción de un nuevo elemento genera información para cambiar solo en los nodos $O(\lg n)$.

Página 268

Para el paso 4, desarrollamos las operaciones OS-SELECT y OS-RANK. Después de todo, la necesidad de nuevas operaciones es la razón por la que nos molestamos en aumentar una estructura de datos en primer lugar. De vez en cuando, en lugar de desarrollar nuevas operaciones, utilizamos la información adicional para agilizar los unos, como en el ejercicio 14.2-1 .

Aumento de árboles rojo-negros

Cuando los árboles rojo-negro subyacen a una estructura de datos aumentada, podemos probar que ciertos tipos de La información adicional siempre se puede mantener de manera eficiente mediante inserción y eliminación, por lo que haciendo el paso 3 muy fácil. La demostración del siguiente teorema es similar al argumento de Sección 14.1 que el campo de *tamaño* se puede mantener para árboles de estadísticas de pedidos.

Teorema 14.1: (Aumento de un árbol rojo-negro)

Sea f un campo que aumenta un árbol rojo-negro T de n nodos, y suponga que el contenido de f para un nodo x se puede calcular usando solo la información en los nodos x , $izquierda[x]$ y $derecha[x]$, incluyendo $f[izquierda[x]]$ y $f[derecha[x]]$. Entonces, podemos mantener los valores de f en todos los nodos de T durante la inserción y eliminación sin afectar asintóticamente el rendimiento $O(\lg n)$ de estas operaciones.

Prueba La idea principal de la prueba es que un cambio en uncampo f en un nodo x *se* propaga solo a antepasados de x en el árbol. Es decir, cambiar $f[x]$ puede requerir que $f[p[x]]$ se actualice, pero nada más; la actualización de $f[p[x]]$ puede requerir que $f[p[p[x]]]$ se actualice, pero nada más; y así sucesivamente hasta el árbol. Cuando se actualiza $f[root[T]]$, ningún otro nodo depende del nuevo valor, por lo que el proceso termina. Dado que la altura de un árbol rojo-negro es $O(\lg n)$, cambiar un campo f en un nodo cuesta $O(\lg n)$ tiempo en la actualización de nodos dependiendo del cambio.

La inserción de un nodo x en T consta de dos fases. (Consulte la Sección 13.3.) Durante la primera fase, x se inserta como hijo de un nodo $p[x]$ existente. El valor de $f[x]$ se puede calcular en $O(1)$ tiempo ya que, por suposición, depende sólo de la información en los otros campos de x mismo y el información en x 's hijos, pero x ' s niños son tanto el centinela $nula[T]$. Una vez que se calcula $f[x]$, el cambio se propaga por el árbol. Por tanto, el tiempo total para la primera fase de inserción es $O(\lg n)$. Durante la segunda fase, los únicos cambios estructurales en el árbol provienen de las rotaciones. Ya que solo dos nodos cambian en una rotación, el tiempo total para actualizar los campos f es $O(\lg n)$ por rotación. Dado que el número de rotaciones durante la inserción es como máximo dos, el tiempo total para la inserción es $O(\lg n)$.

Al igual que la inserción, la eliminación tiene dos fases. (Consulte la Sección 13.4.) En la primera fase, los cambios en el árbol ocurre si el nodo eliminado es reemplazado por su sucesor, y cuando el nodo eliminado o su sucesor está empalmado. Propagar las actualizaciones *af* provocadas por estos cambios cuesta como máximo $O(\lg n)$ ya que los cambios modifican el árbol localmente. Arreglando el árbol rojo-negro durante el la segunda fase requiere como máximo tres rotaciones, y cada rotación requiere como máximo $O(\lg n)$ tiempo para propagar las actualizaciones a f. Por lo tanto, al igual que la inserción, el tiempo total para la eliminación es $O(\lg n)$.

Página 269

En muchos casos, como el mantenimiento de los campos de *tamaño* en árboles de estadísticas de orden, el costo de la actualización después de una rotación es $O(1)$, en lugar del $O(\lg n)$ derivado en la demostración del Teorema 14.1 . El ejercicio 14.2-4 da un ejemplo.

Ejercicios 14.2-1

Muestre cómo las consultas de conjunto dinámico MINIMUM, MAXIMUM, SUCCESSOR y Cada uno de los PREDECESORES se puede admitir en $O(1)$ en el peor de los casos en un orden aumentado. árbol de estadísticas. El desempeño asintótico de otras operaciones en árboles de orden-estadística debería no se verá afectado. (*Sugerencia:* agregue punteros a los nodos).

Ejercicios 14.2-2

¿Se pueden mantener las alturas negras de los nodos en un árbol rojo-negro como campos en los nodos del árbol sin afectar el desempeño asintótico de ninguna de las operaciones de árbol rojo-negro? Muestre cómo o discuta por qué no.

Ejercicios 14.2-3

¿Pueden las profundidades de los nodos en un árbol rojo-negro mantenerse eficientemente como campos en los nodos de ¿el árbol? Muestre cómo o discuta por qué no.

Ejercicios 14.2-4: ⋆

Sea $\otimes$ un operador binario asociativo, y sea a un campo mantenido en cada nodo de un árbol negro. Supongamos que queremos incluir en cada nodo x un campo adicional f tal que $f[x] = a[x_1] \otimes a[x_2] \otimes \cdots \otimes a[x_m]$, donde $x_1, x_2, \ldots, x_m$ es la lista en orden de los nodos en el subárbol arraigado en x . Muestre que los campos f se pueden actualizar correctamente en $O(1)$ tiempo después de una rotación. Modifique su argumento ligeramente para mostrar que los campos de *tamaño* en árboles de estadísticas de orden pueden ser mantenido en $O(1)$ tiempo por rotación.

Ejercicios 14.2-5: ⋆

Deseamos aumentar los árboles rojo-negro con una operación RB-ENUMERATE (x , a , b) que genera todas las claves k tales que $a \le k \le b$ en un árbol rojo-negro enraizado en x . Describa cómo RB-ENUMERATE se puede implementar en $\Theta(m + \lg n)$ tiempo, donde m es el número de teclas que

Página 270

son de salida y n es el número de nodos internos en el árbol. (*Sugerencia:* no es necesario agregar nuevos campos al árbol rojo-negro.)

14.3 Árboles de intervalo

En esta sección, aumentaremos árboles rojo-negro para respaldar operaciones en conjuntos dinámicos de intervalos. Un **intervalo cerrado** es un par ordenado de números reales $[t_1, t_2]$, con $t_1 \le t_2$. El intervalo $[t_1, t_2]$ representa el conjunto $\{t \in \mathbf{R} : t_1 \le t \le t_2\}$. **Abrir** y **semiabiertas** intervalos omiten ambos o uno de los puntos finales del conjunto, respectivamente. En esta sección, asumiremos que los intervalos son cerrado; extender los resultados a intervalos abiertos y semiabiertos es conceptualmente sencillo.

Los intervalos son convenientes para representar eventos que cada uno ocupa un período continuo de tiempo. Podríamos, por ejemplo, desear consultar una base de datos de intervalos de tiempo para averiguar qué eventos ocurrió durante un intervalo dado. La estructura de datos en esta sección proporciona un medio eficiente para mantener dicha base de datos de intervalos.

Podemos representar un intervalo $[t_1, t_2]$ como un objeto i, con campos $bajo[i] = t_1$ (el **punto final bajo**) y $alto[i] = t_2$ (el **punto final alto**). Decimos que los intervalos i e i' se **superponen** si $i \cap i' \neq \emptyset$, es decir, si $bajo[i] \leq alto[i']$ y $bajo[i'] \leq alto[i]$. Dos intervalos cualesquiera i e i' satisfacen el **intervalo tricotomía**; es decir, se cumple exactamente una de las siguientes tres propiedades:

a. i y i' se superponen,
si. i está a la izquierda de i' (es decir, $alto[i] < bajo[i']$),
C. i está a la derecha de i' (es decir, $alto[i'] < bajo[i]$).

La figura 14.3 muestra las tres posibilidades.

Figura 14.3: Tricotomía de intervalo para dos intervalos cerrados i e i'. *(a)* Si i y i' se superponen, hay cuatro situaciones; en cada uno, $bajo[i] \leq alto[i']$ y $bajo[i'] \leq alto[i]$. *(b)* Los intervalos hacen no se superponen, y $alto[i] < bajo[i']$. *(c)* Los intervalos no se superponen y $alto[i'] < bajo[i]$.

Un **árbol de intervalo** es un árbol rojo-negro que mantiene un conjunto dinámico de elementos, con cada elemento x *que* contiene un intervalo $int[x]$. Los árboles de intervalo admiten las siguientes operaciones.

- INTERVAL-INSERT (T, x) agrega el elemento x, cuyo campo int se supone que contiene un intervalo, con el árbol de intervalo T.
- INTERVALO-ELIMINAR (t, x) elimina el elemento x del árbol intervalo T.
- INTERVAL-SEARCH (T, i) devuelve un puntero a un elemento x en el árbol de intervalos T tal que $int[x]$ se superpone al intervalo i, o el centinela $nil[T]$ si no hay tal elemento en el conjunto.

Página 271

La figura 14.4 muestra cómo un árbol de intervalos representa un conjunto de intervalos. Seguiremos los cuatro método de pasos de la Sección 14.2 mientras revisamos el diseño de un árbol de intervalos y las operaciones que se ejecutan en él.

Figura 14.4: Un árbol de intervalos. *(a)* Un conjunto de 10 intervalos, que se muestran ordenados de abajo hacia arriba por la izquierda punto final. *(b)* El árbol de intervalos que los representa. Un árbol en orden a pie del árbol enumera los nodos en orden ordenado por el extremo izquierdo.

Paso 1: Estructura de datos subyacente

Elegimos un árbol rojo-negro en el que cada nodo x contiene un intervalo $int[x]$ y la clave de x es el punto final bajo, $bajo[int[x]]$, del intervalo. Por lo tanto, una caminata de árbol en orden de la estructura de datos enumera los intervalos en orden ordenado por punto final bajo.

Paso 2: información adicional

Además de los intervalos en sí, cada nodo x contiene un valor $max[x]$, que es el valor máximo de cualquier punto final de intervalo almacenado en el subárbol con raíz en x.

Paso 3: mantenimiento de la información

Debemos verificar que la inserción y eliminación se puede realizar en tiempo $O(\lg n)$ en un intervalo árbol de n nodos. Podemos determinar $max[x]$ intervalo dado $int[x]$ y los max valores de nodo x's niños:

$max[x] = \max(alto[int[x]], max[izquierda[x]], max[derecha[x]])$.

Por tanto, según el teorema 14.1, la inserción y la eliminación se ejecutan en el tiempo $O(\lg n)$. De hecho, actualizar el *máximo* Los campos después de una rotación se pueden lograr en $O(1)$ tiempo, como se muestra en los Ejercicios 14.2-4 y 14.3-1.

Paso 4: desarrollo de nuevas operaciones

Página 272

La única operación nueva que necesitamos es INTERVAL-SEARCH (T, i), que encuentra un nodo en el árbol T cuyo intervalo se superpone al intervalo i. Si no hay un intervalo que se superponga a i en el árbol, un puntero a se devuelve el centinela $nil[T]$.

```
INTERVALO DE BÚSQUEDA (T, i)
1 x ← raíz[T]
2 mientras que x ≠ nil[T] e i no se superponen int[x]
3 hacer si se deja[x] ≠ nil[T] y max[left[x]] ≥ bajo[i]
4                 luego x ← izquierda[x]
5                 else x ← derecha[x]
6 retorno x
```

La búsqueda de un intervalo que se superponga a i comienza con x en la raíz del árbol y continúa hacia abajo. Termina cuando se encuentra un intervalo superpuesto o cuando x apunta al centinela $cero[T]$. Dado que cada iteración del ciclo básico toma $O(1)$ tiempo, y dado que la altura de un árbol rojo-negro de n nodos es $O(\lg n)$, el procedimiento INTERVAL-SEARCH toma $O(\lg n)$ tiempo.

Antes de ver por qué INTERVAL-SEARCH es correcto, examinemos cómo funciona en el árbol de intervalos en la Figura 14.4. Suponga que deseamos encontrar un intervalo que se superpone al intervalo $i = [22, 25]$. Comenzamos con x como la raíz, que contiene $[16, 21]$ y no se superpone a i. Ya que $max[left[x]] = 23$ es mayor que $low[i] = 22$, el bucle continúa con x como el hijo izquierdo del raíz: el nodo que contiene $[8, 9]$, que tampoco se superpone i. Esta vez, $max[left[x]] = 10$ es menor que $bajo[i] = 22$, por lo que el ciclo continúa con el hijo derecho de x como el nuevo x. El intervalo $[15, 23]$ almacenado en este nodo se superpone a i, por lo que el procedimiento devuelve este nodo.

Como ejemplo de una búsqueda fallida, suponga que deseamos encontrar un intervalo que se superponga a $i = [11, 14]$ en el árbol de intervalos de la Figura 14.4. Una vez más comenzamos con x como raíz. Desde el el intervalo de la raíz $[16, 21]$ no se superpone a i, y dado que $max[left[x]] = 23$ es mayor que $low[i] = 11$, vamos a la izquierda al nodo que contiene $[8, 9]$. (Tenga en cuenta que ningún intervalo en el subárbol derecho se superpone *i,* veremos por qué más adelante.) El intervalo $[8, 9]$ no se superpone a i, y $max[izquierda[x]] = 10$ es menor que $bajo[i] = 11$, así que vamos a la derecha. (Tenga en cuenta que ningún intervalo en el subárbol izquierdo se superpone i.) Intervalo $[15, 23]$ no se superpone i, y su hijo izquierdo es $nil[T]$, así que vamos a la derecha, el bucle termina y el Se devuelve centinela $nulo[T]$.

Para ver por qué INTERVAL-SEARCH es correcto, debemos entender por qué es suficiente examinar un solo camino desde la raíz. La idea básica es que en cualquier nodo x, si $int[x]$ no se superpone i, el La búsqueda siempre procede en una dirección segura: definitivamente se encontrará un intervalo superpuesto si hay uno en el árbol. El siguiente teorema establece esta propiedad con mayor precisión.

Teorema 14.2

Cualquier ejecución de INTERVAL-SEARCH (*T, i*) devuelve un nodo cuyo intervalo se superpone a *i* , o devuelve *nil* [*T*] y el árbol *T* no contiene ningún nodo cuyo intervalo se superponga a *i* .

Prueba El **tiempo** de bucle de las líneas 2-5 termina ya sea cuando *x* = *nil* [*T*] o *i* superposiciones *int* [*x*]. En el En este último caso, ciertamente es correcto devolver *x* . Por tanto, nos centramos en el primer caso, en el que el **mientras** termina de bucle porque *x* = *nil* [*T*].

Utilizamos la siguiente invariante para el **tiempo** de bucle de las líneas 2-5:

Página 273

- Si el árbol *T* contiene un intervalo que se superpone a *i* , entonces existe tal intervalo en el subárbol enraizado en *x* .

Usamos este ciclo invariante de la siguiente manera:

- **Inicialización:** antes de la primera iteración, la línea 1 establece *x* como la raíz de *T* , de modo que invariante sostiene.
- **Mantenimiento:** En cada iteración del **tiempo** de bucle, o bien se ejecuta la línea 4 o la línea 5. Demostraremos que el ciclo invariante se mantiene en cualquier caso.

  Si se ejecuta la línea 5, entonces debido a la condición de bifurcación en la línea 3, nos *queda* [*x*] = *cero* [*T*], o *máximo* [*izquierda* [*x*]] < *bajo* [*i*]. Si *left* [*x*] = *nil* [*T*], el subárbol enraizado a la *izquierda* [*x*] claramente no contiene ningún intervalo que se superponga a *i* , por lo que establecer *x* a la *derecha* [*x*] mantiene la invariante. Por lo tanto, suponga que *left* [*x*] ≠ *nil* [*T*] y *max* [*left* [*x*]] < *low* [*i*]. Como figura 14.5 (a) muestra, para cada intervalo *i* 'en el subárbol izquierdo de *x* , tenemos

  alto [*i* '] ≤ *max* [*izquierda* [*x*]]
  < *bajo* [*i*].

  Figura 14.5: Intervalos en la demostración del teorema 14.2. Se muestra el valor de *max* [*left* [*x*]] en cada caso como una línea discontinua. *(a)* La búsqueda va bien. Sin intervalo *i* 'en el subárbol izquierdo de *x* puede superponerse *i* . *(b)* La búsqueda va a la izquierda. El subárbol izquierdo de *x* contiene un intervalo que se superpone *i* (situación no mostrada), o hay un intervalo *i* 'en el subárbol izquierdo de *x* tal que *alto* [*i* '] = *máximo* [*izquierda* [*x*]]. Dado que *i* no se superpone a *i* ', tampoco se superpone a ningún intervalo *i* "en el subárbol derecho de *x* , ya que *bajo* [*i* '] ≤ *bajo* [*i* "].

  Por lo tanto, en la tricotomía de intervalo, *i* 'e *i* no se superponen. Por lo tanto, el subárbol izquierdo de *x* no contiene intervalos que se superpongan a *i* , de modo que establecer *x* a la *derecha* [*x*] mantiene la invariante.

  Si, por otro lado, se ejecuta la línea 4, mostraremos que el contrapositivo de el bucle invariante se mantiene. Es decir, si no hay un intervalo superpuesto *i* en el subárbol enraizada en *la izquierda* [*x*], entonces no hay intervalo de superposición *i* en cualquier parte del árbol. Desde linea 4 se ejecuta, entonces debido a la condición de bifurcación en la línea 3, tenemos *max* [*left* [*x*]] ≥ *bajo* [*i*]. Además, por definición del campo *máximo* , debe haber algún intervalo *i* 'en *x* ' s subárbol izquierdo tal que

  alto [*i* '] = *máximo* [*izquierda* [*x*]]
  ≥ *bajo* [*i*].

  (La figura 14.5 (b) ilustra la situación.) Dado que *i* e *i* 'no se superponen, y dado que es No es cierto que *alto* [*i* '] < *bajo* [*i*], sigue la tricotomía de intervalo que *alto* [*i*] < *bajo* [*i* ']. Los árboles de intervalo se codifican en los puntos finales inferiores de los intervalos y, por lo tanto, el árbol de búsqueda La propiedad implica que para cualquier intervalo *i* "en el subárbol derecho de *x* ,

Página 274

$alto[i] < bajo[i']$
$\leq bajo[i'']$.

Por la tricotomía de intervalo, i e i''no se superponen. Concluimos que si cualquier intervalo en el subárbol izquierdo de x *se* superpone a i , estableciendo x a la *izquierda* [x] mantiene el invariante.

- **Terminación:** si el bucle termina cuando $x = cero[T]$, no hay ningún intervalo superpuesto i en el subárbol enraizado en x . El contrapositivo del invariante de bucle implica que T no contiene ningún intervalo que se superponga i . Por tanto, es correcto devolver $x = nil[T]$.

Por tanto, el procedimiento INTERVAL-SEARCH funciona correctamente.

Ejercicios 14.3-1

Escriba un pseudocódigo para ROTACIÓN IZQUIERDA que opera en nodos en un árbol de intervalo y actualiza los campos *máximos* en tiempo $O(1)$.

Ejercicios 14.3-2

Vuelva a escribir el código para INTERVAL-SEARCH para que funcione correctamente cuando todos los intervalos son se supone que está abierto.

Ejercicios 14.3-3

Describe un algoritmo eficiente que, dado un intervalo i , devuelve un intervalo superpuesto i que tiene el punto final mínimo bajo, o $nulo[T]$ si no existe tal intervalo.

Ejercicios 14.3-4

Dado un árbol de intervalos T y un intervalo i , describa cómo todos los intervalos en T que se superponen i pueden ser enumerados en $O(\min(n, k \lg n))$ tiempo, donde k es el número de intervalos en la lista de salida. (*Opcional:* busque una solución que no modifique el árbol).

Ejercicios 14.3-5

Página 275

Sugerir modificaciones a los procedimientos del árbol de intervalos para respaldar la nueva operación INTERVAL-SEARCH-EXACTLY (T, i), que devuelve un puntero a un nodo x en el árbol de intervalos T tal que $bajo[int[x]] = bajo[i]$ y $alto[int[x]] = alto[i]$, o $nil[T]$ si T no contiene tal nodo. Todas las operaciones, incluida INTERVAL-SEARCH-EXACTLY, deben ejecutarse en tiempo $O(\lg n)$ en un árbol de n nodos.

Ejercicios 14.3-6

Muestre cómo mantener un conjunto dinámico Q de números que admita la operación MIN-GAP, lo que da la magnitud de la diferencia de los dos números más cercanos en Q . Por ejemplo, si Q = {1, 5, 9, 15, 18, 22}, luego MIN-GAP (Q) devuelve 18 - 15 = 3, ya que 15 y 18 son los dos los números más cercanos en Q . Realice las operaciones INSERT, DELETE, SEARCH y MIN-GAP como lo más eficiente posible y analizar sus tiempos de ejecución.

Ejercicios 14.3-7: ⋆

Las bases de datos VLSI comúnmente representan un circuito integrado como una lista de rectángulos. Asumir que cada rectángulo está orientada de forma rectilínea (lados paralelos a la x - y y eje x), de modo que una La representación de un rectángulo consta de sus coordenadas x e y mínima y máxima . Dar un algoritmo de tiempo O (n lg n) para decidir si un conjunto de rectángulos se representa o no contiene dos rectángulos que se superponen. Su algoritmo no necesita informar todos los pares que se cruzan, pero debe informar que existe una superposición si un rectángulo cubre completamente a otro, incluso si el las líneas de límite no se cruzan. (*Sugerencia:* mueva una línea de "barrido" a lo largo del conjunto de rectángulos).

Problemas 14-1: Punto de máxima superposición

Supongamos que deseamos realizar un seguimiento de un **punto de máxima superposición** en un conjunto de intervalos: un punto que tiene el mayor número de intervalos en la base de datos superpuestos.

a. Demuestre que siempre habrá un punto de máxima superposición que es un punto final de uno de los segmentos.
si. Diseñar una estructura de datos que soporte eficientemente las operaciones INTERVAL-INSERT, INTERVAL-DELETE y FIND-POM, que devuelve un punto de máxima superposición. (*Sugerencia:* mantenga un árbol rojo-negro de todos los puntos finales. Asocie un valor de +1 con cada extremo izquierdo y asociar un valor de -1 con cada extremo derecho. Aumente cada nodo del árbol con información adicional para mantener el punto de máxima superposición).

Problemas 14-2: Permutación de Josefo

Página 276

El **problema de Josefo** se define como sigue. Suponga que n personas están dispuestas en círculo y que se nos da un entero positivo $m \leq n$. Comenzando con una primera persona designada, proceder alrededor del círculo, la eliminación de todos los m ª persona. Después de retirar a cada persona, el conteo continúa alrededor del círculo que queda. Este proceso continúa hasta que todas las n personas se han eliminado. El orden en el que las personas se eliminan del círculo define el (n, m) **-Josephus permutación** de los enteros 1, 2, ..., n . Por ejemplo, el (7, 3) -Josephus la permutación es 3, 6, 2, 7, 5, 1, 4.

a. Suponga que m es una constante. Describe un algoritmo de tiempo O (n) que, dado un número entero n , genera la permutación (n , m) -Josephus.
si. Suponga que m no es una constante. Describe un algoritmo de tiempo O (n lg n) que, dado números enteros n y m , da salida a la (n , m) -Josephus permutación.

Notas del capítulo

En su libro, Preparata y Shamos [247] describen varios de los árboles de intervalo que aparecen en la literatura, citando el trabajo de H. Edelsbrunner (1980) y EM Mc-Creight (1981). El libro detalla un árbol de intervalos para el cual, dada una base de datos estática de n intervalos, todos los k intervalos que la superposición de un intervalo de consulta dado se puede enumerar en tiempo O (k + lg n).

Parte IV: Diseño y análisis avanzados

Técnicas

Lista de capítulos

Introducción

Esta parte cubre tres técnicas importantes para el diseño y análisis de algoritmos eficientes: programación dinámica (capítulo 15), algoritmos codiciosos (capítulo 16) y análisis amortizado (Capítulo 17). Las partes anteriores han presentado otras técnicas ampliamente aplicables, como dividir y conquistar, aleatorización y solución de recurrencias. Las nuevas técnicas son algo más sofisticados, pero son útiles para atacar eficazmente a muchos problemas computacionales. Los temas presentados en esta parte se repetirán más adelante en el libro.

La programación dinámica normalmente se aplica a problemas de optimización en los que un conjunto de opciones debe hacerse para llegar a una solución óptima. A medida que se toman las decisiones, los subproblemas de la misma forma surge a menudo. La programación dinámica es efectiva cuando un subproblema dado puede surgen de más de un conjunto parcial de opciones; la técnica clave es almacenar la solución para cada uno de estos subproblemas en caso de que reaparezca. El capítulo 15 muestra cómo esta simple idea a veces transforman algoritmos de tiempo exponencial en algoritmos de tiempo polinomial.

Al igual que los algoritmos de programación dinámica, los algoritmos codiciosos suelen aplicarse a la optimización problemas en los que se debe hacer un conjunto de elecciones para llegar a una solución óptima. los

Página 277

La idea de un algoritmo codicioso es hacer cada elección de una manera localmente óptima. Un simple ejemplo es el cambio de monedas: para minimizar la cantidad de monedas estadounidenses necesarias para hacer el cambio determinada cantidad, basta con seleccionar repetidamente la moneda de mayor denominación que no sea mayor que la cantidad aún adeuda. Hay muchos problemas de este tipo para los que un enfoque codicioso proporciona una solución óptima mucho más rápidamente que una programación dinámica Acercarse. Sin embargo, no siempre es fácil saber si un enfoque codicioso será efectivo. El capítulo 16 revisa la teoría matroide, que a menudo puede ser útil para hacer tal determinación.

El análisis amortizado es una herramienta para analizar algoritmos que realizan una secuencia de similares operaciones. En lugar de limitar el costo de la secuencia de operaciones limitando el costo real costo de cada operación por separado, se puede utilizar un análisis amortizado para proporcionar un límite en el costo real de toda la secuencia. Una razón por la que esta idea puede ser eficaz es que puede ser imposible en una secuencia de operaciones para que todas las operaciones individuales se ejecuten en su límites de tiempo conocidos en el peor de los casos. Si bien algunas operaciones son costosas, muchas otras pueden ser barato. Sin embargo, el análisis amortizado no es solo una herramienta de análisis; también es una forma de pensar sobre el diseño de algoritmos, desde el diseño de un algoritmo y el análisis de su ejecución el tiempo suele estar estrechamente entrelazado. El capítulo 17 presenta tres formas de realizar una amortización análisis de un algoritmo.

Capítulo 15: Programación dinámica

Visión general

La programación dinámica, como el método divide y vencerás, resuelve problemas combinando las soluciones a los subproblemas. ("Programación" en este contexto se refiere a un método tabular, no escribir código de computadora). Como vimos en el Capítulo 2 , los algoritmos de divide y vencerás el problema en subproblemas independientes, resuelva los subproblemas de forma recursiva, y luego combinan sus soluciones para resolver el problema original. Por el contrario, la programación dinámica es aplicable cuando los subproblemas no son independientes, es decir, cuando los subproblemas comparten subproblemas. En este contexto, un algoritmo de divide y vencerás hace más trabajo que necesario, resolviendo repetidamente los subproblemas comunes. Una programación dinámica El algoritmo resuelve cada subproblema solo una vez y luego guarda su respuesta en una tabla, por lo tanto evitando el trabajo de recalcular la respuesta cada vez que se encuentra el subproblema.

La programación dinámica se aplica normalmente a problemas de optimización . En tales problemas hay

Puede haber muchas soluciones posibles. Cada solución tiene un valor y deseamos encontrar una solución con el valor óptimo (mínimo o máximo). Llamamos a esta solución *una* solución óptima para el problema, en contraposición a *la* solución óptima, ya que puede haber varias soluciones que logren el valor óptimo.

El desarrollo de un algoritmo de programación dinámica se puede dividir en una secuencia de cuatro pasos.

1. Caracterizar la estructura de una solución óptima.
2. Defina de forma recursiva el valor de una solución óptima.
3. Calcule el valor de una solución óptima de abajo hacia arriba.
4. Construya una solución óptima a partir de información calculada.

Página 278

Los pasos 1 a 3 forman la base de una solución de programación dinámica a un problema. El paso 4 puede ser se omite si solo se requiere el valor de una solución óptima. Cuando realizamos el paso 4, a veces mantener información adicional durante el cálculo en el paso 3 para facilitar la construcción de una solución óptima.

Las secciones que siguen utilizan el método de programación dinámica para resolver algunas optimizaciones problemas. La sección 15.1 examina un problema en la programación de dos líneas de montaje de automóviles, donde después de cada estación, el auto en construcción puede permanecer en la misma línea o moverse a la otro. La sección 15.2 pregunta cómo podemos multiplicar una cadena de matrices para que la menor cantidad total se realizan multiplicaciones escalares. Dados estos ejemplos de programación dinámica, la Sección 15.3 analiza dos características clave que debe tener un problema para que la programación dinámica ser una técnica de solución viable. La sección 15.4 muestra cómo encontrar el común más largo. subsecuencia de dos secuencias. Finalmente, la Sección 15.5 usa programación dinámica para construir árboles de búsqueda binaria que son óptimos, dada una distribución conocida de claves para buscar.

15.1 Programación de la línea de montaje

Nuestro primer ejemplo de programación dinámica resuelve un problema de fabricación. El coronel Motors Corporation produce automóviles en una fábrica que tiene dos líneas de montaje, que se muestran en Figura 15.1 . Un chasis de automóvil ingresa a cada línea de ensamblaje, se le agregan partes en un número de estaciones, y un auto terminado sale al final de la línea. Cada línea de montaje tiene n estaciones, numeradas $j = 1, 2, ..., n$. Denotamos la j- ésima estación en la línea i (donde i es 1 o 2) por $S_{i,j}$. La j- ésima estación de la línea 1 ($S_{1,j}$) realiza la misma función que la j- ésima estación de la línea 2 ($S_{2,j}$). los Las estaciones fueron construidas en diferentes momentos y con diferentes tecnologías, sin embargo, para que el tiempo requerido en cada estación varía, incluso entre estaciones en la misma posición en los dos diferentes líneas. Denotamos el tiempo de montaje requerido en la estación $S_{i,j}$ por $a_{i,j}$. Como muestra la Figura 15.1 , un el chasis ingresa a la estación 1 de una de las líneas de montaje, y avanza desde cada estación hasta la siguiente. También hay un tiempo de entrada e_i para que el chasis entre en la línea de montaje i y un tiempo de salida x_i para que el automóvil terminado salga de la línea de montaje i .

Figura 15.1: Un problema de fabricación para encontrar el camino más rápido a través de una fábrica. Hay dos líneas de montaje, cada una con n estaciones; la j- ésima estación en la línea i se denota $S_{i,j}$ y el conjunto el tiempo en esa estación es $a_{i,j}$. Un chasis de automóvil entra en la fábrica y pasa a la línea i (donde $i = 1$ o 2), tomando e_i tiempo. Después de pasar por la j- ésima estación en una línea, el chasis pasa a la estación ($j + 1$) st en cualquier línea. No hay costo de transferencia si permanece en la misma línea, pero toma tiempo $t_{i,j}$ para transferir a la otra línea después de la estación $S_{i,j}$. Después de salir del n º estación en una línea, toma x_i tiempo para que el auto completado salga de fábrica. El problema es determinar qué estaciones elegir de la línea 1 y cuáles elegir de la línea 2 para minimizar la tiempo total de fábrica para un auto.

Normalmente, una vez que un chasis ingresa a una línea de ensamblaje, solo pasa por esa línea. El tiempo para ir de una estación a la siguiente dentro de la misma línea de montaje es insignificante. Ocasionalmente un llega un pedido urgente especial y el cliente desea que el automóvil se fabrique como

Página 279

lo mas rapido posible. Para pedidos urgentes, el chasis todavía pasa por las n estaciones en orden, pero el gerente de la fábrica puede cambiar el auto parcialmente completado de una línea de ensamblaje a la otro después de cualquier estación. El tiempo para transferir un chasis fuera de la línea de montaje i después de haber pasado por la estación $S_{i,j}$ es $t_{i,j}$, donde $i = 1, 2$ y $j = 1, 2, ..., n - 1$ (ya que después de la n- ésima estación, el montaje está completo). El problema es determinar qué estaciones elegir de la línea 1 y cuál elegir de la línea 2 con el fin de minimizar el tiempo total en la fábrica para un auto. En el ejemplo de la Figura 15.2 (a) , el tiempo total más rápido proviene de elegir las estaciones 1, 3 y 6 de la línea 1 y estaciones 2, 4 y 5 de la línea 2.

Figura 15.2: *(a)* Una instancia del problema de la línea de montaje con costos e_i , $a_{i,j}$, $t_{i,j}$ y x_i indicado. El camino muy sombreado indica el camino más rápido a través de la fábrica. *(b)* El valores de $f_i[j]$, f^*, $l_i[j]$ y l^* para el ejemplo del inciso a).

La forma obvia de "fuerza bruta" de minimizar el tiempo en la fábrica no es factible cuando hay muchas estaciones. Si nos dan una lista de qué estaciones usar en la línea 1 y cuáles utilizar en la línea 2, es fácil calcular en $\Theta(n)$ tiempo cuánto tiempo tarda un chasis en pasar a través del fábrica. Desafortunadamente, hay 2^n formas posibles de elegir estaciones, que vemos al ver el conjunto de estaciones utilizadas en la línea 1 como un subconjunto de $\{1, 2, ..., n\}$ y observando que hay 2^n tales subconjuntos. Por lo tanto, determinar el camino más rápido a través de la fábrica enumerando todos los posibles formas y calcular cuánto tiempo toma cada una requeriría $\Omega(2^n)$ tiempo, lo cual es inviable cuando n es grande.

Paso 1: La estructura del camino más rápido a través de la fábrica.

El primer paso del paradigma de programación dinámica es caracterizar la estructura de un solucion optima. Para el problema de programación de la línea de montaje, podemos realizar este paso como sigue. Consideremos la forma más rápida posible para que un chasis llegue desde el punto de partida a través de la estación $S_{1,j}$. Si $j = 1$, solo hay una forma en que el chasis podría haberse ido, y es es fácil determinar cuánto tiempo se tarda en pasar por la estación $S_{1,j}$. Para $j = 2, 3, ..., n$, sin embargo, Hay dos opciones: el chasis podría haber venido de la estación $S_{1,j-1}$ y luego directamente a estación $S_{1,j}$, siendo insignificante el tiempo para pasar de la estación $j - 1$ a la estación j en la misma línea. Alternativamente, el chasis podría haber venido de la estación $S_{2,j-1}$ y luego haber sido transferido a estación $S_{1,j}$, siendo el tiempo de transferencia $t_{2,j-1}$. Consideraremos estas dos posibilidades por separado, aunque veremos que tienen mucho en común.

Primero, supongamos que el camino más rápido a través de la estación $S_{1,j}$ es a través de la estación $S_{1,j-1}$. La clave La observación es que el chasis debe haber tomado un camino más rápido desde el punto de partida hasta estación $S_{1,j-1}$. ¿Por qué? Si hubiera una forma más rápida de pasar por la estación $S_{1,j-1}$, podríamos sustituir esta forma más rápida de ceder un camino más rápido a través de la estación $S_{1,j}$: una contradicción.

Página 280

De manera similar, supongamos ahora que el camino más rápido a través de la estación $S_{1,j}$ es a través de la estación $S_{2,j-1}$. Ahora observamos que el chasis debe haber tomado un camino más rápido desde el punto de partida hasta estación $S_{2,j-1}$. El razonamiento es el mismo: si hubiera una forma más rápida de pasar por la estación $S_{2,j-1}$, podríamos sustituir esta forma más rápida para ceder una forma más rápida a través de la estación $S_{1,j}$, que sería una contradicción.

En términos más generales, podemos decir que para la programación de la línea de montaje, una solución óptima para un problema (encontrar el camino más rápido a través de la estación $S_{i,j}$) contiene dentro de él una solución óptima para subproblemas (encontrar el camino más rápido a través de $S_{1,j-1}$ o $S_{2,j-1}$). Nos referimos a esta propiedad como *subestructura óptima* , y es uno de los sellos distintivos de la aplicabilidad de dinámica programación, como veremos en el apartado 15.3 .

Usamos una subestructura óptima para demostrar que podemos construir una solución óptima a un problema. desde soluciones óptimas hasta subproblemas. Para la programación de la línea de montaje, razonamos de la siguiente manera. Si miramos una forma más rápida a través de la estación $S_{1,j}$, debe pasar por la estación j - 1 en cualquier línea 1 o la línea 2. Por lo tanto, el camino más rápido a través de la estación $S_{1,j}$ es

- el camino más rápido a través de la estación $S_{1,j-1}$ y luego directamente a través de la estación $S_{1,j}$, o
- la forma más rápida a través de la estación $S_{2,j-1}$, una transferencia de la línea 2 a la línea 1, y luego a través de estación $S_{1,j}$.

Usando el razonamiento simétrico, el camino más rápido a través de la estación $S_{2,j}$ es

- la forma más rápida a través de la estación $S_{2,j-1}$ y luego directamente a través de la estación $S_{2,j}$, o
- la forma más rápida a través de la estación $S_{1,j-1}$, una transferencia de la línea 1 a la línea 2, y luego a través de estación $S_{2,j}$.

Para resolver el problema de encontrar el camino más rápido a través de la estación j de cualquier línea, resolvemos el subproblemas de encontrar las vías más rápidas a través de la estación j - 1 en ambas líneas.

Por lo tanto, podemos construir una solución óptima para una instancia de la programación de la línea de montaje. problema mediante la construcción de soluciones óptimas a los subproblemas.

Paso 2: una solución recursiva

El segundo paso del paradigma de programación dinámica es definir el valor de un óptimo solución recursiva en términos de las soluciones óptimas a los subproblemas. Para la línea de montaje problema de programación, elegimos como nuestros subproblemas los problemas de encontrar la manera más rápida a través de la estación j en ambas líneas, para $j = 1, 2, ..., n$. Sea $f_i[j]$ el tiempo más rápido posible para obtener un chasis desde el punto de partida a través de la estación $S_{i,j}$.

Nuestro objetivo final es determinar el tiempo más rápido para obtener un chasis en todo el fábrica, que denotamos por f^*. El chasis tiene que atravesar completamente la estación n en línea 1 o línea 2 y luego a la salida de fábrica. Dado que la más rápida de estas formas es la más rápida a través de toda la fábrica, tenemos

(15,1)

También es fácil razonar sobre $f_1[1]$ y $f_2[1]$. Para pasar por la estación 1 en cualquier línea, un chasis simplemente va directamente a esa estación. Así,

Página 281

(15,2)

(15,3)

Ahora consideremos cómo calcular $f_i[j]$ para $j = 2, 3, ..., n$ ($yi = 1, 2$). Centrándonos en $f_1[j]$, recuerde que el camino más rápido a través de la estación $S_{1,j}$ es el camino más rápido a través de la estación $S_{1,j-1}$ y luego directamente a través de la estación $S_{1,j}$, o la forma más rápida a través de la estación $S_{2,j-1}$, una transferencia desde la línea 2 a la línea 1, y luego por la estación $S_{1,j}$. En el primer caso, tenemos $f_1[j] = f_1[j - 1] + a_{1,j}$, y en el último caso, $f_1[j] = f_2[j - 1] + t_{2,j-1} + a_{1,j}$. Así,

(15,4)

para $j = 2, 3, ..., n$. Simétricamente, tenemos

(15,5)

para $j = 2, 3, ..., n$. Combinando las ecuaciones (15.2) - (15.5), obtenemos las ecuaciones recursivas

(15,6)

(15,7)

La figura 15.2 (b) muestra los valores de $f_i[j]$ para el ejemplo del inciso a), calculados mediante ecuaciones. (15.6) y (15.7) , junto con el valor de f^*.

Los valores de $f_i[j]$ dan los valores de las soluciones óptimas a los subproblemas. Para ayudarnos a realizar un seguimiento de cómo construir una solución óptima, definamos $l_i[j]$ como el número de línea, 1 o 2, cuyo la estación j - 1 se usa de manera más rápida a través de la estación $S_{i,j}$. Aquí, i = 1, 2 y j = 2, 3, ..., n . (Nosotros evite definir $l_i[1]$ porque ninguna estación precede a la estación 1 en ninguna de las líneas.) También definimos l^* para ser la línea cuya estación n se utilice de forma más rápida por toda la fábrica. Los valores $l_i[j]$ ayúdanos a encontrar el camino más rápido. Usando los valores de l^* y $l_i[j]$ que se muestran en la figura 15.2 (b) , Trace el camino más rápido a través de la fábrica que se muestra en el inciso a) como sigue. Comenzando con l^* = 1, nosotros utilice la estación $S_{1,6}$. Ahora miramos $l_1[6]$, que es 2, por lo que usamos la estación $S_{2,5}$. Continuando, nosotros mire $l_2[5] = 2$ (use la estación $S_{2,4}$), $l_2[4] = 1$ (estación $S_{1,3}$), $l_1[3] = 2$ (estación $S_{2,2}$), y $l_2[2] = 1$ (estación $S_{1,1}$).

Paso 3: calcular los tiempos más rápidos

En este punto, sería muy sencillo escribir un algoritmo recursivo basado en la ecuación (15.1) y las recurrencias (15.6) y (15.7) para calcular el camino más rápido a través de la fábrica. Existe un problema con un algoritmo tan recursivo: su tiempo de ejecución es exponencial en n . A vea por qué, sea $r_i(j)$ el número de referencias hechas a $f_i[j]$ en un algoritmo recursivo. De ecuación (15.1) , tenemos

(15,8)

De las recurrencias (15.6) y (15.7), tenemos

(15,9)

Página 282

para j = 1, 2, ..., n - 1. Como el ejercicio 15.1-2 le pide que muestre, $r_i(j) = 2^{n-j}$. Por tanto, $f_1[1]$ solo es referenciado 2^{n-1} veces! Como el ejercicio 15.1-3 le pide que muestre, el número total de referencias a todos los valores de $f_i[j]$ son $\Theta(2^n)$.

Podemos hacerlo mucho mejor si calculamos los valores de $f_i[j]$ en un orden diferente al recursivo camino. Observe que para $j \geq 2$, cada valor de $f_i[j]$ depende solo de los valores de $f_1[j - 1]$ y $f_2[j - 1]$. Calculando los valores de $f_i[j]$ en orden *creciente* de números de estación j, *de* izquierda a derecha en Figura 15.2 (b): podemos calcular de la manera más rápida a través de la fábrica, y el tiempo que lleva, en $\Theta(n)$ tiempo. El procedimiento FASTEST-WAY toma como entrada los valores $a_{i,j}$, $t_{i,j}$, e_i , y x_i , también como n , el número de estaciones en cada línea de montaje.

```
VÍA MÁS RÁPIDA ( a , t , e , x , n )
  1 f1 [1] ← e1 + a1,1
  2 f2 [1] ← e2 + un2,1
  3 para j ← 2 an
  4 hacer si f1 [j - 1] + a1,j ≤ f2 [j - 1] + t2,j-1 + a1,j
  5                          entonces f1 [j ] ← f1 [j - 1] + a1,j
  6                                  l1 [j ] ← 1
  7                          más f1 [j ] ← f2 [j - 1] + t2,j-1 + a1,j
  8                                  l1 [j ] ← 2
  9                    si f2 [j - 1] + a2,j ≤ f1 [j - 1] + t1,j-1 + a2,j
10                          entonces f2 [j ] ← f2 [j - 1] + a2 , j
11                                  l2 [j ] ← 2
12                          si no f2 [j ] ∞ f1 [j - 1] + t1,j-1 + a2,j
13                                  l2 [j ] ← 1
14 si f1 [ n ] + x1 ≤ f2 [ n ] + x2
15 entonces f* = f1 [ n ] + x1
dieciséis                l* = 1
17 más f* = f2 [ n ] + x2
18                          l* = 2
```

FASTEST-WAY funciona de la siguiente manera. Las líneas 1–2 calculan $f_1[1]$ y $f_2[1]$ usando las ecuaciones (15.2) y (15,3). Entonces el ciclo **for** de las líneas 3-13 calcula $f_i[j]$ y $l_i[j]$ para i = 1, 2 y j = 2, 3, ..., n . Las líneas 4-8 calculan $f_1[j]$ y $l_1[j]$ usando la ecuación (15.4) , y las líneas 9-13 calculan $f_2[j]$ y $l_2[j]$ utilizando la ecuación (15.5) . Finalmente, las líneas 14-18 calculan f^* y l^* usando la ecuación (15.1) . Debido a que las líneas 1–2 y 14–18 toman un tiempo constante y cada una de las n - 1 iteraciones del ciclo **for** de las líneas 3-13 toma un tiempo constante, todo el procedimiento toma $\Theta(n)$ tiempo.

Una forma de ver el proceso de calcular los valores de $f_i[j]$ y $l_i[j]$ es que estamos completando

entradas de la tabla. Con referencia a la figura 15.2 (b), llenamos tablas que contienen valores $f_i[j]$ y $l_i[j]$ de izquierda a derecha (y de arriba hacia abajo dentro de cada columna). Para completar una entrada $f_i[j]$, necesitamos el valores de $f_1[j-1]$ y $f_2[j-1]$ y, sabiendo que ya los hemos calculado y almacenado, determinamos estos valores simplemente buscándolos en la tabla.

Paso 4: Construyendo el camino más rápido a través de la fábrica

Habiendo calculado los valores de $f_i[j]$, f^*, $l_i[j]$ y l^*, necesitamos construir la secuencia de estaciones utilizadas de la manera más rápida a través de la fábrica. Discutimos anteriormente cómo hacerlo en el ejemplo de la Figura 15.2 .

El siguiente procedimiento imprime las estaciones utilizadas, en orden decreciente de número de estación. El ejercicio 15.1-1 le pide que lo modifique para imprimirlos en orden creciente de número de estación.

ESTACIONES DE IMPRESIÓN (l , n)

Página 283

```
1 yo ← l *
2 imprime "línea" i ", estación" n
3 para j ← n hacia abajo 2
4         hacer i ← l i [ j ]
5             imprimir "línea" i ", estación" j - 1
```

En el ejemplo de la Figura 15.2 , PRINT-STATIONS produciría la salida

línea 1, estación 6

línea 2, estación 5

línea 2, estación 4

línea 1, estación 3

línea 2, estación 2

línea 1, estación 1

Ejercicios 15.1-1

Muestre cómo modificar el procedimiento PRINT-STATIONS para imprimir las estaciones en aumento orden del número de estación. (*Sugerencia:* utilice la recursividad).

Ejercicios 15.1-2

Utilice las ecuaciones (15.8) y (15.9) y el método de sustitución para mostrar que $r_i(j)$, el número de las referencias hechas a $f_i[j]$ en un algoritmo recursivo, es igual a 2^{n-j} .

Ejercicios 15.1-3

Usando el resultado del ejercicio 15.1-2 , demuestre que el número total de referencias a todos los valores de $f_i[j]$, o , es exactamente $2^{n+1} - 2$.

Ejercicios 15.1-4

Juntas, las tablas que contienen los valores $f_i[j]$ y $l_i[j]$ contienen un total de $4n - 2$ entradas. Enseña como para reducir los requisitos de espacio a un total de $2n + 2$ entradas, mientras se sigue calculando f^* y todavía pudiendo imprimir todas las estaciones de forma más rápida a través de la fábrica.

Página 284

Ejercicios 15.1-5

El profesor Canty conjetura que podrían existir algunos valores e_i, $a_{i,j}$ y $t_{i,j}$ para los cuales FASTEST-WAY produce valores $l_i[j] = 2$ y $l_2[j] = 1$ para algún número de estación j. Suponiendo que todos los costos de transferencia $t_{i,j}$ no son negativos, demuestre que el profesor está equivocado.

15.2 Multiplicación de cadenas de matrices

Nuestro siguiente ejemplo de programación dinámica es un algoritmo que resuelve el problema de la matriz multiplicación en cadena. Se nos da una secuencia (cadena) $A_1, A_2, ..., A_n$ de n matrices para ser multiplicado, y deseamos calcular el producto

(15,10)

Podemos evaluar la expresión (15.10) usando el algoritmo estándar para multiplicar pares de matrices como una subrutina una vez que la hemos entre paréntesis para resolver todas las ambigüedades en las matrices se multiplican juntas. Un producto de matrices está *completamente entre paréntesis* si es un matriz única o el producto de dos productos de matriz completamente entre paréntesis, rodeados por paréntesis. La multiplicación de matrices es asociativa, por lo que todas las paréntesis dan el mismo producto. Por ejemplo, si la cadena de matrices es A_1, A_2, A_3, A_4, el producto $A_1A_2A_3A_4$ puede estar completamente entre paréntesis de cinco formas distintas:

$(A_1(A_2(A_3A_4)))$,

$(A_1((A_2A_3)A_4))$,

$((A_1A_2)(A_3A_4))$,

$((A_1(A_2A_3))A_4)$,

$(((A_1A_2)A_3)A_4)$.

La forma en que ponemos entre paréntesis una cadena de matrices puede tener un impacto dramático en el costo de evaluar el producto. Considere primero el costo de multiplicar dos matrices. El estandar
El algoritmo viene dado por el siguiente pseudocódigo. Las *filas* y *columnas de* atributos son las números de filas y columnas en una matriz.

```
MATRIZ-MULTIPLICAR ( A , B )
1 si columnas [ A ] ≠ filas [ B ]
2 luego error "dimensiones incompatibles"
3 más para i ← 1 a las filas [ A ]
4                   hacer para j ← 1 a las columnas [ B ]
5                                 hacer C [ i , j ] ← 0
6                                       para k ← 1 a las columnas [ A ]
7                                             hacer C [ i , j ] ← C [ i , j ] + A [ i , k ] · B [ k , j ]
8           volver C
```

Página 285

Podemos multiplicar dos matrices A y B solo si son *compatibles*: el número de columnas de A debe ser igual al número de filas de B. Si A es una matriz $p \times q$ y B es una matriz $q \times r$, el la matriz C resultante es una matriz $p \times r$. El tiempo para calcular C está dominado por el número de multiplicaciones escalares en la línea 7, que es pqr. A continuación, expresaremos los costos en términos

del número de multiplicaciones escalares.

Para ilustrar los diferentes costos incurridos por diferentes paréntesis de un producto de matriz, considere el problema de una cadena A_1, A_2, A_3 de tres matrices. Suponga que las dimensiones de las matrices son 10 × 100, 100 × 5 y 5 × 50, respectivamente. Si multiplicamos según el entre paréntesis $((A_1A_2)A_3)$, realizamos $10 \cdot 100 \cdot 5 = 5000$ multiplicaciones escalares para calcular el producto matricial de 10 × 5 A_1A_2, más otro $10 \cdot 5 \cdot 50 = 2500$ escalar multiplicaciones para multiplicar esta matriz por A_3, para un total de 7500 multiplicaciones escalares. Si en lugar de eso multiplicamos de acuerdo con el paréntesis $(A_1(A_2A_3))$, realizamos $100 \cdot 5 \cdot 50 =$ 25.000 multiplicaciones escalares para calcular el producto matricial de 100 × 50 A_2A_3, más otros $10 \cdot 100 \cdot 50 = 50{,}000$ multiplicaciones escalares para multiplicar A_1 por esta matriz, para un total de 75,000 multiplicaciones escalares. Por lo tanto, calcular el producto de acuerdo con el primer paréntesis es 10 veces más rápido.

El problema de multiplicación matriz-cadena se puede plantear de la siguiente manera: dada una cadena $A_1, A_2, ..., A_n$ de n matrices, donde para $i = 1, 2, ..., n$, la matriz A_i tiene dimensión $p_{i-1} \times p_i$, completamente entre paréntesis el producto $A_1A_2A_n$ de una manera que minimice el número de escalares multiplicaciones.

Tenga en cuenta que en el problema de multiplicación de cadenas de matrices, en realidad no estamos multiplicando matrices. Nuestro objetivo es solo determinar un orden para multiplicar matrices que tenga el menor costo. Normalmente, el tiempo invertido en determinar este pedido óptimo es más que pagado por el tiempo que se ahorra más adelante cuando se realizan las multiplicaciones de matrices (como realizando solo 7500 multiplicaciones escalares en lugar de 75,000).

Contando el número de paréntesis

Antes de resolver el problema de multiplicación de cadenas de matrices mediante programación dinámica, convencernos de que la comprobación exhaustiva de todos los posibles paréntesis no produce un algoritmo eficiente. Denote el número de paréntesis alternativos de una secuencia de n matrices por $P(n)$. Cuando $n = 1$, solo hay una matriz y, por lo tanto, solo una forma de entre paréntesis el producto de la matriz. Cuando $n \geq 2$, un producto de matriz completamente entre paréntesis es el producto de dos subproductos de matriz completamente entre paréntesis, y la división entre los dos los subproductos pueden ocurrir entre las matrices k ésima y $(k + 1)$ st para cualquier $k = 1, 2, ..., n - 1$. Por lo tanto, obtenemos la recurrencia

(15,11)

El problema 12-4 le pidió que mostrara que la solución a una recurrencia similar es la secuencia de Números catalanes, que crece como $\Omega(4^n / n^{3/2})$. Un ejercicio más simple (vea el ejercicio 15.2-3) es demuestre que la solución a la recurrencia (15.11) es $\Omega(2^n)$. El número de soluciones es entonces exponencial en n, y el método de fuerza bruta de búsqueda exhaustiva es por lo tanto una mala estrategia para determinar el paréntesis óptimo de una cadena de matriz.

Página 286

Paso 1: la estructura de un paréntesis óptimo

Nuestro primer paso en el paradigma de la programación dinámica es encontrar la subestructura óptima y luego úselo para construir una solución óptima al problema a partir de soluciones óptimas para subproblemas. Para el problema de multiplicación de cadenas de matrices, podemos realizar este paso como sigue. Por conveniencia, adoptemos la notación A_{ij}, donde $i \leq j$, para la matriz que resulta a partir de la evaluación del producto $A_iA_{i+1}A_j$. Observe que si el problema no es trivial, es decir, $i < j$, entonces cualquier paréntesis del producto $A_iA_{i+1}A_j$ debe dividir el producto entre A_k y A_{k+1} para algún entero k en el rango $i \leq k < j$. Es decir, para algún valor de k, primero calculamos el matrices A_{ik} y A_{k+1j} y luego multiplíquelas para producir el producto final A_{ij}. los El costo de este paréntesis es, por lo tanto, el costo de calcular la matriz A_{ik}, más el costo de calcular A_{k+1j}, más el costo de multiplicarlos.

La subestructura óptima de este problema es la siguiente. Supongamos que un óptimo el paréntesis de $A_iA_{i+1}A_j$ divide el producto entre A_k y A_{k+1}. Entonces el paréntesis de la subcadena "prefijo" $A_iA_{i+1}A_k$ dentro de este paréntesis óptimo de $A_iA_{i+1}A_j$ debe ser un paréntesis óptima de $A_iA_{i+1}A_k$. ¿Por qué? Si hubiera una forma menos costosa de poner entre paréntesis $A_iA_{i+1}A_k$, sustituyendo ese paréntesis en el paréntesis óptimo de $A_iA_{i+1}A_j$ sería producir otro paréntesis de $A_iA_{i+1}A_j$ cuyo costo fue menor que el óptimo: a

contradicción. Una observación similar es válida para el paréntesis de la subcadena $A_{k+1}A_{k+2}A_j$ en el paréntesis óptimo de $A_iA_{i+1}A_j$: debe ser un paréntesis óptimo de $A_{k+1}A_{k+2}A_j$.

Ahora usamos nuestra subestructura óptima para demostrar que podemos construir una solución óptima para problema desde soluciones óptimas hasta subproblemas. Hemos visto que cualquier solución a un Una instancia no trivial del problema de multiplicación de cadenas de matrices nos obliga a dividir el producto, y que cualquier solución óptima contiene en su interior soluciones óptimas para casos de subproblemas. Por lo tanto, podemos construir una solución óptima para una instancia de la multiplicación de la cadena de la matriz problema dividiendo el problema en dos subproblemas (óptimamente entre paréntesis $A_iA_{i+1}A_k$ y $A_{k+1}A_{k+2}A_j$), encontrando soluciones óptimas para instancias de subproblemas y luego combinando estas soluciones óptimas de subproblemas. Debemos asegurarnos de que cuando busquemos el lugar correcto para dividir el producto, hemos considerado todos los lugares posibles para estar seguros de tener examinó el óptimo.

Paso 2: una solución recursiva

A continuación, definimos el costo de una solución óptima de forma recursiva en términos de las soluciones óptimas para subproblemas. Para el problema de multiplicación de cadenas de matrices, elegimos como nuestros subproblemas el problemas de determinar el costo mínimo de un paréntesis de $A_iA_{i+1}A_j$ para $1 \le i \le j \le n$. Sea $m[i, j]$ el número mínimo de multiplicaciones escalares necesarias para calcular la matriz A_{ij}; para el problema completo, el costo de una forma más barata de calcular A_{1n} sería $m[1, n]$.

Podemos definir $m[i, j]$ recursivamente como sigue. Si $i = j$, el problema es trivial; la cadena consiste de una sola matriz $A_{ii} = A_i$, de modo que no se necesitan multiplicaciones escalares para calcular el producto. Por lo tanto, $m[i, i] = 0$ para $i = 1, 2, ..., n$. Para calcular $m[i, j]$ cuando $i < j$, aprovechamos de la estructura de una solución óptima del paso 1. Supongamos que el óptimo el paréntesis divide el producto $A_iA_{i+1}A_j$ entre A_k y A_{k+1}, donde $i \le k < j$. Entonces, $m[i, j]$ es igual al costo mínimo para calcular los subproductos A_{ik} y A_{k+1j}, más el costo de multiplicando estas dos matrices juntas. Recordando que cada matriz A_i es $p_{i-1} \times p_i$, vemos que

Página 287

calcular el producto matricial $A_{ik}A_{k+1j}$ toma $p_{i-1}p_kp_j$ multiplicaciones escalares. Así, nosotros obtener

$$metro[yo, j] = metro[yo, k] + metro[k + 1, j] + p_{yo-1}p_kp_j.$$

Esta ecuación recursiva supone que conocemos el valor de k, lo que no sabemos. Solo hay $j - i$ valores posibles para k, sin embargo, a saber, $k = i, i + 1, ..., j - 1$. Dado que el óptimo el paréntesis debe usar uno de estos valores para k, solo necesitamos verificarlos todos para encontrar el mejor. Por lo tanto, nuestra definición recursiva del costo mínimo de poner entre paréntesis el producto A_iA_{i+1} *Una* $_{jse}$ convierte en

(15,12)

Los valores $m[i, j]$ dan los costos de las soluciones óptimas a los subproblemas. Para ayudarnos a realizar un seguimiento de cómo construir una solución óptima, definamos $s[i, j]$ como un valor de k en el que podamos divida el producto $A_iA_{i+1}A_j$ para obtener un paréntesis óptimo. Es decir, $s[i, j]$ es igual a un valor k tal que $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$.

Paso 3: Calcular los costos óptimos

En este punto, es muy sencillo escribir un algoritmo recursivo basado en la recurrencia (15.12) para calcular el costo mínimo $m[1, n]$ para multiplicar $A_1A_2A_n$. Como veremos en la sección 15.3 , sin embargo, este algoritmo toma un tiempo exponencial, que no es mejor que la fuerza bruta método de comprobar cada forma de poner entre paréntesis el producto.

La observación importante que podemos hacer en este punto es que tenemos relativamente pocos subproblemas: un problema para cada elección de i y j *que* satisfaga $1 \le i \le j \le n$, o en todas. Un algoritmo recursivo puede encontrar cada subproblema muchas veces en diferentes ramas. de su árbol de recursividad. Esta propiedad de subproblemas superpuestos es el segundo sello distintivo de la aplicabilidad de la programación dinámica (el primer sello es la subestructura óptima).

En lugar de calcular la solución a la recurrencia (15.12) de forma recursiva, realizamos el tercer paso

del paradigma de programación dinámica y calcular el costo óptimo utilizando un tabular, enfoque de abajo hacia arriba. El siguiente pseudocódigo asume que la matriz A_i tiene dimensiones $p_{i-1} \times p_i$ para $i = 1, 2, ..., n$. La entrada es una secuencia $p = p_0, p_1, ..., p_n$, donde $longitud[p] = n + 1$. El El procedimiento utiliza una tabla auxiliar $m[1..n, 1..n]$ para almacenar los costos $m[i, j]$ y un auxiliar tabla $s[1..n, 1..n]$ que registra qué índice de k logró el costo óptimo en la computación $m[i, j]$. Vamos a utilizar la tabla *es* para construir una solución óptima.

Para implementar correctamente el enfoque de abajo hacia arriba, debemos determinar qué entradas de las tablas se utilizan para calcular $m[i, j]$. La ecuación (15.12) muestra que el costo $m[i, j]$ de calcular un producto de cadena de matriz de matrices $j - i + 1$ depende solo de los costos de computación productos de cadena de matriz de menos de $j - i + 1$ matrices. Es decir, para $k = i, i + 1, ..., j - 1$, el matriz $A_{i..k}$ es un producto de $k - i + 1 < j - i + 1$ matrices y la matriz $A_{k+1..j}$ es un producto de $j - k < j - i + 1$ matrices. Por lo tanto, el algoritmo debe completar la tabla m de manera que corresponde a resolver el problema de los paréntesis en cadenas de matrices de longitud creciente.

```
ORDEN-MATRIZ-CADENA ( p )
 1 n ← longitud [ p ] - 1
```

Página 288

```
 2 para i ← 1 a n
 3 hacer m [ i , i ] ← 0
 4 para l ← 2 a n                    ▸ l es la longitud de la cadena.
 5 hacer para i ← 1 a n - l + 1
 6                    hacer j ← i + l - 1
 7                        m [ i , j ] ← ∞
 8                        para k ← i a j - 1
 9                            hacer q ← m [ i , k ] + m [ k + 1, j ] + p_{i-1} p_k p_j
10                                si q < m [ i , j ]
11                                    entonces m [ i , j ] ← q
12                                             s [ i , j ] ← k
13 devuelve m y s
```

El algoritmo primero calcula $m[i, i] \leftarrow 0$ para $i = 1, 2, ..., n$ (los costos mínimos para cadenas de longitud 1) en las líneas 2-3. Luego usa la recurrencia (15.12) para calcular $m[i, i + 1]$ para $i = 1, 2, ..., n - 1$ (los costos mínimos para cadenas de longitud $l = 2$) durante la primera ejecución del bucle en líneas 4–12. La segunda vez a través del bucle, calcula $m[i, i + 2]$ para $i = 1, 2, ..., n - 2$ (el costos mínimos para cadenas de longitud $l = 3$), y así sucesivamente. En cada paso, el $m[i, j]$ cuesta calculado en las líneas 9-12 depende solo de las entradas de la tabla $m[i, k]$ y $m[k + 1, j]$ ya calculado.

La figura 15.3 ilustra este procedimiento en una cadena de $n = 6$ matrices. Como hemos definido $m[i, j]$ sólo para $i \le j$, sólo se utiliza la parte de la tabla m estrictamente por encima de la diagonal principal. los La figura muestra la mesa girada para hacer que la diagonal principal corra horizontalmente. La cadena de la matriz es enumerados en la parte inferior. Con este diseño, el costo mínimo $m[i, j]$ para multiplicar una subcadena $A_i A_{i+1} A_j$ de matrices se puede encontrar en la intersección de las líneas que van al noreste de A_i y noroeste de A_j . Cada fila horizontal de la tabla contiene las entradas para las cadenas matriciales de la misma longitud. MATRIX-CHAIN-ORDER calcula las filas de abajo hacia arriba y desde de izquierda a derecha dentro de cada fila. Una entrada $m[i, j]$ se calcula utilizando los productos $p_{i-1} p_k p_j$ para $k = i, i + 1, ..., j - 1$ y todas las entradas suroeste y sureste de $m[i, j]$.

Figura 15.3: El m y s tablas calculadas por MATRIX-CADENA orden para $n = 6$ y el siguientes dimensiones de matriz:

dimensión de la matriz

A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Página 289

Las mesas se rotan para que la diagonal principal se extienda horizontalmente. Solo la diagonal principal y el triángulo superior se usa en la tabla m , y solo el triángulo superior se usa en la tabla s . los El número mínimo de multiplicaciones escalares para multiplicar las 6 matrices es $m[1, 6] = 15,125$. De las entradas más oscuras, los pares que tienen el mismo sombreado se toman juntos en la línea 9 cuando informática

Una simple inspección de la estructura de bucle anidado de MATRIX-CHAIN-ORDER produce una tiempo de ejecución de $O(n^3)$ para el algoritmo. Los bucles están anidados en tres de profundidad, y cada índice de bucle (l , i y k) toma como máximo n -1 valores. El ejercicio 15.2-4 le pide que demuestre que la carrera El tiempo de este algoritmo es también $\Omega(n^3)$. El algoritmo requiere un espacio $\Theta(n^2)$ para almacenar el m y tablas de s . Por lo tanto, MATRIX-CHAIN-ORDER es mucho más eficiente que el exponencial- método de tiempo de enumerar todos los posibles paréntesis y comprobar cada uno.

Paso 4: construir una solución óptima

Aunque MATRIX-CHAIN-ORDER determina el número óptimo de escalares multiplicaciones necesarias para calcular un producto de cadena de matriz, no muestra directamente cómo multiplica las matrices. No es difícil construir una solución óptima a partir del cálculo información almacenada en la tabla $s[1\ n\ ,\ 1\ n\ ]$. Cada entrada $s[\ i\ ,j\ ]$ registra el valor de k tal que el paréntesis óptimo de $A_i A_{i+1} \cdots A_j$ divide el producto entre A_k y A_{k+1} . Así, sabemos que la multiplicación de matrices final en el cálculo óptimo de $A_{1\,n}$ es $A_{1\,s[1,\,n\,]} A_{s[1,\,n\,]+1\,n}$. Las multiplicaciones de matrices anteriores se pueden calcular de forma recursiva, ya que $s[1, s[1, n\ ]]$ determina la última multiplicación de matrices en el cálculo de $A_{1\,s[1,\,n\,]}$, y $s[\ s[1, n\ ] + 1, n\ ]$ determina la última multiplicación de matrices en la computación $A_{s[1,\,n\,]+1\,n}$. El siguiente procedimiento recursivo imprime un paréntesis óptima de A_i , A_{i+1} , ..., A_j , dada la tabla s calculada por MATRIX-CHAIN-ORDER y los índices i y j . La llamada inicial PRINT-OPTIMAL-PARENS (s , 1, n) imprime un paréntesis óptimo de A_1 , A_2 , ..., A_n .

```
IMPRESIÓN-OPTIMAL-PARENS ( s , i , j )
1 si i = j
2 luego imprima " A " i
3 más imprimir "("
4                   IMPRESIÓN-OPTIMAL-PARENS ( s , i , s [ i , j ])
5                   IMPRESIÓN-OPTIMAL-PARENS ( s , s [ i , j ] + 1, j )
6                   impresión ")"
```

En el ejemplo de la Figura 15.3 , la llamada PRINT-OPTIMAL-PARENS (s , 1, 6) imprime el entre paréntesis $((A_1 (A_2 A_3)) ((A_4 A_5) A_6))$.

Ejercicios 15.2-1

Encuentre un paréntesis óptimo de un producto de cadena-matriz cuya secuencia de dimensiones es 5, 10, 3, 12, 5, 50, 6.

Página 290

Ejercicios 15.2-2

Dar un algoritmo recursivo MATRIX-CHAIN-MULTIPLY (A , s , i , j) que realmente realiza la multiplicación de cadena-matriz óptima, dada la secuencia de matrices A_1 , A_2 , ..., A_n , el s calculada por MATRIX-CHAIN-ORDER, y los índices i y j . (La llamada inicial sería MATRIX-CHAIN-MULTIPLY (A , s , 1, n).)

Ejercicios 15.2-3

Utilice el método de sustitución para demostrar que la solución de la recurrencia (15.11) es $\Omega(2^n)$.

Ejercicios 15.2-4

Sea $R(i, j)$ el número de veces que se hace referencia a la entrada de la tabla $m[i, j]$ mientras se calcula otra entradas de la tabla en una llamada de MATRIX-CHAIN-ORDER. Demuestre que el número total de referencias porque toda la mesa es

(*Sugerencia: la* ecuación (A.3) puede resultarle útil).

Ejercicios 15.2-5

Demuestre que un paréntesis completo de una expresión de n elementos tiene exactamente $n - 1$ pares de paréntesis.

15.3 Elementos de programación dinámica

Aunque acabamos de trabajar con dos ejemplos del método de programación dinámica, es posible que todavía se pregunte cuándo se aplica el método. Desde una perspectiva de ingeniería, ¿Cuándo deberíamos buscar una solución de programación dinámica a un problema? En esta sección, Examinar los dos ingredientes clave que debe tener un problema de optimización para una dinámica programación para ser aplicable: subestructura óptima y subproblemas superpuestos. Nosotros también mire un método variante, llamado memorización, [1] para aprovechar la superposición-propiedad de subproblemas.

Página 291

Subestructura óptima

El primer paso para resolver un problema de optimización mediante programación dinámica es caracterizar la estructura de una solución óptima. Recuerde que un problema presenta una subestructura óptima si La solución óptima al problema contiene en su interior soluciones óptimas a los subproblemas. Siempre que un problema exhiba una subestructura óptima, es una buena pista que la dinámica puede aplicarse la programación. (Sin embargo, también podría significar que se aplica una estrategia codiciosa. Consulte Capítulo 16.) En programación dinámica, construimos una solución óptima al problema desde soluciones óptimas a subproblemas. En consecuencia, debemos tener cuidado de asegurar que la gama de Los subproblemas que consideramos incluyen los que se utilizan en una solución óptima.

Descubrimos la subestructura óptima en los dos problemas que hemos examinado en este capítulo. hasta aquí. En la Sección 15.1 , observamos que el camino más rápido a través de la estación j de cualquier línea contenida dentro de la vía más rápida a través de la estación j - 1 en una línea. En la Sección 15.2 , observó que un paréntesis óptimo de $A_i A_{i+1} A_j$ que divide el producto entre A_k y A_{k+1} contiene en su interior soluciones óptimas a los problemas de entre paréntesis $A_i A_{i+1} A_k$ y $A_{k+1} A_{k+2} A_j$.

Se encontrará siguiendo un patrón común para descubrir la subestructura óptima:

1. Muestra que una solución al problema consiste en tomar una decisión, como elegir una estación de la línea de montaje anterior o elegir un índice en el que dividir la matriz cadena. Hacer esta elección deja uno o más subproblemas por resolver.
2. Supone que para un problema dado, se le da la opción que conduce a una solucion optima. Todavía no se preocupa por cómo determinar esta elección. Simplemente asume que se le ha dado.
3. Dada esta opción, usted determina qué subproblemas surgen y cuál es la mejor manera de caracterizar el espacio resultante de subproblemas.
4. Demuestra que las soluciones a los subproblemas utilizadas dentro de la solución óptima al El problema en sí mismo debe ser óptimo mediante el uso de una técnica de "cortar y pegar". Tu lo haces suponiendo que cada una de las soluciones del subproblema no es óptima y luego derivando una contradicción. En particular, al "eliminar" la solución del subproblema no óptima y "pegando" el óptimo, demuestra que puede obtener una mejor solución al original problema, contradiciendo así su suposición de que ya tenía una solución óptima. Si hay más de un subproblema, normalmente son tan similares que el corte y El argumento de pegar para uno se puede modificar para los demás con poco esfuerzo.

Para caracterizar el espacio de los subproblemas, una buena regla general es tratar de mantener el espacio como simple como sea posible, y luego expandirlo según sea necesario. Por ejemplo, el espacio de subproblemas que consideramos para la programación de la línea de montaje era la forma más rápida de entrar en el fábrica a través de las estaciones $S_{1,j}$ y $S_{2,j}$. Este espacio de subproblemas funcionó bien y no hubo Necesito probar un espacio más general de subproblemas.

Por el contrario, suponga que hemos intentado restringir nuestro espacio de subproblemas para la cadena-matriz multiplicación a productos matriciales de la forma $A_1 A_2 A_j$. Como antes, un paréntesis óptimo debe dividir este producto entre A_k y A_{k+1} para algunos $1 \le k \le j$. A menos que podamos garantizar que k siempre es igual a j - 1, encontraríamos que tenemos subproblemas de la forma $A_1 A_2 A_k$ y $A_{k+1} A_{k+2} A_j$, y que el último subproblema no es de la forma $A_1 A_2 A_j$. Para este problema, fue necesario para permitir que nuestros subproblemas varíen en "ambos extremos", es decir, para permitir que tanto i como j varíen en el subproblema $A_i A_{i+1} A_j$.

Página 292

La subestructura óptima varía entre los dominios del problema de dos maneras:

1. cuántos subproblemas se utilizan en una solución óptima al problema original, y
2. cuántas opciones tenemos para determinar qué subproblema (s) utilizar en un óptimo solución.

En la programación de la línea de montaje, una solución óptima utiliza solo un subproblema, pero debemos considere dos opciones para determinar una solución óptima. Para encontrar la forma más rápida través de la estación $S_{i,j}$, utilizamos *ya sea* la manera más rápida a través de $S_{1,j-1}$ *o* la manera más rápida a través de $S_{2,j-1}$; lo que usemos representa el único subproblema que debemos resolver de manera óptima. Matriz-Multiplicación de cadena para la subcadena $A_i A_{i+1} A_j$ sirve como ejemplo con dos subproblemas y j - i opciones. Para una matriz dada A_k en la que dividimos el producto, tenemos dos subproblemas - entre paréntesis $A_i A_{i+1} A_k$ y entre paréntesis $A_{k+1} A_{k+2} A_j$ - y debemos resolver *ambos* de forma óptima. Una vez que determinamos las soluciones óptimas a los subproblemas, elegimos de entre j - i candidatos para el índice k.

De manera informal, el tiempo de ejecución de un algoritmo de programación dinámica depende del producto de dos factores: la cantidad de subproblemas en general y la cantidad de opciones que examinamos para cada subproblema. En la programación de la línea de montaje, teníamos $\Theta(n)$ subproblemas en general, y solo dos opciones para examinar para cada uno, lo que arroja un tiempo de ejecución $\Theta(n)$. Para la multiplicación de cadenas de matrices, había $\Theta(n^2)$ subproblemas en general, y en cada uno teníamos como máximo n - 1 opciones, dando un $O(n^3)$ tiempo de ejecución.

La programación dinámica utiliza una subestructura óptima de forma ascendente. Es decir, primero encontramos soluciones óptimas a los subproblemas y, habiendo resuelto los subproblemas, encontramos un óptimo solución al problema. Encontrar una solución óptima al problema implica tomar una decisión entre los subproblemas en cuanto a cuál usaremos para resolver el problema. El costo del problema La solución suele ser los costos del subproblema más un costo que es directamente atribuible a la elección. sí mismo. En la programación de la línea de montaje, por ejemplo, primero resolvimos los subproblemas de encontrar la forma más rápida a través de las estaciones $S_{1,j-1}$ y $S_{2,j-1}$, y luego elegimos una de estas estaciones como el una estación precedente $S_{i,j}$. El costo atribuible a la elección en sí depende de si conmutar líneas entre las estaciones j - 1 y j; este costo es $a_{i,j}$ si permanecemos en la misma línea, y es $t_{i'}$

$_{j-1} + a_{i,j}$, donde $i' \neq i$, si cambiamos. En la multiplicación de cadenas de matrices, determinamos el paréntesis de las subcadenas de $A_i A_{i+1} A_j$, y luego elegimos la matriz A_k en la que dividir el producto. El costo atribuible a la elección en sí es el término $p_{i-1} p_k p_j$.

En el capítulo 16, examinaremos los "algoritmos codiciosos", que tienen muchas similitudes con programación dinámica. En particular, los problemas a los que se aplican los algoritmos codiciosos han subestructura óptima. Una diferencia destacada entre algoritmos codiciosos y dinámicos La programación es que en los algoritmos codiciosos, utilizamos la subestructura óptima de forma descendente. En lugar de encontrar primero soluciones óptimas a los subproblemas y luego tomar una decisión, los codiciosos Los algoritmos primero toman una decisión, la opción que se ve mejor en ese momento, y luego resuelven un subproblema resultante.

Sutilezas

Se debe tener cuidado de no asumir que la subestructura óptima se aplica cuando no es así. Considere los siguientes dos problemas en los que se nos da una gráfica dirigida $G = (V, E)$ y vértices $u, v \in V$.

Página 293

- **Ruta más corta no ponderada:** [2] Encuentre una ruta de u a v que tenga la menor cantidad de bordes. Tal camino debe ser simple, ya que eliminar un ciclo de un camino produce un camino con menos bordes.
- **Ruta simple más larga no ponderada:** encuentre una ruta simple de u a v *que* consta de la mayoría de los bordes. Necesitamos incluir el requisito de simplicidad porque de lo contrario podemos recorrer un ciclo tantas veces como queramos para crear trayectorias con un tamaño arbitrariamente grande número de aristas.

El problema del camino más corto no ponderado presenta una subestructura óptima, como sigue. Suponer que $u \neq v$, por lo que el problema no es trivial. Entonces cualquier camino p de u a v debe contener un vértice intermedio, digamos w. (Tenga en cuenta que w puede ser u o v.) Por lo tanto, podemos descomponer la ruta en subrutas . Claramente, el número de aristas en p es igual a la suma de los número de aristas en p_1 y el número de aristas en p_2. Afirmamos que si p es un óptimo (es decir, la ruta más corta) de u a v, entonces p_1 debe ser la ruta más corta de u a w. ¿Por qué? Usamos un "cut-y-pegar "argumento: si hubiera otra ruta, digamos, desde u hasta w con menos aristas que p_1, luego podríamos cortar p_1 y pegar para producir un trazado con menos aristas que p, contradiciendo así la optimalidad de p. Simétricamente, p_2 debe ser un camino más corto de w a v. Así, podemos encontrar un camino más corto de u a v considerando todos los vértices intermedios w, encontrando un camino más corto de u a w y un camino más corto de w a v, y elegir un vértice intermedio w que produce el camino más corto en general. En la sección 25.2 usamos una variante de esta observación de subestructura óptima para encontrar un camino más corto entre cada par de vértices en un peso, gráfico dirigido.

Es tentador asumir que el problema de encontrar un camino simple más largo sin ponderar también presenta una subestructura óptima. Después de todo, si descomponemos un camino simple más largo en subrutas , Entonces no debe p_1 ser un camino simple más largo de u a w, y no debe p_2 be ¿Un camino simple más largo de w a v? ¡La respuesta es no! La figura 15.4 da un ejemplo. Considerar la ruta $q \to r \to t$, que es la ruta simple más larga de q a t. ¿Es $q \to r$ un camino simple más largo de q a r? No, porque la ruta $q \to s \to t \to r$ es una ruta simple que es más larga. Es $r \to t$ *el* más largo camino simple de r a t? No, de nuevo, porque la ruta $r \to q \to s \to t$ es una ruta simple que es más larga.

Figura 15.4: Un gráfico dirigido que muestra que el problema de encontrar una ruta simple más larga en un El gráfico dirigido no ponderado no tiene una subestructura óptima. La ruta $q \to r \to t$ es una la ruta simple más larga de q a t, pero la subruta $q \to r$ no es una ruta simple más larga de q a r, tampoco es la subruta $r \to t$ una ruta simple más larga de r a t.

Este ejemplo muestra que para las rutas simples más largas, no solo falta una subestructura óptima, pero no necesariamente podemos armar una solución "legal" al problema a partir de soluciones para subproblemas. Si combinamos las trayectorias simples más largas $q \to s \to t \to r$ y $r \to q \to s \to t$, obtener la ruta $q \to s \to t \to r \to q \to s \to t$, que no es simple. De hecho, el problema de encontrar un camino simple más largo sin ponderar no parece tener ningún tipo de subestructura óptima.

Nunca se ha encontrado un algoritmo de programación dinámica eficiente para este problema. De hecho, este problema es NP-completo, lo que, como veremos en el capítulo 34, significa que es poco probable que se puede resolver en tiempo polinomial.

Página 294

¿Qué hay en la subestructura de un camino simple más largo que es tan diferente al de un ¿camino más corto? Aunque se utilizan dos subproblemas en una solución a un problema para los dos y caminos más cortos, los subproblemas para encontrar el camino simple más largo no son independientes, mientras que para los caminos más cortos lo son. ¿Qué entendemos por subproblemas independientes? Queremos decir que la solución de un subproblema no afecta la solución de otro. subproblema del mismo problema. Para el ejemplo de la figura 15.4 , tenemos el problema de encontrar un camino simple más largo de q a t con dos subproblemas: encontrar caminos simples más largos de q a r y de r a t . Para el primero de estos subproblemas, elegimos la ruta $q \rightarrow s \rightarrow t \rightarrow r$, y así también hemos utilizado los vértices s y t . Ya no podemos usar estos vértices en el segundo subproblema, ya que la combinación de las dos soluciones a los subproblemas produciría un camino que no es simple. Si no podemos usar el vértice t en el segundo problema, entonces no podemos resolver todo, ya que se requiere que t esté en el camino que encontramos, y no es el vértice en el que estamos "empalmar" las soluciones del subproblema (ese vértice es r). Nuestro uso de vértices s y t en una solución de subproblema evita que se utilicen en la otra solución de subproblema. Debemos usar al menos uno de ellos para resolver el otro subproblema, sin embargo, y debemos usar ambos para solucionarlo de forma óptima. Por eso decimos que estos subproblemas no son independientes. Visto de otra manera, nuestro uso de recursos para resolver un subproblema (esos recursos son vértices) los ha dejado disponibles para el otro subproblema.

Entonces, ¿por qué los subproblemas son independientes para encontrar el camino más corto? La respuesta es que por naturaleza, los subproblemas no comparten recursos. Afirmamos que si un vértice w está en un ruta p de u a v , entonces podemos empalmar *cualquier* camino más corto y *cualquier* camino más corto para producir un camino más corto de u a v . Se nos asegura que, aparte de w , ningún vértice puede aparecen en ambos caminos p_1 y p_2 . ¿Por qué? Suponga que aparece algún vértice $x \neq w$ tanto en p_1 como en p_2 , de modo que podemos descomponer p_1 como y p_2 como . Por la subestructura óptima de este problema, la trayectoria p tiene tantas aristas como p_1 y p_2 juntas; digamos que p tiene aristas e . Ahora construyamos un camino de u a v . Este camino tiene como máximo e - 2 aristas, que contradice la suposición de que p es un camino más corto. Por tanto, tenemos la seguridad de que los subproblemas porque el problema del camino más corto son independientes.

Ambos problemas examinados en las Secciones 15.1 y 15.2 tienen subproblemas independientes. En matriz Multiplicación de cadenas, los subproblemas son las subcadenas de multiplicación $A_i A_{i+1} A_k$ y $A_{k+1} A_{k+2} A_j$. Estas subcadenas son disjuntas, por lo que ninguna matriz podría incluirse en ambas. En programación de la línea de montaje, para determinar el camino más rápido a través de la estación $S_{i,j}$, observamos el vías más rápidas a través de las estaciones $S_{1,j-1}$ y $S_{2,j-1}$. Porque nuestra solución de la manera más rápida La estación Si, j incluirá solo una de estas soluciones de subproblemas, ese subproblema es automáticamente independiente de todos los demás utilizados en la solución.

Subproblemas superpuestos

El segundo ingrediente que debe tener un problema de optimización para que la programación dinámica ser aplicable es que el espacio de subproblemas debe ser "pequeño" en el sentido de que un recursivo El algoritmo para el problema resuelve los mismos subproblemas una y otra vez, en lugar de siempre generando nuevos subproblemas. Típicamente, el número total de subproblemas distintos es un polinomio en el tamaño de entrada. Cuando un algoritmo recursivo revisa el mismo problema una y otra vez una vez más, decimos que el problema de optimización tiene subproblemas superpuestos . [3] Por el contrario, un problema para el cual es adecuado un enfoque de divide y vencerás, generalmente genera nuevos problemas en cada paso de la recursividad. Los algoritmos de programación dinámica suelen tomar ventaja de la superposición de subproblemas resolviendo cada subproblema una vez y luego almacenando el solución en una tabla donde se puede buscar cuando sea necesario, utilizando un tiempo constante por búsqueda.

Página 295

En la Sección 15.1 , examinamos brevemente cómo una solución recursiva para la programación de la línea de montaje hace 2^{n-j} referencias a $f_i[j]$ para $j = 1, 2, \ldots, n$. Nuestra solución tabular toma un tiempo exponencial algoritmo recursivo hasta tiempo lineal.

Para ilustrar la propiedad de subproblemas superpuestos con mayor detalle, volvamos a examinar la Problema de multiplicación matriz-cadena. Volviendo a la Figura 15.3 , observe que MATRIX-CHAIN-ORDER busca repetidamente la solución a los subproblemas en las filas inferiores al resolver subproblemas en filas superiores. Por ejemplo, se hace referencia a la entrada $m[3, 4]$ 4 veces: durante el cálculos de $m[2, 4]$, $m[1, 4]$, $m[3, 5]$ y $m[3, 6]$. Si se volvieron a calcular $m[3, 4]$ cada vez, en lugar de simplemente mirar hacia arriba, el aumento en el tiempo de ejecución sería dramático. Para ver esto Considere el siguiente procedimiento recursivo (ineficiente) que determina $m[i, j]$, el mínimo número de multiplicaciones escalares necesarias para calcular el producto matriz-cadena $A_{i..j} = A_i A_{i+1} \cdots A_j$. El procedimiento se basa directamente en la recurrencia (15.12).

```
CADENA-MATRIZ-RECURSIVA ( p , i , j )
1 si i = j
2 luego devuelve 0
3 m [ i , j ] ← ∞
4 para k ← i a j - 1
5 hacer q ← CADENA-MATRIZ-RECURSIVA ( p , i , k )
                    + CADENA-MATRIZ-RECURSIVA ( p , k + 1, j )
                    + p_{i-1} p_k p_j
6           si q < m [ i , j ]
7               entonces m [ i , j ] ← q
8 devuelve m [ i , j ]
```

La figura 15.5 muestra el árbol de recursividad producido por la llamada RECURSIVE-MATRIX-CHAIN (p , 1, 4). Cada nodo está etiquetado por los valores de los parámetros i y j . Observe que algunos pares de los valores ocurren muchas veces.

Figura 15.5: El árbol de recursividad para el cálculo de RECURSIVE-MATRIX-CHAIN (p , 1, 4). Cada nodo contiene los parámetros i y j . Los cálculos realizados en un sombreado Los subárboles se reemplazan por una única búsqueda de tabla en MEMOIZED-MATRIX-CHAIN (p , 1, 4).

De hecho, podemos demostrar que el tiempo para calcular $m[1, n]$ mediante este procedimiento recursivo es al menos exponencial en n . Sea $T(n)$ el tiempo que tarda la CADENA-MATRIZ-RECURSIVA en Calcule un paréntesis óptimo de una cadena de n matrices. Si asumimos que la ejecución de las líneas 1–2 y de las líneas 6–7 cada una toma al menos un tiempo unitario, luego la inspección del procedimiento produce la recurrencia

Página 296

Observando que para $i = 1, 2, \ldots, n - 1$, cada término $T(i)$ aparece una vez como $T(k)$ y una vez como $T(n - k)$, y recolectando los $n - 1$ 1 en la suma junto con el 1 al frente, podemos reescribir el recurrencia como

(15,13)

Demostraremos que $T(n) = \Omega(2^n)$ usando el método de sustitución. Específicamente, mostraremos que $T(n) \geq 2^{n-1}$ para todo $n \geq 1$. La base es fácil, ya que $T(1) \geq 1 = 2^0$. Inductivamente, para $n \geq 2$, tener

que completa la prueba. Por lo tanto, la cantidad total de trabajo realizado por la llamada RECURSIVE-MATRIX-CHAIN (p , 1, n) es al menos exponencial en n .

Compare este algoritmo recursivo de arriba hacia abajo con el de programación dinámica de abajo hacia arriba algoritmo. Este último es más eficiente porque aprovecha la superposición propiedad de subproblemas. Solo hay $\Theta(n^2)$ subproblemas diferentes, y la dinámica El algoritmo de programación resuelve cada uno exactamente una vez. El algoritmo recursivo, por otro lado, debe resolver repetidamente cada subproblema cada vez que reaparece en el árbol de recursividad. Siempre que un árbol de recursividad para la solución recursiva natural de un problema contenga el mismo subproblema repetidamente, y el número total de subproblemas diferentes es pequeño, es una buena idea para ver si se puede hacer que la programación dinámica funcione.

Reconstruyendo una solución óptima

En la práctica, a menudo almacenamos la elección que hicimos en cada subproblema en una tabla para que no tenemos que reconstruir esta información a partir de los costos que almacenamos. En montaje programación de línea, almacenamos en $l_i[j]$ la estación que precede a $S_{i,j}$ de una manera más rápida a través de $S_{i,j}$. Alternativamente, habiendo llenado toda la tabla $f_i[j]$, podríamos determinar qué estación precede $S_{1,j}$ de una manera más rápida a través de $S_{i,j}$ con un poco de trabajo extra. Si $f_1[j] = f_1[j-1] + a_{1,j}$, entonces la estación $S_{1,j-1}$ precede a $S_{1,j}$ de forma más rápida a través de $S_{1,j}$. De lo contrario, debe darse el caso de que $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$, por lo que $S_{2,j-1}$ precede a $S_{1,j}$. Para la programación de la línea de montaje, la reconstrucción del las estaciones predecesoras solo toman $O(1)$ tiempo por estación, incluso sin la tabla $l_i[j]$.

Sin embargo, para la multiplicación de cadenas de matrices, la tabla $s[i, j]$ nos ahorra una cantidad significativa de trabajar al reconstruir una solución óptima. Supongamos que no mantenemos el $s[i, j]$ tabla, habiendo completado sólo la tabla $m[i, j]$ que contiene los costos óptimos del subproblema. Hay $j - i$ opciones para determinar qué subproblemas utilizar en una solución óptima para poner entre paréntesis $A_i A_{i+1} A_j$, y $j - i$ no es una constante. Por lo tanto, se necesitaría $\Theta(j - i) = \omega(1)$ tiempo para reconstruir qué subproblemas elegimos para una solución a un problema dado. Al almacenar en $s[i, j]$ el índice

Página 297

de la matriz en la que dividimos el producto $A_i A_{i+1} A_j$, podemos reconstruir cada elección en $O(1)$ hora.

Memorización

Existe una variación de programación dinámica que a menudo ofrece la eficiencia de lo habitual. enfoque de programación dinámica manteniendo una estrategia de arriba hacia abajo. La idea es *memorizar* el algoritmo recursivo natural, pero ineficiente. Como en la dinámica ordinaria programación, mantenemos una tabla con soluciones de subproblemas, pero la estructura de control para completar la tabla se parece más al algoritmo recursivo.

Un algoritmo recursivo memorizado mantiene una entrada en una tabla para la solución de cada subproblema. Cada entrada de la tabla contiene inicialmente un valor especial para indicar que la entrada tiene aún por completar. Cuando el subproblema se encuentra por primera vez durante la ejecución del algoritmo recursivo, su solución se calcula y luego se almacena en la tabla. Cada subsiguiente vez que se encuentra el subproblema, el valor almacenado en la tabla simplemente se busca y regresó. [4]

Aquí hay una versión memorizada de RECURSIVE-MATRIX-CHAIN:

```
CADENA-MATRIZ-MEMOIZADA ( p )
1 n ← longitud [ p ] - 1
2 para i ← 1 a n
3 hacen para j ← i a n
4                    hacer m [ i , j ] ← ∞
5 devolver LOOKUP-CHAIN ( p , 1, n )
CADENA DE BÚSQUEDA ( p , i , j )
1 si m [ i , j ] <∞
2 luego devuelve m [ i , j ]
3 si i = j
4 luego m [ i , j ] ← 0
5 más para k ← i a j - 1
```

```
6                 hacer q ← CADENA DE BÚSQUEDA ( p , i , k )
                              + CADENA DE BÚSQUEDA ( p , k + 1, j ) + p i-1 p k p j
7                    si q < m [ i , j ]
8                         entonces m [ i , j ] ← q
9 devuelve m [ i , j ]
```

CADENA-MATRIZ-MEMOIZADA, como ORDEN-CADENA-MATRIZ, mantiene una tabla m [1 n , 1 n] de valores calculados de m [i , j], el número mínimo de multiplicaciones escalares necesarias para calcular la matriz A_{ij} . Cada entrada de la tabla contiene inicialmente el valor ∞ para indicar que el la entrada aún no se ha completado. Cuando se ejecuta la llamada LOOKUP-CHAIN (p , i , j), si m [i, j] <∞ en la línea 1, el procedimiento simplemente devuelve el costo previamente calculado m [i , j] (línea 2). De lo contrario, el costo se calcula como en RECURSIVE-MATRIX-CHAIN, almacenado en m [i , j], y regresó. (El valor ∞ es conveniente de usar para una entrada de tabla sin completar ya que es el valor usado para inicializar m [i , j] en la línea 3 de RECURSIVE-MATRIX-CHAIN.) Por lo tanto, LOOKUP-CHAIN (p , i , j) siempre devuelve el valor de m [i , j], pero solo lo calcula si es la primera vez que LOOKUP-CHAIN ha sido llamado con los parámetros i y j .

La figura 15.5 ilustra cómo MEMOIZED-MATRIX-CHAIN ahorra tiempo en comparación con CADENA DE MATRIZ RECURSIVA. Los subárboles sombreados representan valores que se buscan en lugar de de lo calculado.

Página 298

Al igual que el algoritmo de programación dinámica MATRIX-CHAIN-ORDER, el procedimiento MEMOIZED-MATRIX-CHAIN se ejecuta en tiempo O (n^3). Cada una de las entradas de la tabla Θ (n^2) se inicializa una vez en la línea 4 de MEMOIZED-MATRIX-CHAIN. Podemos categorizar las llamadas de LOOKUP-CADENA en dos tipos:

1. llamadas en las que m [i , j] = ∞, de modo que se ejecutan las líneas 3-9, y
2. llamadas en las que m [i , j] <∞, de modo que LOOKUP-CHAIN simplemente regresa en la línea 2.

Hay Θ (n^2) llamadas del primer tipo, una por entrada de tabla. Todas las llamadas del segundo tipo son realizadas como llamadas recursivas por llamadas del primer tipo. Siempre que una determinada llamada de LOOKUP-CHAIN hace llamadas recursivas, hace O (n) de ellas. Por tanto, hay O (n^3) llamadas del segundo escriba todo. Cada llamada del segundo tipo toma O (1) tiempo, y cada llamada del primer tipo toma O (n) tiempo más el tiempo empleado en sus llamadas recursivas. El tiempo total, por tanto, es O (n^3). Por tanto, la memorización convierte un algoritmo de tiempo Ω (2^n) en un algoritmo de tiempo O (n^3).

En resumen, el problema de la multiplicación de la cadena de la matriz se puede resolver ya sea de arriba hacia abajo, algoritmo memorizado o un algoritmo de programación dinámica ascendente en tiempo O (n^3). Ambos Los métodos aprovechan la propiedad de subproblemas superpuestos. Solo hay Θ (n^2) diferentes subproblemas en total, y cualquiera de estos métodos calcula la solución a cada subproblema una vez. Sin memorización, el algoritmo recursivo natural se ejecuta en exponencial tiempo, ya que los subproblemas resueltos se resuelven repetidamente.

En la práctica general, si todos los subproblemas deben resolverse al menos una vez, una dinámica ascendente El algoritmo de programación generalmente supera a un algoritmo memorizado de arriba hacia abajo por una constante factor, porque no hay sobrecarga para la recursividad y menos sobrecarga para mantener la tabla. Además, existen algunos problemas para los que el patrón regular de accesos a la tabla en el El algoritmo de programación dinámica se puede aprovechar para reducir los requisitos de tiempo o espacio incluso más lejos. Alternativamente, si algunos subproblemas en el espacio de subproblemas no necesitan ser resueltos en absoluto, La solución memorizada tiene la ventaja de resolver solo aquellos subproblemas que definitivamente son necesario.

Ejercicios 15.3-1

¿Cuál es una forma más eficiente de determinar el número óptimo de multiplicaciones en una matriz? Problema de multiplicación de cadenas: enumerando todas las formas de poner entre paréntesis el producto y calculando el número de multiplicaciones para cada uno, o ejecutando RECURSIVE-MATRIX-¿CADENA? Justifica tu respuesta.

Ejercicios 15.3-2

Dibuje el árbol de recursividad para el procedimiento MERGE-SORT de la Sección 2.3.1 en una matriz de 16 elementos. Explique por qué la memorización es ineficaz para acelerar una buena división y conquistar algoritmos como MERGE-SORT.

Página 299

Ejercicios 15.3-3

Considere una variante del problema de multiplicación de cadenas de matrices en el que el objetivo es entre paréntesis la secuencia de matrices para maximizar, en lugar de minimizar, el número de multiplicaciones escalares. ¿Este problema presenta una subestructura óptima?

Ejercicios 15.3-4

Describa cómo la programación de la línea de montaje tiene subproblemas superpuestos.

Ejercicios 15.3-5

Como se dijo, en la programación dinámica primero resolvemos los subproblemas y luego elegimos cuál de para utilizarlos en una solución óptima al problema. El profesor Capuleto afirma que no es siempre es necesario resolver todos los subproblemas para encontrar una solución óptima. Ella sugiere que una solución óptima al problema de multiplicación de cadenas de matrices se puede encontrar mediante eligiendo siempre la matriz A_k en la que dividir el subproducto $A_i A_{i+1} A_j$ (seleccionando k para minimizar la cantidad $p_{i-1} p_k p_j$) *antes de* resolver los subproblemas. Encuentra una instancia del problema de multiplicación de la cadena de la matriz para el cual este enfoque codicioso produce un subóptimo solución.

[1] Esto no es un error ortográfico. La palabra realmente es *memorización* , no *memorización* . *Memorización* viene de *memo* , ya que la técnica consiste en registrar un valor para que podamos buscarlo luego.

[2] Usamos el término "no ponderado" para distinguir este problema del de encontrar el más corto caminos con bordes ponderados, que veremos en los capítulos 24 y 25 . Podemos usar la amplitud primera técnica de búsqueda del capítulo 22 para resolver el problema no ponderado.

[3] Puede parecer extraño que la programación dinámica se base en subproblemas tanto independientes y superpuestos. Aunque estos requisitos pueden parecer contradictorios, describen dos nociones diferentes, en lugar de dos puntos en el mismo eje. Dos subproblemas de el mismo problema son independientes si no comparten recursos. Dos subproblemas son superpuestos si realmente son el mismo subproblema que ocurre como un subproblema de diferentes problemas.

[4] Este enfoque presupone que el conjunto de todos los posibles parámetros del subproblema es conocido y que se establezca la relación entre las posiciones de la mesa y los subproblemas. Otro enfoque es para memorizar utilizando hash con los parámetros del subproblema como claves.

Página 300

15.4 Subsecuencia común más larga

En aplicaciones biológicas, a menudo queremos comparar el ADN de dos (o más) diferentes organismos. Una hebra de ADN consta de una serie de moléculas llamadas **bases**, donde el posible las bases son adenina, guanina, citosina y timina. Representando cada una de estas bases por su letras iniciales, una cadena de ADN se puede expresar como una cadena sobre el conjunto finito {A, C, G, T}. (Consulte el Apéndice C para obtener una definición de cadena.) Por ejemplo, el ADN de un organismo puede ser S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, mientras que el ADN de otro organismo puede ser S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA. Un objetivo de comparar dos hebras de ADN es para determinar qué tan "similares" son las dos hebras, como una medida de cómo estrechamente relacionados los dos organismos están. La similitud puede definirse y se define de muchas formas diferentes. Por ejemplo, podemos decir que dos cadenas de ADN son similares si una es una subcadena de la otra. (El capítulo 32 explora algoritmos para resolver este problema). En nuestro ejemplo, ni S_1 ni S_2 son subcadena de la otra. Alternativamente, podríamos decir que dos hebras son similares si el número de los cambios necesarios para convertir uno en otro es pequeño. (El problema 15-3 analiza esta noción). Otra forma de medir la similitud de las hebras S_1 y S_2 es encontrando una tercera hebra S_3 en que las bases en S_3 aparecen en cada uno de S_1 y S_2 ; estas bases deben aparecer en el mismo orden, pero no necesariamente de forma consecutiva. Cuanto más larga sea la hebra S_3 que podamos encontrar, más similar S_1 y S_2 son. En nuestro ejemplo, la hebra más larga S_3 es GTCGTCGGAAGCCGGCCGAA.

Formalizamos esta última noción de similitud como el problema de subsecuencia común más larga. UNA La subsecuencia de una secuencia dada es solo la secuencia dada con cero o más elementos restantes fuera. Formalmente, dada una secuencia $X = \langle x_1, x_2, ..., x_m \rangle$, otra secuencia $Z = \langle z_1, z_2, ..., z_k \rangle$ es una **subsecuencia** de X si existe una secuencia estrictamente creciente $\langle i_1, i_2, ..., i_k \rangle$ de índices de X tal que para todo $j = 1, 2, ..., k$, tenemos $x_{i_j} = z_j$. Por ejemplo, $Z = \langle B, C, D, B \rangle$ es un subsecuencia de $X = \langle A, B, C, B, D, A, B \rangle$ con la secuencia de índice correspondiente 2, 3, 5, 7.

Dadas dos secuencias X e Y, decimos que una secuencia Z es una **subsecuencia común** de X y Y si Z es una subsecuencia de ambos X y Y. Por ejemplo, si $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$, la secuencia $\langle B, C, A \rangle$ es una subsecuencia común de ambos X y Y. los la secuencia $\langle B, C, A \rangle$ no es una subsecuencia común *más larga* (LCS) de X e Y, sin embargo, ya que tiene longitud 3 y la secuencia $\langle B, C, B, A \rangle$, que también es común tanto a X como a Y, tiene longitud 4. La secuencia $\langle B, C, B, A \rangle$ es una LCS de X e Y, al igual que la secuencia $\langle B, D, A, B \rangle$, ya que no hay una subsecuencia común de longitud 5 o mayor.

En el **problema de la subsecuencia común más larga**, se nos dan dos secuencias $X = \langle x_1, x_2, ..., x_m \rangle$ y $Y = \langle y_1, y_2, ..., y_n \rangle$ y desea encontrar una subsecuencia común de longitud máxima de X y Y. Esta sección muestra que el problema LCS se puede resolver de manera eficiente utilizando programación.

Paso 1: caracterizar una subsecuencia común más larga

Un enfoque de fuerza bruta para resolver el problema de LCS es enumerar todas las subsecuencias de X y verifique cada subsecuencia para ver si también es una subsecuencia de Y, manteniendo un registro de la más larga subsecuencia encontrada. Cada subsecuencia de X corresponde a un subconjunto de los índices {1, 2, ..., m } de X. Hay 2^m subsecuencias de X, por lo que este enfoque requiere un tiempo exponencial, por lo que impráctico para secuencias largas.

El problema LCS tiene una propiedad de subestructura óptima, sin embargo, como el siguiente teorema muestra. Como veremos, las clases naturales de subproblemas corresponden a pares de "prefijos" de

Página 301

las dos secuencias de entrada. Para ser precisos, dada una secuencia $X = \langle x_1, x_2, ..., x_m \rangle$, definimos el i- ésimo **prefijo** de X, para $i = 0, 1, ..., m$, como $X_i = \langle x_1, x_2, ..., x_i \rangle$. Por ejemplo, si $X = \langle A, B, C, B, D, A, B \rangle$, luego $X_4 = \langle A, B, C, B \rangle$ y X_0 es la secuencia vacía.

Teorema 15.1: (Subestructura óptima de un LCS)

Sea $X = \langle x_1, x_2, ..., x_m \rangle$ y $Y = \langle y_1, y_2, ..., y_n \rangle$ ser secuencias, y sean $Z = \langle z_1, z_2, ..., z_k \rangle$ ser cualquier LCS de X y Y.

1. Si $x_m = y_n$, entonces $z_k = x_m = y_n$ y Z_{k-1} es una LCS de X_{m-1} e Y_{n-1}.
2. Si $x_m \neq y_n$, entonces $z_k \neq x_m$ implica que Z es un LCS de X_{m-1} y Y.
3. Si $x_m \neq y_n$, entonces $z_k \neq y_n$ implica que Z es una LCS de X e Y_{n-1}.

Prueba (1) Si $z_k \neq x_m$, entonces podríamos añadir $x_m = y_n$ a Z para obtener una subsecuencia común de X e Y de longitud $k + 1$, lo que contradice la suposición de que Z es una subsecuencia común *más larga* de X y Y. Por lo tanto, debemos tener $z_k = x_m = y_n$. Ahora, el prefijo Z_{k-1} es una longitud- (k - 1) común subsecuencia de X_{m-1} e Y_{n-1}. Deseamos demostrar que es una LCS. Supongamos con el propósito de contradicción de que hay una subsecuencia común W de X_{m-1} e Y_{n-1} con una longitud mayor que k - 1. Luego, agregar $x_m = y_n$ a W produce una subsecuencia común de X e Y cuya longitud es mayor que k, lo cual es una contradicción.

(2) Si $z_k \neq x_m$, entonces Z es una subsecuencia común de X_{m-1} y Y. Si hubiera un común subsecuencia W de X_{m-1} e Y con longitud mayor que k, entonces W también sería un común subsecuencia de X_m y Y, lo que contradice la suposición de que Z es un LCS de X y Y.

(3) La demostración es simétrica a (2).

La caracterización del teorema 15.1 muestra que una LCS de dos secuencias contiene dentro de ella un LCS de prefijos de las dos secuencias. Por tanto, el problema LCS tiene una subestructura óptima propiedad. Una solución recursiva también tiene la propiedad de subproblemas superpuestos, como veremos. en un momento.

Paso 2: una solución recursiva

El teorema 15.1 implica que hay uno o dos subproblemas para examinar al encontrar una LCS de $X = x_1, x_2, ..., x_m$ y $Y = y_1, y_2, ..., y_n$. Si $x_m = y_n$, debemos encontrar una LCS de X_{m-1} e Y_{n-1}. Al añadir $x_m = y_n$ a este LCS produce un LCS de X y Y. Si $x_m \neq y_n$, entonces debe resolver dos subproblemas: encontrar una LCS de X_{m-1} e Y y encontrar una LCS de X e Y_{n-1}. Cualquiera de estos dos LCS de es más largo es un LCS de X y Y. Porque estos casos agotan todos posibilidades, sabemos que una de las soluciones óptimas de subproblema debe utilizarse dentro de un LCS de X y Y.

Podemos ver fácilmente la propiedad de subproblemas superpuestos en el problema LCS. Para encontrar un LCS de X y Y, que pueden necesitar para encontrar el LCS de de X y Y_{n-1} y de X_{m-1} y Y. Pero cada uno de estos subproblemas tiene el subproblema de encontrar la LCS de X_{m-1} e Y_{n-1}. Muchos otros los subproblemas comparten subproblemas.

Página 302

Como en el problema de multiplicación de cadenas de matrices, nuestra solución recursiva al problema LCS implica establecer una recurrencia por el valor de una solución óptima. Definamos $c[i, j]$ como ser la longitud de un LCS de las secuencias X_i e Y_j. Si $i = 0$ o $j = 0$, uno de los secuencias tiene una longitud de 0, por lo que el LCS tiene una longitud de 0. La subestructura óptima del LCS problema da la fórmula recursiva

(15,14)

Observe que en esta formulación recursiva, una condición en el problema restringe qué subproblemas que podemos considerar. Cuando $x_i = y_j$, podemos y debemos considerar el subproblema de hallar la LCS de X_{i-1} e Y_{j-1}. De lo contrario, consideramos en su lugar los dos subproblemas de hallar la LCS de X_i e Y_{j-1} y de X_{i-1} e Y_j. En la programación dinámica anterior algoritmos que hemos examinado, para la programación de la línea de montaje y la cadena de matrices multiplicación: no se descartaron subproblemas debido a las condiciones del problema. Encontrar el LCS no es el único algoritmo de programación dinámica que descarta subproblemas basados en condiciones en el problema. Por ejemplo, el problema de la distancia de edición (vea el problema 15-3) tiene este característica.

Paso 3: Calcular la longitud de un LCS

Según la ecuación (15.14), podríamos escribir fácilmente un algoritmo recursivo de tiempo exponencial para calcular la longitud de un LCS de dos secuencias. Dado que solo hay $\Theta(mn)$ distintos subproblemas, sin embargo, podemos utilizar la programación dinámica para calcular las soluciones bottom arriba.

El procedimiento LCS-LENGTH toma dos secuencias $X = x_1, x_2, ..., x_m$ y $Y = y_1, y_2, ..., y_n$

como entradas. Almacena los valores $c[i, j]$ en una tabla $c[0..m, 0..n]$ cuyas entradas se calculan en orden de fila mayor. (Es decir, la primera fila de c se completa de izquierda a derecha, luego la segunda fila, y así sucesivamente.) También mantiene la tabla $b[1..m, 1..n]$ para simplificar la construcción de un óptimo solución. Intuitivamente, $b[i, j]$ apunta a la entrada de la tabla correspondiente al subproblema óptimo solución elegida al calcular $c[i, j]$. El procedimiento devuelve los b y c tablas; $c[m, n]$ contiene la longitud de un LCS de X y Y.

```
LCS-LONGITUD ( X , Y )
  1 m ← longitud [ X ]
  2 n ← longitud [ Y ]
  3 para i ← 1 a m
  4 do c [ i , 0] ← 0
  5 para j ← 0 a n
  6 hacer c [0, j ] ← 0
  7 para i ← 1 a m
  8 hacer para j ← 1 a n
  9                       hacer si x i = y j
10                                luego c [ i , j ] ← c [ i - 1, j - 1] + 1
11                                     b [ i , j ] ← ""
12                                de lo contrario, si c [ i - 1, j ] ≥ c [ i , j - 1]
13                                          luego c [ i , j ] ← c [ i - 1, j ]
14                                               b [ i , j ] ← "↑"
15                                          más c [ i , j ] ← c [ i , j - 1]
dieciséis                                        b [ i , j ] ← ←
17 de retorno c y b
```

Página 303

La figura 15.6 muestra las tablas producidas por LCS-LENGTH en las secuencias $X = A, B, C, B, D, A, B$ y $Y = B, D, C, A, B, A$. El tiempo de ejecución del procedimiento es $O(mn)$, ya que cada entrada de la tabla toma $O(1)$ tiempo para calcular.

Figura 15.6: La c y b tablas calculadas por LCS-LONGITUD en las secuencias de $X = A, B, C, B, D, A, B$ y $Y = B, D, C, A, B, A$. El cuadrado en la fila i y la columna j contiene el valor de $c[i, j]$ y la flecha apropiada para el valor de $b[i, j]$. La entrada 4 en $c[7, 6]$: la parte inferior esquina de la derecha de la tabla-es la longitud de un LCS B, C, B, A de X y Y. Para $i, j > 0$, la entrada $c[i, j]$ depende solo de si $x_i = y_j$ y los valores en las entradas $c[i - 1, j]$, $c[i, j - 1]$, y $c[i - 1, j - 1]$, que se calculan antes de $c[i, j]$. Para reconstruir los elementos de un LCS, siga las flechas $b[i, j]$ de la esquina inferior derecha; el camino está sombreado. Cada "" en el camino corresponde a una entrada (resaltada) para la cual $x_i = y_j$ es miembro de una LCS.

Paso 4: construcción de un LCS

La tabla b devuelta por LCS-LENGTH se puede utilizar para construir rápidamente un LCS de $X = x_1, x_2, ..., x_m$ y $Y = y_1, y_2, ..., y_n$. Simplemente comenzamos en $b[m, n]$ y seguimos la tabla siguiendo las flechas. Siempre que encontramos un "" en la entrada $b[i, j]$, implica que $x_i = y_j$ es un elemento de la LCS. Los elementos de la LCS se encuentran en orden inverso por este método. El siguiente procedimiento recursivo imprime una LCS de X e Y en el correcto, orden de avance. La invocación inicial es PRINT-LCS (b , X , *longitud* [X], *longitud* [Y]).

```
PRINT-LCS ( b , X , i , j )
```

```
1 si i = 0 o j = 0
2 luego regresa
3 si b [ i , j ] = ""
4 luego PRINT-LCS ( b , X , i - 1, j - 1)
5                 imprimir x i
6 elseif b [ i , j ] = "↑"
7 luego PRINT-LCS ( b , X , i - 1, j )
8 más PRINT-LCS ( b , X , i , j - 1)
```

Página 304

Para la tabla b de la figura 15.6 , este procedimiento imprime " *BCBA* ". El procedimiento lleva tiempo $O(m + n)$, ya que al menos uno de i y j se reduce en cada etapa de la recursividad.

Mejorando el código

Una vez que haya desarrollado un algoritmo, a menudo encontrará que puede mejorar el tiempo o espacio que utiliza. Esto es especialmente cierto en el caso de los algoritmos sencillos de programación dinámica. Algunos cambios pueden simplificar el código y mejorar los factores constantes, pero por lo demás no producen mejora asintótica en el rendimiento. Otros pueden producir ahorros asintóticos sustanciales en tiempo y espacio.

Por ejemplo, podemos eliminar la tabla b *por* completo. Cada entrada $c[i, j]$ depende de solo tres otras entradas de la tabla c : $c[i - 1, j - 1]$, $c[i - 1, j]$ y $c[i, j - 1]$. Dado el valor de $c[i, j]$, podemos determinar en $O(1)$ tiempo cuál de estos tres valores se utilizó para calcular $c[i, j]$, sin mesa de inspección b . Por lo tanto, podemos reconstruir una LCS en tiempo $O(m + n)$ usando un procedimiento similar a PRINT-LCS. (El ejercicio 15.4-2 le pide que proporcione el pseudocódigo.) Aunque guardamos $\Theta(mn)$ de espacio por este método, el requisito de espacio auxiliar para calcular un LCS no Disminuir asintóticamente, ya que de todos modos necesitamos $\Theta(mn)$ espacio para la tabla c .

Sin embargo, podemos reducir los requisitos de espacio asintótico para LCS-LENGTH, ya que solo necesita dos filas de la tabla c a la vez: la fila que se calcula y la fila anterior. (En De hecho, podemos usar solo un poco más que el espacio para una fila de c para calcular la longitud de un LCS. Consulte el ejercicio 15.4-4 .) Esta mejora funciona si solo necesitamos la longitud de una LCS; Si necesitamos reconstruir los elementos de un LCS, la tabla más pequeña no guarda suficiente información para volver sobre nuestros pasos en el tiempo $O(m + n)$.

Ejercicios 15.4-1

Determine una LCS de 1, 0, 0, 1, 0, 1, 0, 1 y 0, 1, 0, 1, 1, 0, 1, 1, 0.

Ejercicios 15.4-2

Muestre cómo reconstruir un LCS a partir de la tabla c completada y las secuencias originales $X = x_1, x_2, ..., x_m$ y $Y = y_1, y_2, ..., y_n$ en tiempo $O(m + n)$, sin usar la tabla b .

Ejercicios 15.4-3

Proporcione una versión memorizada de LCS-LENGTH que se ejecute en tiempo $O(mn)$.

Ejercicios 15.4-4

Página 305

Muestre cómo calcular la longitud de un LCS usando solo $2 \cdot \min(m, n)$ entradas en la tabla c más $O(1)$ espacio adicional. Luego muestre cómo hacer esto usando entradas mínimas (m, n) más $O(1)$ adicionales espacio.

Ejercicios 15.4-5

Dar un algoritmo de tiempo $O(n^2)$ para encontrar la subsecuencia creciente monótonamente más larga de un secuencia de n números.

Ejercicios 15.4-6: ⋆

Dar un algoritmo de tiempo $O(n \lg n)$ para encontrar la subsecuencia más larga que aumenta monótonamente de una secuencia de n números. (*Sugerencia:* observe que el último elemento de una subsecuencia candidata de longitud i es al menos tan grande como el último elemento de una subsecuencia candidata de longitud i - 1. Mantenga las subsecuencias candidatas vinculándolas a través de la secuencia de entrada).

15.5 Árboles de búsqueda binarios óptimos

Supongamos que estamos diseñando un programa para traducir texto del inglés al francés. Para cada aparición de cada palabra en inglés en el texto, debemos buscar su equivalente en francés. Uno La forma de realizar estas operaciones de búsqueda es construir un árbol de búsqueda binario con n palabras en inglés. como claves y equivalentes en francés como datos de satélite. Porque buscaremos en el árbol cada palabra individual en el texto, queremos que el tiempo total dedicado a la búsqueda sea lo más bajo posible. Podríamos asegurar un tiempo de búsqueda de $O(\lg n)$ por ocurrencia usando un árbol rojo-negro o cualquier otro árbol de búsqueda binaria equilibrado. Las palabras aparecen con diferentes frecuencias, sin embargo, y puede ser el caso de que una palabra de uso frecuente como "el" aparezca lejos de la raíz mientras que una palabra rara vez una palabra usada como "micofagista" aparece cerca de la raíz. Una organización así ralentizaría la traducción, ya que el número de nodos visitados al buscar una clave en un binario El árbol de búsqueda es uno más la profundidad del nodo que contiene la clave. Queremos palabras que ocurran frecuentemente en el texto para ser colocado más cerca de la raíz. [5] Además, puede haber palabras en el texto. para el que no hay traducción al francés, y es posible que tales palabras no aparezcan en la búsqueda binaria árbol en absoluto. ¿Cómo organizamos un árbol de búsqueda binario para minimizar el número de nodos? visitado en todas las búsquedas, dado que sabemos con qué frecuencia aparece cada palabra?

Lo que necesitamos se conoce como un árbol de búsqueda binario óptimo . Formalmente, se nos da una secuencia $K = \langle k_1, k_2, \ldots, k_n \rangle$ de n claves distintas en orden ordenado (de modo que $k_1 < k_2 < \cdots < k_n$), y deseamos construya un árbol de búsqueda binario a partir de estas claves. Para cada clave k_i , tenemos una probabilidad p_i de que un la búsqueda será para k_i . Algunas búsquedas pueden ser para valores que no están en K , por lo que también tenemos $n + 1$ "teclas ficticias" $d_0, d_1, d_2, \ldots, d_n$ representan valores no en K . En particular, d_0 representa todos valores menores que k_1 , d_n representa todos los valores mayores que k_n , y para $i = 1, 2, \ldots, n$ -1, el La tecla ficticia d_i representa todos los valores entre k_i y k_{i+1} . Para cada tecla dummy d_i , tenemos un probabilidad q_i de que una búsqueda corresponda a d_i . La figura 15.7 muestra dos árboles de búsqueda binarios para un juego de $n = 5$ llaves. Cada clave k_i es un nodo interno y cada clave ficticia d_i es una hoja. Cada

Página 306

la búsqueda es exitosa (encontrando alguna clave k_i) o no exitosa (encontrando alguna clave ficticia d_i), y así tenemos

Figura 15.7: Dos árboles de búsqueda binaria para un conjunto de $n = 5$ claves con lo siguiente Probabilidades:

yo	**0**	**1**	**2**	**3**	**4**	**5**
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10

(a) Un árbol de búsqueda binario con un costo de búsqueda esperado de 2,80. **(b)** Un árbol de búsqueda binaria con costo de búsqueda esperado 2.75. Este árbol es óptimo.

(15,15)

Debido a que tenemos probabilidades de búsquedas para cada clave y cada clave ficticia, podemos determinar el costo esperado de una búsqueda en un árbol de búsqueda binaria dada T . Supongamos que el El costo real de una búsqueda es el número de nodos examinados, es decir, la profundidad del nodo encontrado por la búsqueda en T , más 1. Entonces el costo esperado de una búsqueda en T es

(15,16)

donde la profundidad $_T$ denota la profundidad de un nodo en el árbol T . La última igualdad se sigue de la ecuación (15,15) . En la Figura 15.7 (a) , podemos calcular el costo de búsqueda esperado nodo por nodo:

		contribución de probabilidad de profundidad de nodo	
k_1	1	0,15	0,30
k_2	0	0,10	0,10
k_3	2	0,05	0,15
k_4	1	0,10	0,20
k_5	2	0,20	0,60
d_0	2	0,05	0,15
d_1	2	0,10	0,30

Página 307

		contribución de probabilidad de profundidad de nodo	
d_2	3	0,05	0,20
d_3	3	0,05	0,20
d_4	3	0,05	0,20
d_5	3	0,10	0.40
Total			2,80

Para un conjunto dado de probabilidades, nuestro objetivo es construir un árbol de búsqueda binario cuyo esperado el costo de búsqueda es el más pequeño. Llamamos a ese árbol un *árbol de búsqueda binario óptimo* . Figura 15.7 (b) muestra un árbol de búsqueda binario óptimo para las probabilidades dadas en el título de la figura; sus El costo esperado es 2,75. Este ejemplo muestra que un árbol de búsqueda binario óptimo no es necesariamente un árbol cuya altura total es menor. Tampoco podemos construir necesariamente un binario óptimo árbol de búsqueda poniendo siempre la clave con la mayor probabilidad en la raíz. Aquí, la clave k_5 tiene la mayor probabilidad de búsqueda de cualquier clave, sin embargo, se muestra la raíz del árbol de búsqueda binario óptimo es k_2 . (El costo esperado más bajo de cualquier árbol de búsqueda binaria con k_5 en la raíz es 2.85).

Al igual que con la multiplicación de cadenas de matrices, la verificación exhaustiva de todas las posibilidades no produce una algoritmo eficiente. Podemos etiquetar los nodos de cualquier árbol binario de n nodos con las claves k_1 , k_2 , ..., k_n para construir un árbol de búsqueda binario, y luego agregue las claves ficticias como hojas. En el problema 12-4 , vimos que el número de árboles binarios con n nodos es $\Omega(4^n / n^{3/2})$, por lo que hay un número exponencial de árboles de búsqueda binaria que tendríamos que examinar de forma exhaustiva buscar. Como era de esperar, resolveremos este problema con la programación dinámica.

Paso 1: la estructura de un árbol de búsqueda binario óptimo

Para caracterizar la subestructura óptima de árboles de búsqueda binarios óptimos, comenzamos con un observación sobre subárboles. Considere cualquier subárbol de un árbol de búsqueda binario. Debe contener claves en un rango contiguo k_i, ..., k_j, para algunos $1 \le i \le j \le n$. Además, un subárbol que contiene claves k_i, ..., k_j también debe tener como hojas las claves ficticias d_{i-1}, ..., d_j.

Ahora podemos establecer la subestructura óptima: si un árbol de búsqueda binario óptimo T tiene un subárbol T' que contiene claves k_i, ..., k_j, entonces este subárbol T' debe ser óptimo también para el subproblema con teclas k_i, ..., k_j y teclas ficticias d_{i-1}, ..., d_j. Se aplica el argumento habitual de cortar y pegar. Sí hay fuera un subárbol T"cuyo costo esperado es menor que el de T', entonces podríamos eliminar T' de T y pegar en T", lo que da como resultado un árbol de búsqueda binaria de menor costo esperado que T, por lo tanto contradiciendo la optimalidad de T.

Necesitamos utilizar la subestructura óptima para demostrar que podemos construir una solución óptima para el problema de las soluciones óptimas a los subproblemas. Dadas las claves k_i, ..., k_j, una de estas claves, digamos que k_r ($i \le r \le j$), será la raíz de un subárbol óptimo que contenga estas claves. El subárbol izquierdo de la raíz k_r contendrá las claves k_i, ..., k_{r-1} (y las claves ficticias d_{i-1}, ..., d_{r-1}), y la derecha El subárbol contendrá las claves k_{r+1}, ..., k_j (y las claves ficticias d_r, ..., d_j). Mientras examinemos todos raíces candidatas k_r, donde $i \le r \le j$, y determinamos todos los árboles de búsqueda binarios óptimos que contienen k_i, ..., k_{r-1} y los que contienen k_{r+1}, ..., k_j, tenemos la garantía de que encontraremos un óptimo árbol de búsqueda binaria.

Página 308

Hay un detalle que vale la pena señalar sobre los subárboles "vacíos". Supongamos que en un subárbol con claves k_i, ..., k_j, seleccionamos k_i como raíz. Por el argumento anterior, k_i's subárbol izquierdo contiene las claves k_i, ..., k_{i-1}. Es natural interpretar que esta secuencia no contiene claves. Sin embargo, tenga en cuenta que los subárboles también contienen claves ficticias. Adoptamos la convención de que un subárbol que contiene keys k_i, ..., k_{i-1} no tiene claves reales pero contiene la clave ficticia única d_{i-1}. Simétricamente, si seleccionamos k_j como la raíz, entonces k_j's subárbol derecho contiene las claves k_{j+1}, ..., k_j; este subárbol derecho no contiene claves reales, pero contiene la clave ficticia d_j.

Paso 2: una solución recursiva

Estamos listos para definir el valor de una solución óptima de forma recursiva. Elegimos nuestro subproblema dominio como encontrar un árbol de búsqueda binario óptimo que contenga las claves k_i, ..., k_j, donde $i \ge 1, j \le n$, y $j \ge i - 1$. (Es cuando $j = i - 1$ no hay claves reales; solo tenemos la clave ficticia d_{i-1}.) Definamos $e[i, j]$ como el costo esperado de buscar un árbol de búsqueda binario óptimo que contiene las claves k_i, ..., k_j. En última instancia, deseamos calcular $e[1, n]$.

El caso fácil ocurre cuando $j = i - 1$. Entonces solo tenemos la clave ficticia d_{i-1}. Lo esperado el costo de búsqueda es $e[i, i - 1] = q_{i-1}$.

Cuando $j \ge i$, necesitamos seleccionar una raíz k_r de entre k_i, ..., k_j y luego hacer un binario óptimo árbol de búsqueda con claves k_i, ..., k_{r-1} su subárbol izquierdo y un árbol de búsqueda binario óptimo con claves k_{r+1}, ..., k_j su subárbol derecho. ¿Qué sucede con el costo de búsqueda esperado de un subárbol cuando se convierte en un subárbol de un nodo? La profundidad de cada nodo en el subárbol aumenta en 1. Por ecuación (15.16), el costo de búsqueda esperado de este subárbol aumenta por la suma de todas las probabilidades en el subárbol. Para un subárbol con claves k_i, ..., k_j, denotemos esta suma de probabilidades como

(15,17)

Por lo tanto, si k_r es la raíz de un subárbol óptimo que contiene claves k_i, ..., k_j, tenemos

$$e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j)).$$

Señalando que

$$w(yo, j) = w(yo, r - 1) + p_r + w(r + 1, j),$$

reescribimos $e[i, j]$ como

(15,18)

La ecuación recursiva (15.18) supone que sabemos qué nodo k_r usar como raíz. Nosotros elegir la raíz que ofrezca el menor costo de búsqueda esperado, lo que nos da nuestro recursivo final formulación:

(15,19)

Página 309

Los valores $e[i, j]$ dan los costos de búsqueda esperados en árboles de búsqueda binarios óptimos. Para ayudarnos realizar un seguimiento de la estructura de los árboles de búsqueda binaria óptimos, definimos la *raíz* $[i, j]$, para $1 \le i \le j \le n$, para ser el índice r para el cual k_r es la raíz de un árbol de búsqueda binario óptimo que contiene las claves $k_i, \ldots, k_j$. Aunque veremos cómo calcular los valores de *raíz* $[i, j]$, dejamos la construcción de el árbol de búsqueda binario óptimo a partir de estos valores como el ejercicio 15.5-1 .

Paso 3: Calcular el costo de búsqueda esperado de un árbol de búsqueda binario óptimo

En este punto, es posible que haya notado algunas similitudes entre nuestras caracterizaciones de árboles de búsqueda binaria óptimos y multiplicación de cadenas de matrices. Para ambos dominios problemáticos, nuestro los subproblemas consisten en subintervalos de índice contiguos. Una implementación directa y recursiva de La ecuación (15.19) sería tan ineficiente como una multiplicación directa y recursiva de cadenas de matrices. algoritmo. En su lugar, almacenamos los valores $e[i, j]$ en una tabla $e[1\ldots n+1, 0\ldots n]$. El primer índice necesita ejecutarse en $n + 1$ en lugar de n porque para tener un subárbol que contenga solo el clave ficticia d_n, necesitaremos calcular y almacenar $e[n + 1, n]$. El segundo índice debe comenzar desde 0 porque para tener un subárbol que contenga solo la clave ficticia d_0, necesitaremos calcular y almacenar $e[1, 0]$. Usaremos solo las entradas $e[i, j]$ para las cuales $j \ge i - 1$. También usamos un *raíz de* tabla $[i, j]$, para registrar la raíz del subárbol que contiene las claves $k_i, \ldots, k_j$. Esta tabla utiliza sólo las entradas para las que $1 \le i \le j \le n$.

Necesitaremos otra tabla para mayor eficiencia. En lugar de calcular el valor de $w(i, j)$ a partir de scratch cada vez que calculamos $e[i, j]$, lo que requeriría $\Theta(j - i)$ adiciones, almacenamos estos valores en una tabla $w[1\ldots n+1, 0\ldots n]$. Para el caso base, calculamos $w[i, i - 1] = q_{i-1}$ para $1 \le yo \le n$. Para $j \ge i$, calculamos

(15,20)

Por lo tanto, podemos calcular los valores $\Theta(n^2)$ de $w[i, j]$ en $\Theta(1)$ tiempo cada uno.

El pseudocódigo que sigue toma como entradas las probabilidades $p_1, \ldots, p_n$ y $q_0, \ldots, q_n$ y la tamaño n, y devuelve las tablas e y *de la raíz* .

```
OPTIMAL-BST (p, q, n)
 1 para i ← 1 a n + 1
 2 ¿ e[i, i - 1] ← q_{i-1}
 3                 w[yo, yo - 1] ← q_{yo-1}
 4 para l ← 1 a n
 5 hacer para i ← 1 a n - l + 1
 6                 hacer j ← i + l - 1
 7                     e[i, j] ← ∞
 8                     w[i, j] ← w[i, j - 1] + p_j + q_j
 9                     para r ← i a j
10                         hacer t ← e[i, r - 1] + e[r + 1, j] + w[i, j]
11                             si t < e[i, j]
12                                 entonces e[i, j] ← t
13                                         raíz[i, j] ← r
14 return e y root
```

De la descripción anterior y la similitud con el procedimiento MATRIX-CHAIN-ORDER en En la sección 15.2 , el funcionamiento de este procedimiento debería ser bastante sencillo. El bucle **for** de las líneas 1–3 inicializan los valores de $e[i, i - 1]$ y $w[i, i - 1]$. El bucle **for** de las líneas 4-13 luego usa las recurrencias (15.19) y (15.20) para calcular $e[i, j]$ $yw[i, j]$ para todo $1 \le i \le j \le n$. En el

Página 310

primera iteración, cuando $l = 1$, el ciclo calcula $e[i, i]$ $yw[i, i]$ para $i = 1, 2, ..., n$. El segundo iteración, con $l = 2$, calcula $e[i, i + 1]$ $yw[i, i+1]$ para $i = 1, 2, ..., n - 1$, y así sucesivamente. los más interna **de** bucle, en líneas 9-13, trata cada índice candidato r para determinar qué tecla k_r a utilizar como raíz de un árbol de búsqueda binario óptimo que contenga las claves $k_i, ..., k_j$. Esto **para** bucle guarda el valor actual del índice r en $root[i, j]$ siempre que encuentre una clave mejor para usar como raíz.

La figura 15.8 muestra las tablas $e[i, j]$, $w[i, j]$ y *raíz* $[i, j]$ calculadas por el procedimiento OPTIMAL-BST en la distribución de claves que se muestra en la Figura 15.7 . Como en la cadena de matriz Ejemplo de multiplicación, las tablas se rotan para hacer que las diagonales corran horizontalmente. OPTIMAL-BST calcula las filas de abajo hacia arriba y de izquierda a derecha dentro de cada fila.

Figura 15.8: Las tablas $e[i, j]$, w $[i, j]$ y *raíz* $[i, j]$ calculadas por OPTIMAL-BST en la clave distribución que se muestra en la Figura 15.7. Las mesas se rotan para que las diagonales corran horizontalmente.

El procedimiento OPTIMAL-BST toma $\Theta(n^3)$ tiempo, al igual que MATRIX-CHAIN-ORDER. Es Es fácil ver que el tiempo de ejecución es $O(n^3)$, ya que sus bucles **for** están anidados en tres profundos y cada uno el índice de bucle toma como máximo n valores. Los índices de bucle en OPTIMAL-BST no tienen exactamente los mismos límites que los de MATRIX-CHAIN-ORDER, pero están dentro de como máximo 1 en todos direcciones. Por lo tanto, como MATRIX-CHAIN-ORDER, el procedimiento OPTIMAL-BST toma $\Omega(n^3)$ hora.

Ejercicios 15.5-1

Escriba un pseudocódigo para el procedimiento CONSTRUCT-OPTIMAL-BST (*root*) que, dada la *raíz de la* tabla , genera la estructura de un árbol de búsqueda binario óptimo. Para el ejemplo de la Figura 15.8 , su procedimiento debe imprimir la estructura

- k_2 es la raíz
- k_1 es el hijo izquierdo de k_2
- d_0 es el hijo izquierdo de k_1
- d_1 es el hijo derecho de k_1
- k_5 es el hijo derecho de k_2
- k_4 es el hijo izquierdo de k_5
- k_3 es el hijo izquierdo de k_4
- d_2 es el hijo izquierdo de k_3
- d_3 es el hijo correcto de k_3

Página 311

- d_4 es el hijo correcto de k_4
- d_5 es el hijo correcto de k_5

correspondiente al árbol de búsqueda binario óptimo que se muestra en la Figura 15.7 (b) .

Ejercicios 15.5-2

Determine el costo y la estructura de un árbol de búsqueda binario óptimo para un conjunto de $n = 7$ claves con las siguientes probabilidades:

yo	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
p_i		0,04	0,06	0,08	0,02	0,10	0,12	0,14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

Ejercicios 15.5-3

Suponga que en lugar de mantener la tabla $w[i, j]$, calculamos el valor de $w(i, j)$ directamente de la ecuación (15.17) en la línea 8 de OPTIMAL-BST y usó este valor calculado en la línea 10. ¿Cómo afectaría este cambio al tiempo de funcionamiento asintótico de OPTIMAL-BST?

Ejercicios 15.5-4: ⋆

Knuth [184] ha demostrado que siempre hay raíces de subárboles óptimos tales que $root[i, j - 1] \le raíz[i, j] \le raíz[i + 1, j]$ para todo $1 \le i < j \le n$. Utilice este hecho para modificar el OPTIMAL-BST procedimiento para ejecutar en $\Theta(n^2)$ tiempo.

Problemas 15-1: Problema bitónico euclidiano del viajante de comercio

El **problema del viajante de comercio euclidiano** es el problema de determinar la recorrido que conecta un conjunto dado de n puntos en el plano. La figura 15.9 (a) muestra la solución a un 7-problema de punto. El problema general es NP-completo y, por lo tanto, se cree que su solución requieren más que el tiempo polinomial (ver Capítulo 34).

Página 312

Figura 15.9: Siete puntos en el plano, mostrados en una cuadrícula unitaria. *(a)* El recorrido cerrado más corto, con longitud aproximadamente 24,89. Este recorrido no es bitónico. *(b)* El recorrido bitónico más corto para el mismo cambio de agujas. Su longitud es de aproximadamente 25,58.

JL Bentley ha sugerido que simplifiquemos el problema restringiendo nuestra atención a **bitonic recorridos**, es decir, recorridos que comienzan en el punto más a la izquierda, van estrictamente de izquierda a derecha hacia el extremo derecho punto, y luego vaya estrictamente de derecha a izquierda de regreso al punto de partida. La figura 15.9 (b) muestra la recorrido bitónico más corto de los mismos 7 puntos. En este caso, un algoritmo de tiempo polinomial es posible.

Describa un algoritmo de tiempo $O(n^2)$ para determinar un recorrido bitónico óptimo. Puedes asumir que no hay dos puntos que tengan la misma coordenada x. (*Sugerencia:* escanee de izquierda a derecha, manteniendo posibilidades para las dos partes del recorrido.)

Problemas 15-2: Impresión ordenada

Considere el problema de imprimir ordenadamente un párrafo en una impresora. El texto de entrada es una secuencia de n palabras de longitudes $l_1, l_2, ..., l_n$, medidas en caracteres. Queremos imprimir este párrafo prolijamente en una serie de líneas que contienen un máximo de M caracteres cada una. Nuestro criterio de "pulcritud" es la siguiente. Si una línea dada contiene palabras i a j, donde $i \le j$, y dejamos exactamente un espacio entre palabras, el número de caracteres de espacio extra al final de la línea es , que debe ser no negativo para que las palabras quepan en la línea. Deseamos minimizar la suma, en todas las líneas excepto en la última, de los cubos de los números de espacio extra caracteres al final de las líneas. Dar un algoritmo de programación dinámica para imprimir un párrafo de n palabras cuidadosamente en una impresora. Analice el tiempo de ejecución y los requisitos de espacio de su algoritmo.

Problemas 15-3: Editar distancia

Para transformar una cadena de origen de texto $x[1\ m]$ en una cadena de destino $y[1\ n]$, podemos realizar diversas operaciones de transformación. Nuestro objetivo es, dado X y Y, para producir una serie de transformaciones que cambian de x a y. Usamos una matriz z: *se* supone que es lo suficientemente grande para contener todos los caracteres que necesitará para contener los resultados intermedios. Inicialmente, z está vacío y en terminación, deberíamos tener $z[j] = y[j]$ para $j = 1, 2, ..., n$. Mantenemos los índices actuales i en x y j en z, y las operaciones se les permite alter z y estos índices. Inicialmente, $i = j = 1$. Nosotros deben examinar cada carácter en x durante la transformación, lo que significa que en al final de la secuencia de operaciones de transformación, debemos tener $i = m + 1$.

Página 313

Hay seis operaciones de transformación:

- **Copiar** un personaje de X a Z estableciendo $z[j] \leftarrow x[i]$ y luego incrementando tanto i y j. Esta operación examina $x[i]$.
- **Reemplace** un carácter de x por otro carácter c, estableciendo $z[j] \leftarrow c$, y luego incrementando tanto i como j. Esta operación examina $x[i]$.
- **Elimine** un carácter de x incrementando i pero dejando j solo. Esta operacion examina $x[i]$.
- **Inserte** el carácter c en z estableciendo $z[j] \leftarrow cy$ luego incrementando j, pero dejando i solo. Esta operación no examina ningún carácter de x.
- **Twiddle** (es decir, de cambio) los siguientes dos caracteres copiándolos desde x a z pero en el orden opuesto; lo hacemos configurando $z[j] \leftarrow x[i+1]$ y $z[j+1] \leftarrow x[i]$ y luego configurando $i \leftarrow i + 2$ y $j \leftarrow j + 2$. Esta operación examina $x[i]$ y $x[i+1]$.
- **Elimina** el resto de x configurando $i \leftarrow m + 1$. Esta operación examina todos los caracteres en x que aún no han sido examinados. Si se realiza esta operación, debe ser la última operación.

Como ejemplo, una forma de transformar el algoritmo de la cadena de origen en la cadena de destino altruista es utilizar la siguiente secuencia de operaciones, donde los caracteres subrayados son $x[i]$ y $z[j]$ después de la operación:

Operación	x	z
	algoritmo de *cadenas iniciales* _	
Copiar	algoritmo a_	
Copiar	algoritmo al_	
reemplazar por t	algoritmo alt_	
Eliminar	algoritmo alt_	
Copiar	algoritmo altr_	
inserta u	algoritmo altru_	
insertar yo	algoritmo altrui_	
insertar s	algoritmo altruis_	
girar	algoritmo altruisti_	

insertar c	algoritmo altruista_
matar	algoritmo_altruista_

Tenga en cuenta que hay varias otras secuencias de operaciones de transformación que transforman algoritmo a altruista.

Cada una de las operaciones de transformación tiene un costo asociado. El costo de una operación depende de la aplicación específica, pero asumimos que el costo de cada operación es una constante que es conocido por nosotros. También asumimos que los costos individuales de las operaciones de copia y reemplazo son menores que los costos combinados de las operaciones de eliminación e inserción; de lo contrario, la copia y No se utilizarían operaciones de reemplazo. El costo de una secuencia de transformación dada

Página 314

operaciones es la suma de los costos de las operaciones individuales en la secuencia. Para el secuencia anterior, el costo de transformar el algoritmo en altruista es

(3 · costo (copiar)) + costo (reemplazar) + costo (eliminar) + (4 · costo (insertar)) + costo (twiddle) + costo (matar).

a. Dadas dos secuencias x [1 m] e y [1 n] y un conjunto de costos de operación de transformación, la *distancia de edición* de x a y es el costo de la secuencia de operación costosa menos que transforma x en y . Describe un algoritmo de programación dinámica que encuentra la edición distancia desde x [1 m] ay [1 n] e imprime una secuencia de operación óptima. Analizar el tiempo de ejecución y los requisitos de espacio de su algoritmo.

El problema de la distancia de edición es una generalización del problema de alinear dos secuencias de ADN (ver, por ejemplo, Setubal y Meidanis [272 , Sección 3.2]). Hay varios métodos para midiendo la similitud de dos secuencias de ADN alineándolas. Uno de esos métodos para alinear dos secuencias x y y consiste en insertar espacios en ubicaciones arbitrarias en las dos secuencias (incluyendo en cualquier extremo) de modo que las secuencias resultantes x' e y' tengan la misma longitud pero no no tener un espacio en la misma posición (es decir, para ninguna posición j son tanto $x'[j]$ como $y'[j]$ un espacio). Luego asignamos una "puntuación" a cada puesto. La posición j recibe una puntuación de la siguiente manera:

- +1 si $x'[j] = y'[j]$ y ninguno es un espacio,
- -1 si $x'[j] \neq y'[j]$ y ninguno es un espacio,
- -2 si $x'[j]$ o $y'[j]$ es un espacio.

La puntuación de la alineación es la suma de las puntuaciones de las posiciones individuales. Por ejemplo, dadas las secuencias x = GATCGGCAT ey = CAATGTGAATC, una alineación es

```
G ATCG GCAT
CAAT GTGAATC
- * ++ * + * + - ++ *
```

Un + debajo de una posición indica una puntuación de +1 para esa posición, un - indica una puntuación de -1 y una ⋆ indica una puntuación de -2, por lo que esta alineación tiene una puntuación total de 6 · 1 - 2 · 1 - 4 · 2 _ -4.

si. Explique cómo plantear el problema de encontrar una alineación óptima como distancia de edición. problema al utilizar un subconjunto de las operaciones de transformación copiar, reemplazar, eliminar, insertar, jugar y matar.

Problemas 15-4: planificación de una fiesta de empresa

El profesor Stewart es consultor del presidente de una corporación que está planeando una empresa. partido. La empresa tiene una estructura jerárquica; es decir, la relación de supervisor forma un árbol arraigado en el presidente. La oficina de personal ha calificado a cada empleado con una cordialidad rating, que es un número real. Para que la fiesta sea divertida para todos los asistentes, el presidente no quiere que asistan tanto un empleado como su supervisor inmediato.

El profesor Stewart recibe el árbol que describe la estructura de la corporación, utilizando el representación de hijo izquierdo, hermano derecho descrita en la Sección 10.4 . Cada nodo del árbol contiene, además de los indicadores, el nombre de un empleado y la clasificación de cordialidad de ese empleado.

Página 315

Describe un algoritmo para crear una lista de invitados que maximice la suma de la convivencia.
calificaciones de los invitados. Analiza el tiempo de ejecución de tu algoritmo.

Problemas 15-5: algoritmo de Viterbi

Podemos utilizar la programación dinámica en un gráfico dirigido $G = (V , E)$ para el reconocimiento de voz. Cada borde $(u , v) \in E$ está etiquetado con un sonido $\sigma (u , v)$ de un conjunto finito Σ de sonidos. El etiquetado El gráfico es un modelo formal de una persona que habla un idioma restringido. Cada camino en el gráfico partiendo de un vértice distinguido $v_0 \in V$ corresponde a una posible secuencia de sonidos producido por el modelo. La etiqueta de una ruta dirigida se define como la concatenación de etiquetas de los bordes en ese camino.

a. Describe un algoritmo eficiente que, dado un gráfico G con etiqueta de borde con distintivo vértice v_0 y una secuencia $s = \sigma_1 , \sigma_2 , ..., \sigma_k$ de caracteres de Σ, devuelve una ruta en G que comienza en v_0 y tiene s como su etiqueta, si existe tal ruta. De lo contrario, el algoritmo debe devolver NO-SUCH-PATH. Analiza el tiempo de ejecución de tu algoritmo. (*Pista:* Puede que le resulten útiles los conceptos del Capítulo 22).

Ahora, suponga que a cada arista $(u , v) \in E$ también se le ha dado un asociado no negativo probabilidad $p (u , v)$ de atravesar la arista (u , v) desde el vértice u y así producir el sonido correspondiente. La suma de las probabilidades de que las aristas salgan de cualquier vértice es igual a 1. La probabilidad de un camino se define como el producto de las probabilidades de sus bordes. Podemos ver la probabilidad de que una ruta comience en v_0 como la probabilidad de que una "caminata aleatoria" comenzando en v_0 seguirá la ruta especificada, donde la elección de qué borde tomar en un El vértice u se hace probabilísticamente de acuerdo con las probabilidades de que los bordes disponibles salgan u .

si. Amplíe su respuesta al inciso a) para que, si se devuelve una ruta, sea la *ruta más probable* comenzando en v_0 y con la etiqueta s . Analiza el tiempo de ejecución de tu algoritmo.

Problemas 15-6: moverse en un tablero de ajedrez

Suponga que le dan un tablero de ajedrez de $n \times n$ y un tablero de ajedrez. Debes mover la ficha desde el borde inferior del tablero hasta el borde superior del tablero de acuerdo con lo siguiente regla. En cada paso, puede mover la ficha a uno de los tres cuadrados:

1. el cuadrado inmediatamente superior,
2. el cuadrado que está uno arriba y otro a la izquierda (pero solo si el corrector no está ya en la columna más a la izquierda),
3. el cuadrado que está uno arriba y otro a la derecha (pero solo si el corrector no está ya en la columna de la derecha).

Página 316

Cada vez que pasa del cuadrado x al cuadrado y , recibe $p (x , y)$ dólares. Se le da $p (x , y)$ para todos los pares (x , Y) para los que un movimiento de x a y es legal. No asuma que $p (x , y)$ es positivo.

Proporcione un algoritmo que descubra el conjunto de movimientos que moverán al corrector de en algún lugar del borde inferior hasta en algún lugar del borde superior mientras se reúnen tantos dólares como sea posible. Su algoritmo es libre de elegir cualquier cuadrado a lo largo del borde inferior como un

punto de partida y cualquier cuadrado a lo largo del borde superior como destino para maximizar la cantidad de dólares reunidos en el camino. ¿Cuál es el tiempo de ejecución de su algoritmo?

Problemas 15-7: Programación para maximizar las ganancias

Suponga que tiene una máquina y un conjunto de n trabajos $a_1, a_2, ..., a_n$ para procesar en esa máquina. Cada trabajo a_j tiene un tiempo de procesamiento t_j, una ganancia p_j y una fecha límite d_j. La máquina puede procesar sólo un trabajo a la vez, y el trabajo a_j debe ejecutarse ininterrumpidamente durante t_j unidades de tiempo consecutivas. Si trabajo a_j se completa en su fecha límite d_j, usted recibe una ganancia p_j, pero si se completa después de su fecha límite, recibe una ganancia de 0. Proporcione un algoritmo para encontrar el horario que obtiene el cantidad máxima de beneficio, asumiendo que todos los tiempos de procesamiento son números enteros entre 1 y n. ¿Cuál es el tiempo de ejecución de su algoritmo?

[5] Si el tema del texto son los hongos comestibles, podríamos querer que "micofagista" aparezca cerca la raíz.

Notas del capítulo

R. Bellman comenzó el estudio sistemático de la programación dinámica en 1955. La palabra "programación", tanto aquí como en la programación lineal, se refiere al uso de una solución tabular método. Aunque las técnicas de optimización que incorporan elementos de programación dinámica se conocían antes, Bellman proporcionó al área una sólida base matemática [34].

Hu y Shing [159 , 160] dan un algoritmo de tiempo $O(n \lg n)$ para la multiplicación de cadenas de matrices problema.

El algoritmo de tiempo $O(mn)$ para el problema de subsecuencia común más larga parece ser un problema popular. algoritmo. Knuth [63] planteó la cuestión de si los algoritmos subcuadráticos para el LCS existe el problema. Masek y Paterson [212] respondieron afirmativamente a esta pregunta dando un algoritmo que se ejecuta en tiempo $O(mn / \lg n)$, donde $n \leq m$ y las secuencias se extraen de un conjunto de tamaño acotado. Para el caso especial en el que ningún elemento aparece más de una vez en una secuencia de entrada, Szymanski [288] muestra que el problema se puede resolver en $O((n + m) \lg(n + m))$ tiempo. Muchos de estos resultados se extienden al problema de calcular distancias de edición de cadenas (Problema 15-3).

Uno de los primeros artículos sobre codificaciones binarias de longitud variable de Gilbert y Moore [114] había aplicaciones para construir árboles de búsqueda binarios óptimos para el caso en el que todas las probabilidades

Página 317

p_i son 0; este artículo contiene un algoritmo de tiempo $O(n^3)$. Aho, Hopcroft y Ullman [5] presentan el algoritmo de la Sección 15.5 . El ejercicio 15.5-4 se debe a Knuth [184] . Hu y Tucker [161] ideó un algoritmo para el caso en el que todas las probabilidades p_i son 0 que usa $O(n^2)$ tiempo y $O(n)$ espacio; posteriormente, Knuth [185] redujo el tiempo a $O(n \lg n)$.

Capítulo 16: Algoritmos codiciosos

Visión general

Los algoritmos para problemas de optimización normalmente pasan por una secuencia de pasos, con un conjunto de opciones en cada paso. Para muchos problemas de optimización, el uso de programación dinámica para determinar las mejores opciones es excesivo; Algoritmos más simples y eficientes serán suficientes. Un codicioso El algoritmo siempre toma la decisión que se ve mejor en ese momento. Es decir, hace un local elección óptima con la esperanza de que esta elección conduzca a una solución óptima a nivel mundial. Esta El capítulo explora problemas de optimización que se pueden resolver mediante algoritmos codiciosos. antes de Al leer este capítulo, debería leer sobre programación dinámica en el Capítulo 15 , particularmente Sección 15.3 .

Los algoritmos codiciosos no siempre producen soluciones óptimas, pero para muchos problemas lo hacen. Nosotros Primero examinaré en la Sección 16.1 un problema simple pero no trivial, la selección de actividades

problema, para el cual un algoritmo codicioso calcula eficientemente una solución. Llegaremos al algoritmo codicioso considerando primero una solución de programación dinámica y luego mostrando que siempre podemos tomar decisiones ambiciosas para llegar a una solución óptima. La sección 16.2 revisa el elementos básicos del enfoque codicioso, dando un enfoque más directo para demostrar codicia algoritmos correctos que el proceso basado en programación dinámica de la Sección 16.1 . Sección 16.3 presenta una aplicación importante de técnicas codiciosas: el diseño de compresión de datos (Huffman) códigos. En la sección 16.4 , investigamos algunas de las teorías que subyacen a la combinatoria estructuras llamadas "matroides" para las que un algoritmo codicioso siempre produce un óptimo solución. Finalmente, la Sección 16.5 ilustra la aplicación de matroides usando un problema de programar tareas de tiempo unitario con plazos y sanciones.

El método codicioso es bastante poderoso y funciona bien para una amplia gama de problemas. Luego Los capítulos presentarán muchos algoritmos que pueden verse como aplicaciones de los codiciosos método, incluidos los algoritmos de árbol de expansión mínimo (Capítulo 23), el algoritmo de Dijkstra para caminos más cortos de una sola fuente (Capítulo 24), y la codiciosa heurística de cobertura de conjuntos de Chv´atal (Capítulo 35). Los algoritmos de árbol de expansión mínima son un ejemplo clásico del método codicioso. Aunque este capítulo y el Capítulo 23 se pueden leer de forma independiente, es posible que encuentre es útil leerlos juntos.

16.1 Un problema de selección de actividades

Nuestro primer ejemplo es el problema de programar varias actividades en competencia que requieren uso exclusivo de un recurso común, con el objetivo de seleccionar un conjunto de tamaño máximo de actividades compatibles. Supongamos que tenemos un conjunto $S = \{ a_1 , a_2 , ..., a_n \}$ de n actividades propuestas que desea utilizar un recurso, como una sala de conferencias, que solo puede utilizar una actividad a la vez. Cada actividad a_i tiene una hora de inicio s_i y una hora de finalización f_i , donde $0 \le s_i < f_i < \infty$. Si se selecciona, la actividad a_i tiene lugar durante el intervalo de tiempo semiabierto $[s_i , f_i)$. Las actividades a_i y a_j son compatible si los intervalos $[s_i , f_i)$ y $[s_j f_j)$ no se superponen (es decir, a_i y a_j son compatibles si $s_i \ge$

Página 318

f_j o $s_j \ge f_i$). El problema de la selección de actividades es seleccionar un subconjunto de tamaño máximo de actividades compatibles. Por ejemplo, considere el siguiente conjunto S de actividades, que tenemos ordenados en orden creciente monótona de tiempo de finalización:

yo **1 2 3 4 5 6 7 8 9 10 11**

s_i 1 3 0 5 3 5 6 8 8 2 12
f_i 4 5 6 7 8 9 10 11 12 13 14

(Veremos en breve por qué es ventajoso considerar las actividades en orden ordenado).
Por ejemplo, el subconjunto $\{ a_3 , a_9 , a_{11} \}$ consta de actividades mutuamente compatibles. No es un maximo sin embargo, ya que el subconjunto $\{ a_1 , a_4 , a_8 , a_{11} \}$ es mayor. De hecho, $\{ a_1 , a_4 , a_8 , a_{11} \}$ es el mayor subconjunto de actividades mutuamente compatibles; otro subconjunto más grande es $\{ a_2 , a_4 , a_9 , a_{11} \}$.

Resolveremos este problema en varios pasos. Comenzamos por formular una programación dinámica solución a este problema en el que combinamos soluciones óptimas a dos subproblemas para formar una solución óptima al problema original. Consideramos varias opciones al determinar qué subproblemas utilizar en una solución óptima. Entonces observaremos que solo necesitamos considerar una elección, la elección codiciosa, y que cuando hacemos la elección codiciosa, una de se garantiza que los subproblemas estarán vacíos, de modo que solo quede un subproblema no vacío. Con base en estas observaciones, desarrollaremos un algoritmo codicioso recursivo para resolver la Problema de programación de actividades. Completaremos el proceso de desarrollo de una solución codiciosa. convirtiendo el algoritmo recursivo en uno iterativo. Aunque los pasos que iremos en esta sección están más involucrados de lo que es típico para el desarrollo de un codicioso algoritmo, ilustran la relación de algoritmos codiciosos y programación dinámica.

La subestructura óptima del problema de selección de actividades

Como se mencionó anteriormente, comenzamos por desarrollar una solución de programación dinámica para la actividad-problema de selección. Como en el Capítulo 15 , nuestro primer paso es encontrar la subestructura óptima y luego Úselo para construir una solución óptima al problema, desde soluciones óptimas hasta subproblemas.

Vimos en el Capítulo 15 que necesitamos definir un espacio apropiado de subproblemas. Nos deja empezar por definir conjuntos

$S_{ij} = \{ a_k : f_{yo} \le s_k < f_k \le s_j \}$,

de modo que S_{ij} es el subconjunto de actividades en S que pueden comenzar después de que la actividad a_i finaliza y terminar antes comienza la actividad a_j . De hecho, S_{ij} consta de todas las actividades que son compatibles con a_i y a_j y son también compatible con todas las actividades que terminan antes de que termine *una* $_i$ y todas las actividades que comienzan no antes de que comience *una* $_j$. Para representar el problema completo, agregamos actividades ficticias a_0 y a_{n+1} y adoptan las convenciones de que $f_0 = 0$ y $s_{n+1} = \infty$. Entonces $S = S_{0,n+1}$, y los rangos para i y j están dados por $0 \le i, j \le n + 1$.

Podemos restringir aún más los rangos de i y j *de la* siguiente manera. Supongamos que las actividades son ordenados en orden creciente monótona de tiempo de finalización:

(16,1)

Página 319

Afirmamos que $S_{ij} = Ø$ siempre que $i \ge j$. ¿Por qué? Supongamos que existe una actividad $a_k \in S_{ij}$ para algunos $i \ge j$, de modo que a_i sigue a_j en el orden ordenado. Entonces tendríamos $f_i \le s_k < f_k \le s_j < f_j$. Por lo tanto, $f_i < f_j$, lo que contradice nuestra suposición de que a_i sigue a_j en el orden ordenado. Podemos concluir que, asumiendo que hemos ordenado las actividades en orden creciente monótona del tiempo de finalización, nuestro espacio de subproblemas es seleccionar un subconjunto de tamaño máximo de actividades compatibles de S_{ij} , para $0 \le i < j \le n + 1$, sabiendo que todos los demás S_{ij} están vacíos.

Para ver la subestructura del problema de selección de actividades, considere algunos subproblema S_{ij} , [1] y suponga que una solución a S_{ij} incluye alguna actividad a_k , de modo que $f_i \le s_k < f_k \le s_j$. Uso de la actividad *de un* $_k$ genera dos subproblemas, S_{ik} (actividades que se inician después de *un* $_i$ acabados y terminar antes de que comience *una* $_k$) y S_{kj} (actividades que comienzan después de que termina *una* $_k$ y terminan antes de que comience *una* $_j$), cada uno de los cuales consta de un subconjunto de las actividades en S_{ij} . Nuestra solución a S_{ij} es la unión del soluciones de S_{ik} y S_{kj} , junto con la actividad a_k . Así, el número de actividades en nuestro solución a S_{ij} es el tamaño de nuestra solución a S_{ik} , más el tamaño de nuestra solución a S_{kj} , más uno (para a_k).

La subestructura óptima de este problema es la siguiente. Supongamos ahora que una solución óptima A_{ij} a S_{ij} incluye la actividad a_k . Entonces las soluciones A_{ik} a S_{ik} y A_{kj} a $S_{kj\,se}$ utilizan dentro de este óptimo La solución a $S_{ij\,también}$ debe ser óptima. Se aplica el argumento habitual de cortar y pegar. Si tuviéramos un solución a S_{ik} que incluía más actividades que A_{ik} , podríamos cortar A_{ik} de A_{ij} y pegar in, produciendo así otra solución para S_{ij} con más actividades que A_{ij} . Porque nosotros Suponiendo que A_{ij} es una solución óptima, hemos derivado una contradicción. Del mismo modo, si tuviéramos un solución a S_{kj} con más actividades que A_{kj} , podríamos reemplazar A_{kj} por para producir una solución a S_{ij} con más actividades que A_{ij} .

Ahora usamos nuestra subestructura óptima para demostrar que podemos construir una solución óptima para problema desde soluciones óptimas hasta subproblemas. Hemos visto que cualquier solución a un El subproblema no vacío S_{ij} incluye alguna actividad a_k , y que cualquier solución óptima contiene dentro de él, soluciones óptimas para las instancias de subproblemas S_{ik} y S_{kj} . Por lo tanto, podemos construir un subconjunto de tamaño máximo de actividades mutuamente compatibles en S_{ij} dividiendo el problema en dos subproblemas (encontrar subconjuntos de tamaño máximo de actividades mutuamente compatibles en S_{ik} y S_{kj}), encontrar subconjuntos de tamaño máximo A_{ik} y A_{kj} de actividades mutuamente compatibles para estos subproblemas, y formando nuestro subconjunto de tamaño máximo A_{ij} de actividades mutuamente compatibles como

(16,2)

Una solución óptima para todo el problema es una solución a $S_{0,n+1}$.

Una solución recursiva

El segundo paso en el desarrollo de una solución de programación dinámica es definir recursivamente el valor de una solución óptima. Para el problema de selección de actividad, dejamos que $c[i, j]$ sea el número de actividades en un subconjunto de tamaño máximo de actividades mutuamente compatibles en S_{ij} . Tenemos $c[i, j] =$ 0 siempre que $S_{ij} = Ø$; en particular, $c[i, j] = 0$ para $i \ge j$.

Ahora considere un subconjunto no vacío S_{ij} . Como hemos visto, si *un* $_k$ se utiliza en un subconjunto de tamaño máximo de actividades mutuamente compatibles de S_{ij} , también usamos subconjuntos de tamaño máximo de mutuamente actividades compatibles para los subproblemas S_{ik} y S_{kj} . Usando la ecuación (16.2) , tenemos la reaparición

Página 320

$c[i, j] = c[i, k] + c[k, j] + 1$.

Esta ecuación recursiva supone que conocemos el valor de k, lo que no sabemos. Hay $j - i - 1$ valores posibles para k, a saber, $k = i + 1, \ldots, j - 1$. Dado que el subconjunto de tamaño máximo de S_{ij} debe use uno de estos valores para k, los revisamos todos para encontrar el mejor. Por lo tanto, nuestro completo recursivo la definición de $c[i, j]$ se convierte en

(16,3)

Convertir una solución de programación dinámica en una solución codiciosa

En este punto, sería un ejercicio sencillo escribir una tabla, de abajo hacia arriba, dinámica algoritmo de programación basado en la recurrencia (16.3). De hecho, el ejercicio 16.1-1 le pide que haga solo eso. Sin embargo, hay dos observaciones clave que nos permiten simplificar nuestra solución.

Teorema 16.1

Considere cualquier subproblema no vacío S_{ij}, y sea a_m la actividad en S_{ij} con el final más temprano hora:

$f_m = \min \{ f_k : a_k \quad S_{ij} \}$.

Luego

1. La actividad a_m se usa en algún subconjunto de tamaño máximo de actividades mutuamente compatibles de S_{ij}.
2. El subproblema S_{im} está vacío, por lo que elegir a_m deja el subproblema S_{mj} como el sólo uno que puede no estar vacío.

Prueba Primero probaremos la segunda parte, ya que es un poco más simple. Supongamos que S_{im} es no vacío, de modo que hay alguna actividad a_k tal que $f_i \le s_k < f_k \le s_m < f_m$. Entonces a_k también está en S_{ij} y tiene un tiempo de finalización anterior a *una* $_m$, lo que contradice nuestra elección de *una* $_m$. Concluimos que S_{im} está vacío.

Para probar la primera parte, suponemos que A_{ij} es un subconjunto de tamaño máximo de actividades de S_{ij}, y ordenemos las actividades en A_{ij} en orden creciente monótona de finalización hora. Sea a_k la primera actividad en A_{ij}. Si $a_k = a_m$, hemos terminado, ya que hemos demostrado que a_m es utilizado en algún subconjunto de tamaño máximo de actividades mutuamente compatibles de S_{ij}. Si $a_k \ne a_m$, nosotros construir el subconjunto Las actividades en son inconexas, ya que las actividades en A_{ij} son, a_k es la primera actividad en A_{ij} en terminar, y $f_m \le f_k$. Observando que tiene el mismo número de actividades como A_{ij}, vemos que es un subconjunto de tamaño máximo de actividades mutuamente compatibles de S_{ij} que incluye *una* $_m$.

¿Por qué es tan valioso el teorema 16.1 ? Recuerde de la Sección 15.3 que la subestructura óptima varía en cuántos subproblemas se utilizan en una solución óptima al problema original y en cómo

Página 321

tenemos muchas opciones para determinar qué subproblemas utilizar. En nuestra dinámica solución de programación, se utilizan dos subproblemas en una solución óptima, y hay $j - i - 1$ opciones al resolver el subproblema S_{ij}. El teorema 16.1 reduce ambas cantidades significativamente: solo se utiliza un subproblema en una solución óptima (el otro subproblema es garantizado que está vacío), y al resolver el subproblema S_{ij}, necesitamos considerar solo una

elección: la que tiene el tiempo de finalización más temprano en S_{ij} . Afortunadamente, podemos determinar fácilmente qué actividad que es.

Además de reducir el número de subproblemas y el número de opciones, el teorema 16.1 produce otro beneficio: podemos resolver cada subproblema de arriba hacia abajo, en lugar de la forma de abajo hacia arriba que se utiliza normalmente en la programación dinámica. Para resolver el subproblema S_{ij} , Elija la actividad a_m en S_{ij} con el tiempo de finalización más temprano y agregue a esta solución el conjunto de actividades utilizadas en una solución óptima al subproblema S_{ij} . Porque sabemos que tener elegido un_m , ciertamente estaremos usando una solución para S_{mj} en nuestra solución óptima para S_{ij} , no Necesito resolver S_{mj} *antes de* resolver S_{ij} . Para resolver S_{ij} , *primero* podemos elegir a_m como la actividad en S_{ij} con el tiempo de finalización más temprano y *luego* resuelva S_{mj} .

Tenga en cuenta también que hay un patrón en los subproblemas que resolvemos. Nuestro problema original es $S = S_{0,n+1}$. Supongamos que elegimos como actividad en $S_{0,n+1}$ con el tiempo de finalización más temprano. (Ya que Hemos ordenado las actividades aumentando monótonamente los tiempos de finalización y $f_0 = 0$, debemos tener $m_1 = 1$.) Nuestro siguiente subproblema es Ahora suponga que elegimos como actividad en con el tiempo de finalización más temprano. (No es necesariamente el caso de que $m_2 = 2$.) Nuestro siguiente subproblema es . Continuando, vemos que cada subproblema será de la forma para alguna actividad número m_i . En otras palabras, cada subproblema consta de las últimas actividades en terminar, y el El número de tales actividades varía de un subproblema a otro.

También hay un patrón para las actividades que elegimos. Porque siempre elegimos la actividad con el tiempo de finalización más temprano en , los tiempos de finalización de las actividades elegidas sobre todas los subproblemas irán aumentando estrictamente con el tiempo. Además, podemos considerar cada actividad sólo una vez en general, en un orden monótonamente creciente de tiempos de finalización.

La actividad a_m que elegimos al resolver un subproblema es siempre la que tiene la primera hora de finalización que se puede programar legalmente. La actividad elegida es, por tanto, una elección "codiciosa" en el sentir que, intuitivamente, deja tantas oportunidades como sea posible para que las actividades restantes ser programado. Es decir, la elección codiciosa es la que maximiza la cantidad de tiempo restante.

Un algoritmo codicioso recursivo

Ahora que hemos visto cómo optimizar nuestra solución de programación dinámica y cómo tratarlo como un método de arriba hacia abajo, estamos listos para ver un algoritmo que funciona de una manera puramente codiciosa, moda de arriba hacia abajo. Damos una solución sencilla y recursiva como procedimiento SELECTOR-ACTIVIDAD-RECURSIVA. Toma las horas de inicio y finalización de las actividades, representado como matrices s y f , así como los índices de partida i y j de la subproblema $S_{i,j}$ es para resolver. Devuelve un conjunto de tamaño máximo de actividades mutuamente compatibles en $S_{i,j}$. Asumimos que las n actividades de entrada se ordenan aumentando monótonamente el tiempo de finalización, de acuerdo con ecuación (16.1). Si no, podemos clasificarlos en este orden en $O(n \lg n)$ tiempo, rompiendo los lazos arbitrariamente. La llamada inicial es SELECTOR-ACTIVIDAD-RECURSIVA (s, f , 0, n + 1).

SELECTOR-ACTIVIDAD-RECURSIVA (s, f, i, j)

Página 322

1 $m \leftarrow i + 1$
2 **, mientras que** $m < j$ y $s_m < f_i$ ▸ encontrar la primera actividad en S_{ij} .
3 **hacer** $m \leftarrow m + 1$
4 **si** $m < j$
5 **luego devuelve** { a_m } SELECTOR-ACTIVIDAD-RECURSIVA (s, f, m, j)
6 **más volver** Ø

La figura 16.1 muestra el funcionamiento del algoritmo. En una llamada recursiva dada RECURSIVE-ACTIVIDAD-SELECTOR (s, f, i, j), el **tiempo** de bucle de las líneas 2-3 busca la primera actividad en S_{ij} . El bucle examina a_{i+1} , a_{i+2} , ..., a_{j-1} , hasta que encuentra la primera actividad a_m que es compatible con a_i ; tal actividad tiene $s_m \geq f_i$. Si el ciclo termina porque encuentra tal actividad, el El procedimiento devuelve en la línea 5 la unión de { a_m } y el subconjunto de tamaño máximo de S_{mj} devuelto por la llamada recursiva RECURSIVE-ACTIVITY-SELECTOR (s, f, m, j). Alternativamente, el bucle puede terminar porque $m \geq j$, en cuyo caso hemos examinado todas las actividades cuyo final los tiempos son anteriores al de una_j sin encontrar uno que sea compatible con una_i . En este caso, S_{ij} = Ø, y entonces el procedimiento devuelve Ø en la línea 6.

Figura 16.1: El funcionamiento del SELECTOR-ACTIVIDAD-RECURSIVA en las 11 actividades dado antes. Las actividades consideradas en cada llamada recursiva aparecen entre líneas horizontales. los actividad ficticia a_0 termina en el tiempo 0, y en la llamada inicial, ACTIVIDAD-RECURSIVA-SELECTOR (s, f , 0, 12), se selecciona la actividad a_1 . En cada llamada recursiva, las actividades que tienen ya seleccionados están sombreados y se está considerando la actividad mostrada en blanco. Si el la hora de inicio de una actividad ocurre antes de la hora de finalización de la actividad agregada más recientemente

Página 323

(la flecha entre ellos apunta a la izquierda), se rechaza. De lo contrario (la flecha apunta directamente hacia arriba o a la derecha), está seleccionado. La última llamada recursiva, RECURSIVE-ACTIVITY-SELECTOR (s, f , 11, 12), devuelve Ø. El conjunto resultante de actividades seleccionadas es $\{ a_1 , a_4 , a_8 , a_{11} \}$.

Suponiendo que las actividades ya han sido ordenadas por tiempos de finalización, el tiempo de ejecución del llamar RECURSIVE-ACTIVITY-SELECTOR (s, f , 0, n + 1) es $\Theta (n)$, que podemos ver como sigue. Durante todas las llamadas recursivas, cada actividad se examina exactamente una vez en el **tiempo** de prueba de lazo de la línea 2. En particular, la actividad *ak* se examina en la última llamada realizada en la que $i < k$.

Un algoritmo codicioso iterativo

Podemos convertir fácilmente nuestro procedimiento recursivo en uno iterativo. El procedimiento RECURSIVE-ACTIVITY-SELECTOR es casi "cola recursiva" (vea el problema 7-4): termina con una llamada recursiva a sí mismo seguida de una operación sindical. Suele ser una tarea sencilla para transformar un procedimiento recursivo de cola en una forma iterativa; de hecho, algunos compiladores para ciertos Los lenguajes de programación realizan esta tarea automáticamente. Como está escrito, RECURSIVO-ACTIVITY-SELECTOR funciona para cualquier subproblema S_{ij} , pero hemos visto que necesitamos considere solo los subproblemas para los cuales $j = n + 1$, es decir, los subproblemas que consisten en el último actividades para terminar.

El procedimiento GREEDY-ACTIVITY-SELECTOR es una versión iterativa del procedimiento SELECTOR-ACTIVIDAD-RECURSIVA. También asume que las actividades de entrada están ordenadas aumentando monótonamente el tiempo de finalización. Recopila actividades seleccionadas en un conjunto A y devuelve este conjunto cuando esté hecho.

```
SELECTOR-ACTIVIDAD-CODICION ( s, f )
1 n ← longitud [ s ]
2 A ← { a 1 }
3 i ← 1
4 para m ← 2 an
5 hacer si s m ≥ f i
6                    luego A ← A          { a m }
7                         yo ← m
8 devuelve A
```

El procedimiento funciona de la siguiente manera. La variable i indexa la adición más reciente a A ,

correspondiente a la actividad a_i en la versión recursiva. Dado que las actividades se consideran en orden de aumento monotónico del tiempo de finalización, f_i es siempre el tiempo de finalización máximo de cualquier actividad en A . Es decir,

(16,4)

Las líneas 2 a 3 seleccionan la actividad a_1 , inicializan A para que contenga solo esta actividad e inicializan i para indexar esta actividad. El bucle **for** de las líneas 4–7 encuentra la primera actividad que finaliza en $S_{i,n+1}$. El lazo considera cada actividad a_m por turno y agrega una_m a A si es compatible con todas las anteriores actividades seleccionadas; tal actividad es la más temprana en terminar en $S_{i,n+1}$. Para ver si la actividad a_m es compatible con todas las actividades actualmente en A , basta con la ecuación (16.4) para verificar (línea 5) que su hora de inicio s_m no es anterior a la hora de finalización f_i de la actividad añadida más recientemente a *Una* . Si la actividad a_m es compatible, entonces las líneas 6 a 7 agregan la actividad a_m a A y establecen i a m . El conjunto A devuelto por la llamada GREEDY-ACTIVITY-SELECTOR (s, f) es precisamente el conjunto devuelto por la llamada SELECTOR-ACTIVIDAD-RECURSIVA (s, f , 0, n + 1).

Página 324

Al igual que la versión recursiva, GREEDY-ACTIVITY-SELECTOR programa un conjunto de n actividades en Θ (n) tiempo, asumiendo que las actividades ya estaban ordenadas inicialmente por sus tiempos de finalización.

Ejercicios 16.1-1

Dar un algoritmo de programación dinámica para el problema de selección de actividad, basado en recurrencia (16,3). Haga que su algoritmo calcule los tamaños c [i, j] como se definió anteriormente y también producir el subconjunto A de tamaño máximo de actividades. Suponga que las entradas se han ordenado como en la ecuación (16.1). Compare el tiempo de ejecución de su solución con el tiempo de ejecución de SELECTOR-ACTIVIDAD-CODICION.

Ejercicios 16.1-2

Supongamos que en lugar de seleccionar siempre la primera actividad para finalizar, seleccionamos la última actividad para iniciar que sea compatible con todas las actividades previamente seleccionadas. Describe cómo esto El enfoque es un algoritmo codicioso y demuestra que produce una solución óptima.

Ejercicios 16.1-3

Supongamos que tenemos un conjunto de actividades para programar entre un gran número de salas de conferencias. Nosotros Desea programar todas las actividades utilizando el menor número posible de salas de conferencias. Dar un eficiente algoritmo codicioso para determinar qué actividad debe utilizar qué sala de conferencias.

(Esto también se conoce como el problema de coloración de gráficos de intervalos . Podemos crear un gráfico de intervalos cuyos vértices son las actividades dadas y cuyas aristas conectan actividades incompatibles. los el menor número de colores necesarios para colorear cada vértice de modo que no haya dos vértices adyacentes dado el mismo color corresponde a encontrar la menor cantidad de salas de conferencias necesarias para programar todos los las actividades dadas.)

Ejercicios 16.1-4

No cualquier enfoque codicioso del problema de selección de actividades produce un conjunto de tamaño máximo de actividades mutuamente compatibles. Dé un ejemplo para mostrar que el enfoque de seleccionar el actividad de menor duración de aquellas que sean compatibles con actividades previamente seleccionadas No funciona. Haga lo mismo para los enfoques de seleccionar siempre la actividad compatible que superpone la menor cantidad de otras actividades restantes y siempre seleccionando el compatible actividad restante con la hora de inicio más temprana.

Página 325

[1] A veces hablaremos de los conjuntos S_{ij} como subproblemas en lugar de simplemente conjuntos de actividades. Eso Siempre quedará claro del contexto si nos referimos a S_{ij} como un conjunto de actividades o subproblema cuya entrada es ese conjunto.

16.2 Elementos de la estrategia codiciosa

Un algoritmo codicioso obtiene una solución óptima a un problema haciendo una secuencia de opciones. Para cada punto de decisión del algoritmo, la elección que parece mejor en este momento es elegido. Esta estrategia heurística no siempre produce una solución óptima, pero como vimos en el problema de la selección de actividades, a veces lo hace. Esta sección analiza algunas de las propiedades de los métodos codiciosos.

El proceso que seguimos en la Sección 16.1 para desarrollar un algoritmo codicioso fue un poco más involucrado de lo que es típico. Pasamos por los siguientes pasos:

1. Determine la subestructura óptima del problema.
2. Desarrolle una solución recursiva.
3. Demuestre que en cualquier etapa de la recursividad, una de las opciones óptimas es la codiciosa elección. Por lo tanto, siempre es seguro tomar una decisión codiciosa.
4. Muestre que todos menos uno de los subproblemas inducidos por haber tomado la decisión codiciosa están vacíos.
5. Desarrolle un algoritmo recursivo que implemente la estrategia codiciosa.
6. Convierta el algoritmo recursivo en un algoritmo iterativo.

Al seguir estos pasos, vimos con gran detalle los fundamentos de la programación dinámica de un algoritmo codicioso. En la práctica, sin embargo, normalmente simplificamos los pasos anteriores cuando diseñando un algoritmo codicioso. Desarrollamos nuestra subestructura con miras a hacer una elección codiciosa que deja solo un subproblema para resolver de manera óptima. Por ejemplo, en la actividad-problema de selección, primero definimos los subproblemas S_{ij}, donde tanto i como j variaban. Nosotros entonces Descubrí que si siempre tomábamos la decisión codiciosa, podríamos restringir los subproblemas para que fueran de la forma $S_{i,n+1}$.

Alternativamente, podríamos haber diseñado nuestra subestructura óptima con una elección codiciosa en mente. Es decir, podríamos haber eliminado el segundo subíndice y haber definido subproblemas del forma $S_i = \{ a_k \in S : f_{yo} \le s_k \}$. Entonces, podríamos haber demostrado que una elección codiciosa (la primera actividad a_m para terminar en S_i), combinado con una solución óptima para el conjunto restante S_m de compatible actividades, produce una solución óptima para S_i. De manera más general, diseñamos algoritmos codiciosos según la siguiente secuencia de pasos:

1. Considere el problema de optimización como uno en el que tomamos una decisión y nos queda una subproblema a resolver.
2. Demuestre que siempre hay una solución óptima al problema original que hace que elección codiciosa, para que la elección codiciosa sea siempre segura.
3. Demuestre que, habiendo tomado la decisión codiciosa, lo que queda es un subproblema con la propiedad de que si combinamos una solución óptima al subproblema con el codicioso elección que hemos hecho, llegamos a una solución óptima al problema original.

Usaremos este proceso más directo en secciones posteriores de este capítulo. Sin embargo, debajo cada algoritmo codicioso, casi siempre hay una programación dinámica más engorrosa solución.

Página 326

¿Cómo se puede saber si un algoritmo codicioso resolverá un problema de optimización particular? Ahi esta de ninguna manera en general, pero la propiedad de elección codiciosa y la subestructura óptima son las dos claves Ingredientes Si podemos demostrar que el problema tiene estas propiedades, entonces estamos bien

la forma de desarrollar un algoritmo codicioso para ello.

Propiedad de elección codiciosa

El primer ingrediente clave es la *propiedad de elección codiciosa* : una solución globalmente óptima puede ser llegó tomando una elección localmente óptima (codiciosa). En otras palabras, cuando estamos considerando qué elección tomar, tomamos la decisión que se ve mejor en el problema actual, sin considerar los resultados de los subproblemas.

Aquí es donde los algoritmos codiciosos se diferencian de la programación dinámica. En dinámica programación, hacemos una elección en cada paso, pero la elección generalmente depende de las soluciones a los subproblemas. En consecuencia, normalmente resolvemos problemas de programación dinámica en un de abajo hacia arriba, pasando de subproblemas más pequeños a subproblemas más grandes. En un codicioso algoritmo, tomamos la decisión que nos parezca mejor en este momento y luego resolvemos el subproblema que surja después de que se haga la elección. La elección hecha por un algoritmo codicioso puede depender de opciones hasta ahora, pero no puede depender de opciones futuras o de las soluciones a los subproblemas. Así, a diferencia de la programación dinámica, que resuelve los subproblemas de abajo hacia arriba, un codicioso La estrategia generalmente avanza de arriba hacia abajo, haciendo una elección codiciosa tras otra, reduciendo cada instancia de problema dada a una más pequeña.

Por supuesto, debemos demostrar que una elección codiciosa en cada paso produce una solución globalmente óptima, y aquí es donde puede ser necesaria la inteligencia. Típicamente, como en el caso del teorema 16.1 , el La prueba examina una solución globalmente óptima para algún subproblema. Luego muestra que el La solución se puede modificar para usar la opción codiciosa, lo que resulta en una solución similar pero más pequeña. subproblema.

La propiedad de elección codiciosa a menudo nos otorga cierta eficiencia al hacer nuestra elección en un subproblema. Por ejemplo, en el problema de selección de actividad, suponiendo que ya clasificamos las actividades en un orden cada vez mayor de tiempos de finalización, necesitábamos examinar cada actividad solo una vez. Con frecuencia ocurre que al preprocesar la entrada o al utilizar un estructura de datos apropiada (a menudo una cola de prioridad), podemos tomar decisiones codiciosas rápidamente, por lo tanto produciendo un algoritmo eficiente.

Subestructura óptima

Un problema presenta *una subestructura óptima* si una solución óptima al problema contiene dentro de ella soluciones óptimas a subproblemas. Esta propiedad es un ingrediente clave para evaluar la aplicabilidad de programación dinámica así como algoritmos codiciosos. Como ejemplo de subestructura óptima, recuerde cómo demostramos en la Sección 16.1 que si una solución óptima Si el subproblema S_{ij} incluye una actividad a_k , entonces también debe contener soluciones óptimas para subproblemas S_{ik} y S_{kj} . Dada esta subestructura óptima, argumentamos que si supiéramos qué actividad para utilizar como una_k , podríamos construir una solución óptima a S_{ij} mediante la selección de un_k junto con todos actividades en soluciones óptimas a los subproblemas S_{ik} y S_{kj} . Basado en esta observación de subestructura óptima, pudimos idear la recurrencia (16.3) que describía el valor de una solución óptima.

Página 327

Usualmente usamos un enfoque más directo con respecto a la subestructura óptima cuando lo aplicamos a algoritmos codiciosos. Como se mencionó anteriormente, tenemos el lujo de asumir que llegamos a un subproblema por haber hecho la elección codiciosa en el problema original. Todo lo que realmente necesitamos hacer es argumentar que una solución óptima al subproblema, combinada con la elección codiciosa ya hecho, da una solución óptima al problema original. Este esquema utiliza implícitamente inducción en los subproblemas para demostrar que hacer la elección codiciosa en cada paso produce una solución óptima.

Programación codiciosa versus dinámica

Debido a que la propiedad de la subestructura óptima es explotada tanto por los codiciosos como por los dinámicos estrategias de programación, uno podría tener la tentación de generar una solución de programación dinámica a un problema cuando una solución codiciosa es suficiente, o uno podría pensar erróneamente que un codicioso La solución funciona cuando en realidad se requiere una solución de programación dinámica. Para ilustrar el sutilezas entre las dos tcnicas, investiguemos dos variantes de un clsico problema de optimizacion.

El problema de la mochila 0-1 se plantea de la siguiente manera. Un ladrón que roba una tienda encuentra n artículos; el i th el artículo vale v_i dólares y pesa w_i libras, donde v_i y w_i son números enteros. El quiere tomar carga lo más valiosa posible, pero puede llevar como máximo W libras en su mochila por un tiempo número entero W . ¿Qué artículos debería llevarse? (Esto se llama el problema de la mochila 0-1 porque cada artículo debe ser tomado o dejado atrás; el ladrón no puede tomar una fracción de un artículo o tomar un artículo más de una vez).

En el problema de la mochila fraccionada , la configuración es la misma, pero el ladrón puede tomar fracciones de elementos, en lugar de tener que hacer una elección binaria (0-1) para cada elemento. Puedes pensar en un artículo en el problema de la mochila 0-1 como si fuera un lingote de oro, mientras que un artículo en la fracción El problema de la mochila se parece más al polvo de oro.

Ambos problemas de mochila exhiben la propiedad de subestructura óptima. Para el problema 0-1, considere la carga más valiosa que pesa como máximo W libras. Si eliminamos el elemento j de este carga, la carga restante debe ser la carga más valiosa con un peso máximo de $W - w_j$ que el ladrón puede tomar de los $n - 1$ artículos originales excluyendo j . Para el problema fraccional comparable, considere que si quitamos un peso w de un artículo j de la carga óptima, la carga restante debe ser la carga más valiosa con un peso máximo de $W - w$ que el ladrón pueda tomar del $n - 1$ artículos originales más $w_j - w$ libras del artículo j .

Aunque los problemas son similares, el problema de la mochila fraccionada se puede resolver con un codicioso estrategia, mientras que el problema 0-1 no lo es. Para resolver el problema fraccional, primero calculamos el valor por libra v_i / w_i para cada artículo. Obedeciendo una estrategia codiciosa, el ladrón comienza tomando tanto como sea posible del artículo con el mayor valor por libra. Si el suministro de ese artículo es agotado y todavía puede cargar más, toma la mayor cantidad posible del artículo con el siguiente mayor valor por libra, y así sucesivamente hasta que no pueda cargar más. Por lo tanto, al ordenar los elementos por valor por libra, el algoritmo codicioso se ejecuta en $O(n \lg n)$ tiempo. La prueba de que lo fraccionario El problema de la mochila tiene la propiedad de elección codiciosa que se deja como ejercicio 16.2-1 .

Para ver que esta estrategia codiciosa no funciona para el problema de la mochila 0-1, considere la ejemplo de problema ilustrado en la figura 16.2 (a) . Hay 3 elementos y la mochila puede contener 50 libras. El artículo 1 pesa 10 libras y vale 60 dólares. El artículo 2 pesa 20 libras y es vale 100 dólares. El artículo 3 pesa 30 libras y vale 120 dólares. Así, el valor por

Página 328

libra del artículo 1 es de 6 dólares por libra, que es mayor que el valor por libra de artículo 2 (5 dólares por libra) o artículo 3 (4 dólares por libra). La estrategia codiciosa, por tanto, tomaría el artículo 1 primero. Sin embargo, como puede verse en el análisis de caso de la Figura 16.2 (b) , La solución óptima toma los elementos 2 y 3, dejando 1 atrás. Las dos posibles soluciones que implican el artículo 1 son ambos subóptimos.

Figura 16.2: La estrategia codiciosa no funciona para el problema de la mochila 0-1. *(a)* El ladrón debe seleccionar un subconjunto de los tres artículos mostrados cuyo peso no debe exceder las 50 libras. *(si)* El subconjunto óptimo incluye los elementos 2 y 3. Cualquier solución con el elemento 1 es subóptima, incluso aunque el artículo 1 tiene el mayor valor por libra. *(c)* Para el problema de la mochila fraccionada, tomar los artículos en orden de mayor valor por libra produce una solución óptima.

Para el problema fraccional comparable, sin embargo, la estrategia codiciosa, que toma el elemento 1 primero, produce una solución óptima, como se muestra en la figura 16.2 (c) . Tomar el elemento 1 no funciona en el 0-1 problema porque el ladrón no puede llenar su mochila al máximo de su capacidad y el espacio vacío reduce el valor efectivo por libra de su carga. En el problema 0-1, cuando consideramos un artículo para su inclusión en la mochila, debemos comparar la solución con el subproblema en el que el artículo se incluye con la solución al subproblema en el que se excluye el artículo antes podemos tomar la decisión. El problema formulado de esta manera da lugar a muchas superposiciones subproblemas: un sello distintivo de la programación dinámica y, de hecho, la programación dinámica puede utilizarse para resolver el problema 0-1. (Vea el ejercicio 16.2-2 .)

Ejercicios 16.2-1

Demuestre que el problema de la mochila fraccionada tiene la propiedad de elección codiciosa.

Ejercicios 16.2-2

Dar una solución de programación dinámica al problema de la mochila 0-1 que se ejecuta en $O(nW)$ tiempo, donde n es el número de elementos y W es el peso máximo de elementos que el ladrón puede poner en su mochila.

Ejercicios 16.2-3

Suponga que en un problema de mochila 0-1, el orden de los elementos cuando se clasifican aumentando el peso es el mismo que su orden cuando se ordena por valor decreciente. Dar un algoritmo eficiente

Página 329

para encontrar una solución óptima a esta variante del problema de la mochila, y argumentar que su el algoritmo es correcto.

Ejercicios 16.2-4

El profesor Midas conduce un automóvil de Newark a Reno por la Interestatal 80. El combustible de su automóvil tanque, cuando está lleno, tiene suficiente gasolina para viajar n millas, y su mapa da las distancias entre gasolineras en su ruta. El profesor desea hacer el menor número posible de paradas de gas a lo largo del camino. Dar un método eficiente mediante el cual el profesor Midas pueda determinar en qué gasolineras debería detenerse y demostrar que su estrategia produce una solución óptima.

Ejercicios 16.2-5

Describe un algoritmo eficiente que, dado un conjunto $\{x_1, x_2, ..., x_n\}$ de puntos en la línea real, determina el conjunto más pequeño de intervalos cerrados de longitud unitaria que contiene todos los puntos dados. Argumenta que tu algoritmo es correcto.

Ejercicios 16.2-6:

Muestre cómo resolver el problema de la mochila fraccionaria en $O(n)$ tiempo. Suponga que tiene un solución al problema 9-2 .

Ejercicios 16.2-7

Suponga que le dan dos conjuntos A y B , cada uno de los cuales contiene n números enteros positivos. Tu puedes elegir para reordenar cada juego como quieras. Después de reordenar, sea a_i el elemento i del conjunto A , y dejar que b_i ser el i -ésimo elemento del conjunto B . Luego recibe una recompensa de $\prod_{i=1}^{n} a_i^{b_i}$. Dar un algoritmo que maximizará su beneficio. Demuestre que su algoritmo maximiza la recompensa y establezca su tiempo de ejecución.

16.3 Códigos Huffman

Los códigos de Huffman son una técnica muy utilizada y muy eficaz para comprimir datos; ahorros de 20% a 90% son típicos, dependiendo de las características de los datos que se comprimen. Nosotros

considere los datos como una secuencia de caracteres. El codicioso algoritmo de Huffman utiliza una tabla de las frecuencias de ocurrencia de los personajes para construir una forma óptima de representar cada carácter como una cadena binaria.

Página 330

Supongamos que tenemos un archivo de datos de 100.000 caracteres que deseamos almacenar de forma compacta. Observamos que los caracteres en el archivo ocurren con las frecuencias dadas por la Figura 16.3 . Es decir, solo seis aparecen diferentes caracteres, y el carácter a aparece 45.000 veces.

	a	**B**	**C**	**D**	**e**	**F**
Frecuencia (en miles)	45	13	12	16	9	5
Palabra de código de longitud fija	00000	101	011	100	101	
Palabra de código de longitud variable	0	101	100	111	1101	1100

Figura 16.3: Un problema de codificación de caracteres. Un archivo de datos de 100.000 caracteres contiene solo el caracteres a – f, con las frecuencias indicadas. Si a cada carácter se le asigna una palabra de código de 3 bits, el archivo se puede codificar en 300.000 bits. Usando el código de longitud variable que se muestra, el archivo puede ser codificado en 224.000 bits.

Hay muchas formas de representar un archivo de información de este tipo. Consideramos el problema de diseñar un *código de carácter binario* (o *código* para abreviar) en el que se representa cada carácter por una cadena binaria única. Si usamos un *código de longitud fija* , necesitamos 3 bits para representar seis caracteres: a = 000, b = 001, ..., f = 101. Este método requiere 300.000 bits para codificar el expediente. ¿Podemos hacerlo mejor?

Un *código de longitud variable* puede funcionar considerablemente mejor que un código de longitud fija, dando caracteres frecuentes palabras de código cortas y caracteres poco frecuentes palabras de código largas. Figura 16.3 muestra tal código; aquí, la cadena de 1 bit 0 representa a, y la cadena de 4 bits 1100 representa f. Este código requiere

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

para representar el archivo, un ahorro de aproximadamente un 25%. De hecho, este es un código de carácter óptimo. para este archivo, como veremos.

Códigos de prefijo

Consideramos aquí solo códigos en los que ninguna palabra de código es también un prefijo de alguna otra palabra de código. Estos códigos se denominan *códigos de prefijo* . [2] Es posible mostrar (aunque no lo haremos aquí) que la compresión de datos óptima que se puede lograr mediante un código de caracteres siempre se puede lograr con un código de prefijo, por lo que no hay pérdida de generalidad al restringir la atención a los códigos de prefijo.

La codificación es siempre sencilla para cualquier código de carácter binario; simplemente concatenamos las palabras de código que representa cada carácter del archivo. Por ejemplo, con el código de prefijo de longitud variable de Figura 16.3 , codificamos el archivo de 3 caracteres abc como $0 \cdot 101 \cdot 100 = 0101100$, donde usamos "$\cdot$" para denotar concatenación.

Los códigos de prefijo son deseables porque simplifican la decodificación. Dado que ninguna palabra en clave es un prefijo de cualquier otra, la palabra de código que comienza un archivo codificado es inequívoca. Simplemente podemos identificar la palabra clave inicial, traducirla de nuevo al carácter original y repetir la decodificación

Página 331

proceso en el resto del archivo codificado. En nuestro ejemplo, la cadena 001011101 analiza
únicamente como 0 - 0 - 101 - 1101, que decodifica a aabe.

El proceso de decodificación necesita una representación conveniente del código de prefijo para que la inicial
La palabra clave se puede seleccionar fácilmente. Un árbol binario cuyas hojas son los caracteres dados.
proporciona una de esas representaciones. Interpretamos la palabra de código binaria para un carácter como la ruta
de la raíz a ese carácter, donde 0 significa "ir al hijo de la izquierda" y 1 significa "ir al
niño correcto ". La figura 16.4 muestra los árboles para los dos códigos de nuestro ejemplo. Tenga en cuenta que estos son
no árboles de búsqueda binaria, ya que las hojas no necesitan aparecer en orden ordenado y los nodos internos sí
no contiene claves de caracteres.

Figura 16.4: Árboles correspondientes a los esquemas de codificación de la Figura 16.3. Cada hoja está etiquetada
con un personaje y su frecuencia de aparición. Cada nodo interno está etiquetado con la suma de
las frecuencias de las hojas en su subárbol. *(a)* El árbol correspondiente al código de longitud fija
a = 000, ..., f = 101. *(b)* El árbol correspondiente al código de prefijo óptimo a = 0, b = 101, ..., f
= 1100.

Un código óptimo para un archivo siempre está representado por un árbol binario *completo* , en el que cada no hoja
nodo tiene dos hijos (consulte el ejercicio 16.3-1). El código de longitud fija en nuestro ejemplo no es
óptimo ya que su árbol, que se muestra en la Figura 16.4 (a) , no es un árbol binario completo: hay palabras de código
comenzando 10 ..., pero ninguno comenzando 11 .... Ya que ahora podemos restringir nuestra atención al binario completo
árboles, podemos decir que si C es el alfabeto del que se extraen los caracteres y todos
las frecuencias de caracteres son positivas, entonces el árbol para un código de prefijo óptimo tiene exactamente $|C|$
hojas, una por cada letra del alfabeto, y exactamente $|C| - 1$ nodos internos (ver Ejercicio B.5-
3).

Dado un árbol T correspondiente a un código de prefijo, es una cuestión simple calcular el número de
bits necesarios para codificar un archivo. Para cada carácter c en el alfabeto C , sea $f(c)$ la
frecuencia de c en el archivo y sea $d_T(c)$ la profundidad de la hoja de c en el árbol. Tenga en cuenta que $d_T(c)$ es
también la longitud de la palabra de código para el carácter c . El número de bits necesarios para codificar un archivo es
así

(16,5)

que definimos como el coste del árbol T .

Construyendo un código Huffman

Huffman inventó un algoritmo codicioso que construye un código de prefijo óptimo llamado
Código Huffman . Manteniéndose en línea con nuestras observaciones en la Sección 16.2 , su prueba de corrección
se basa en la propiedad de elección codiciosa y la subestructura óptima. En lugar de demostrar eso

Página 332

estas propiedades se mantienen y luego, al desarrollar el pseudocódigo, primero presentamos el pseudocódigo. Haciendo
por lo que ayudará a aclarar cómo el algoritmo toma decisiones codiciosas.

En el pseudocódigo que sigue, asumimos que C es un conjunto de n caracteres y que cada
el carácter $c \in C$ es un objeto con una frecuencia definida $f[c]$. El algoritmo construye el árbol T
correspondiente al código óptimo de forma ascendente. Comienza con un conjunto de $|C|$ hojas
y realiza una secuencia de $|C| - 1$ operaciones de "fusión" para crear el árbol final. Una prioridad mínima
la cola Q , tecleada en f , se usa para identificar los dos objetos menos frecuentes que se fusionan. los
resultado de la fusión de dos objetos es un nuevo objeto cuya frecuencia es la suma de los
frecuencias de los dos objetos que se fusionaron.

```
HUFFMAN ( C )
1 n ← | C |
2 Q ← C
3 para i ← 1 a n - 1
4 hacer asignar un nuevo nodo z
```

```
5        izquierda[z] ← x ← EXTRACTO-MIN(Q)
6        derecha[z] ← y ← EXTRACTO-MIN(Q)
7        f[z] ← f[x] + f[y]
8        INSERTAR(Q, z)
9 return EXTRACT-MIN(Q) ▸ Return la raíz del árbol.
```

Para nuestro ejemplo, el algoritmo de Huffman procede como se muestra en la figura 16.5 . Dado que hay 6 letras en el alfabeto, el tamaño de la cola inicial es $n = 6$, y se requieren 5 pasos de combinación para construir el árbol. El árbol final representa el código de prefijo óptimo. La palabra clave de una letra es secuencia de etiquetas de borde en el camino desde la raíz hasta la letra.

Figura 16.5: Los pasos del algoritmo de Huffman para las frecuencias dadas en la Figura 16.3. Cada parte muestra el contenido de la cola ordenado en orden creciente por frecuencia. En cada paso, los dos árboles con las frecuencias más bajas se fusionan. Las hojas se muestran como rectángulos que contienen un carácter y su frecuencia. Los nodos internos se muestran como círculos que contienen la suma de los frecuencias de sus hijos. Un borde que conecta un nodo interno con sus hijos se etiqueta como 0 si es una ventaja para un hijo izquierdo y 1 si es una ventaja para un hijo derecho. La palabra clave de una letra es la secuencia de etiquetas en los bordes que conectan la raíz con la hoja de esa letra. *(a)* La inicial conjunto de $n = 6$ nodos, uno para cada letra. *(b)* - *(e)* Etapas intermedias. *(f)* El árbol final.

Línea 2 inicializa la cola de prioridad min- Q con los personajes C . El bucle **for** en las líneas 3 a 8 extrae repetidamente los dos nodos x y y de frecuencia más baja de la cola, y sustituye ellos en la cola con un nuevo nodo z *que* representa su fusión. La frecuencia de z es

calculado como la suma de las frecuencias de x y y en la línea 7. El nodo z tiene x como su hijo izquierdo y Y como su hijo derecho. (Este orden es arbitrario; cambiar el hijo izquierdo y derecho de cualquier nodo produce un código diferente del mismo costo). Después de n - 1 fusiones, el único nodo que queda en la cola: la raíz del árbol de código: se devuelve en la línea 9.

El análisis del tiempo de ejecución del algoritmo de Huffman asume que Q se implementa como un binary min-heap (consulte el Capítulo 6). Para un conjunto C de n caracteres, la inicialización de Q en la línea 2 se puede realizar en $O(n)$ tiempo usando el procedimiento BUILD-MIN-HEAP en la Sección 6.3 . los **El** bucle **for** en las líneas 3-8 se ejecuta exactamente n - 1 veces, y dado que cada operación de montón requiere tiempo $O(\lg n)$, el bucle contribuye con $O(n \lg n)$ al tiempo de ejecución. Por lo tanto, el tiempo de ejecución total de HUFFMAN en un conjunto de n caracteres es $O(n \lg n)$.

Corrección del algoritmo de Huffman

Para demostrar que el algoritmo codicioso HUFFMAN es correcto, mostramos que el problema de determinar un código de prefijo óptimo exhibe la elección codiciosa y la subestructura óptima propiedades. El siguiente lema muestra que se cumple la propiedad de elección codiciosa.

Lema 16.2

Sea C un alfabeto en el que cada carácter c C tiene una frecuencia $f[c]$. Vamos x e y haber dos los caracteres en C tienen las frecuencias más bajas. Entonces existe un código de prefijo óptimo para C en el que las palabras de código para x y y tienen la misma longitud y se diferencian sólo en el último bit.

Prueba La idea de la prueba es tomar el árbol T *que* representa un código de prefijo óptimo arbitrario y modificarlo para hacer un árbol que represente otro código de prefijo óptimo tal que los caracteres x e y aparecen como hermanos hojas de profundidad máxima en el nuevo árbol. Si podemos hacer esto, entonces sus palabras de código tendrán la misma longitud y solo se diferenciarán en el último bit.

Vamos a y b haber dos caracteres que se hermano hojas de profundidad máxima en T . Sin perdida de
En general, asumimos que $f[a] \le f[b]$ y $f[x] \le f[y]$. Dado que $f[x]$ y $f[y]$ son las dos hojas más bajas
frecuencias, en orden, y $f[a]$ y $f[b]$ son dos frecuencias arbitrarias, en orden, tenemos $f[x] \le f[a]$ y $f[y] \le f[b]$. Como se muestra en la Figura 16.6 , que el intercambio de las posiciones en T de *una* y x a producir un árbol T' , y luego que el intercambio de las posiciones en T' de b y Y para producir un árbol T'' . Por ecuación (16.5), la diferencia de costo entre T y T' es

Figura 16.6: Una ilustración del paso clave en la demostración del Lema 16.2. En el árbol óptimo T , deja *un* y b son dos de las hojas más profundos y son hermanos. Hojas x y y son los dos deja que el algoritmo de Huffman se fusiona primero; en que aparecen en posiciones arbitrarias en T .

Página 334

Hojas a y x se intercambian para obtener el árbol T' . A continuación, las hojas B y Y se intercambian para obtener el árbol T'' . Dado que cada intercambio no aumenta el costo, el árbol T'' resultante también es un árbol óptimo.

porque tanto $f[a] - f[x]$ como $d_T(a) - d_T(x)$ no son negativos. Más específicamente, $f[a] - f[x]$ es no negativo porque x es una hoja de frecuencia mínima, y $d_T(a) - d_T(x)$ no es negativo porque *una* es una hoja de máxima profundidad en T . De manera similar, intercambiar y y b no aumenta el costo, por lo que $B(T') - B(T'')$ no es negativo. Por lo tanto, $B(T'') \le B(T)$, y dado que T es óptimo, $B(T) \le B(T'')$, lo que implica $B(T'') = B(T)$. Por lo tanto, T'' es un árbol óptimo en el que x e y aparecen como hojas hermanas de máxima profundidad, de las que se deriva el lema.

El lema 16.2 implica que el proceso de construcción de un árbol óptimo mediante fusiones puede, sin pérdida de generalidad, comience con la codiciosa elección de fusionar esos dos personajes de frecuencia más baja. ¿Por qué es esta una elección codiciosa? Podemos ver el costo de una sola fusión como siendo la suma de las frecuencias de los dos elementos que se fusionan. El ejercicio 16.3-3 muestra que el costo total del árbol construido es la suma de los costos de sus fusiones. De todo lo posible fusiones en cada paso, HUFFMAN elige la que incurre en el menor costo.

El siguiente lema muestra que el problema de construir códigos de prefijo óptimos tiene el propiedad de la subestructura.

Lema 16.3

Deje C ser un alfabeto dado con frecuencia $f[c]$ definido para cada carácter $c \in C$. Vamos x e y sean dos caracteres en C con frecuencia mínima. Sea C' el alfabeto C con los caracteres x, y eliminado y (nuevo) carácter z agregado, de modo que $C' = C - \{x, y\} \cup \{z\}$; definir f para C' como para C , excepto que $f[z] = f[x] + f[y]$. Sea T' cualquier árbol que represente un código de prefijo óptimo para el alfabeto C' . Entonces el árbol T , obtenido de T' reemplazando el nodo hoja por z con un interno nodo que tiene x y y como los niños, representa un código óptimo prefijo para el alfabeto C .

Prueba Primero mostramos que el costo $B(T)$ del árbol T se puede expresar en términos del costo $B(T')$ de árbol T' considerando los costos de los componentes en la ecuación (16.5). Para cada $c \in C - \{x, y\}$, tienen $d_T(c) = d_{T'}(c)$, y por lo tanto $f[c] d_T(c) = f[c] d_{T'}(c)$. Dado que $d_T(x) = d_T(y) = d_{T'}(z) + 1$, tenemos

$$\begin{aligned} f[x] re_T(x) + f[y] re_T(y) &= (f[x] + f[y])(re_{T'}(z) + 1) \\ &= f[z] d_{T'}(z) + (f[x] + f[y]), \end{aligned}$$

de lo cual concluimos que

$$B(T) = B(T') + f[x] + f[y]$$

o equivalente,

$B(T') = B(T) - f[x] - f[y]$.

Ahora probamos el lema por contradicción. Suponga que T no representa un óptimo código de prefijo para C. Entonces existe un árbol T'' tal que $B(T'') < B(T)$. Sin perdida de

Página 335

generalidad (por Lemma 16.2), T "tiene x y y como hermanos. Sea T''' el árbol T'' con el padre común de x y y reemplazados por una hoja de z con la frecuencia $f[z] = f[x] + f[y]$. Luego

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T'), \end{aligned}$$

lo que contradice la suposición de que T' representa un código de prefijo óptimo para C'. Por lo tanto, T debe representar un código de prefijo óptimo para el alfabeto C.

Teorema 16.4

Procedimiento HUFFMAN produce un código de prefijo óptimo.

Prueba inmediata de los Lemas 16.2 y 16.3 .

Ejercicios 16.3-1

Demuestre que un árbol binario que no está lleno no puede corresponder a un código de prefijo óptimo.

Ejercicios 16.3-2

¿Cuál es un código Huffman óptimo para el siguiente conjunto de frecuencias, basado en las primeras 8 ¿Números de Fibonacci?

a: 1 b: 1 c: 2 d: 3 e: 5 f: 8 g: 13 h: 21

¿Puede generalizar su respuesta para encontrar el código óptimo cuando las frecuencias son las primeras n ¿Números de Fibonacci?

Ejercicios 16.3-3

Demuestre que el costo total de un árbol para un código también se puede calcular como la suma, sobre todos los nodos, de las frecuencias combinadas de los dos hijos del nodo.

Página 336

Ejercicios 16.3-4

Demuestre que si ordenamos los caracteres en un alfabeto de modo que sus frecuencias sean monótonamente decreciente, entonces existe un código óptimo cuyas longitudes de palabras de código son monótonamente creciente.

Ejercicios 16.3-5

Supongamos que tenemos un código de prefijo óptimo en un conjunto $C = \{0, 1, ..., n - 1\}$ de caracteres y desea transmitir este código utilizando la menor cantidad de bits posible. Muestre cómo representar cualquier óptimo código de prefijo en C usando solo $2n - 1 + n \lceil \lg n \rceil$ bits. (*Sugerencia:* use $2n - 1$ bits para especificar el estructura del árbol, como se descubre al caminar por el árbol).

Ejercicios 16.3-6

Generalizar el algoritmo de Huffman a palabras de código ternarias (es decir, palabras de código que utilizan los símbolos 0, 1, y 2) y demostrar que produce códigos ternarios óptimos.

Ejercicios 16.3-7

Suponga que un archivo de datos contiene una secuencia de caracteres de 8 bits tal que los 256 caracteres son casi igual de común: la frecuencia máxima de caracteres es menos del doble del mínimo frecuencia de caracteres. Demuestre que la codificación de Huffman en este caso no es más eficiente que usar un código ordinario de longitud fija de 8 bits.

Ejercicios 16.3-8

Demuestre que ningún esquema de compresión puede esperar comprimir un archivo de 8 bits elegido al azar. caracteres incluso por un solo bit. (*Sugerencia:* compare el número de archivos con el número de posibles archivos codificados.)

[2] Quizás "códigos sin prefijo" sería un mejor nombre, pero el término "códigos de prefijo" es estándar en la literatura.

Página 337

16.4 Fundamentos teóricos de los métodos codiciosos

Existe una hermosa teoría sobre los algoritmos codiciosos, que esbozamos en esta sección. Esta La teoría es útil para determinar cuándo el método codicioso produce soluciones óptimas. Involucra estructuras combinatorias conocidas como "matroides". Aunque esta teoría no cubre todos los casos para el que se aplica un método codicioso (por ejemplo, no cubre la selección de actividades problema de la Sección 16.1 o el problema de codificación de Huffman de la Sección 16.3), cubre muchos casos de interés práctico. Además, esta teoría se está desarrollando rápidamente y extendiéndose a cubrir muchas más aplicaciones; consulte las notas al final de este capítulo para obtener referencias.

Matroides

Un *matroide* es un par ordenado $M = (S, \ell)$ que satisface las siguientes condiciones.

1. S es un conjunto finito no vacío.
2. ℓ es una familia no vacía de subconjuntos de S , llamados subconjuntos **independientes** de S , tal que si $B \ell$ y AB , entonces $A \ell$. Decimos que es **hereditario** si satisface esta propiedad. Tenga en cuenta que el conjunto vacío Ø es necesariamente un miembro deℓ.
3. Si $A \ell$, $B \ell$ y $| A | <| B |$, entonces hay algún elemento $x B - A$ tal que $A \{ x \}$ ℓ. Decimos que M satisface la **propiedad de intercambio** .

La palabra "matroid" se debe a Hassler Whitney. Estaba estudiando **matroides matriciales** , en los que los elementos de S son las filas de una matriz dada y un conjunto de filas es independiente si son linealmente independientes en el sentido habitual. Es fácil demostrar que esta estructura define una matroide (vea el ejercicio 16.4-2).

Como otro ejemplo de matroides, considere el **gráfico matroide** $M_G = (S_G , \ell_G)$ definido en términos de un gráfico no dirigido dado $G = (V, E)$ como sigue.

- El conjunto S_G se define como E , el conjunto de bordes de G .
- Si A es un subconjunto de E , entonces $A \ell_G$ si y solo si A es acíclico. Es decir, un conjunto de aristas A es independiente si y solo si el subgrafo $G_A = (V, A)$ forma un bosque.

La matriz gráfica M_G está estrechamente relacionada con el problema del árbol de expansión mínimo, que es cubierto en detalle en el Capítulo 23 .

Teorema 16.5

Si $G = (V, E)$ es un gráfico no dirigido, entonces $M_G = (S_G , \ell_G)$ es un matroide.

Prueba Claramente, $S_G = E$ es un conjunto finito. Además, ℓ_G es hereditario, ya que un subconjunto de un bosque es un bosque. Dicho de otra manera, la eliminación de bordes de un conjunto acíclico de bordes no puede crear ciclos.

Por tanto, queda por demostrar que M_G satisface la propiedad de intercambio. Suponga que $G_A = (V, A)$ y $G_B = (V, B)$ son bosques de G y que $| B | >; | A |$. Es decir, A y B son conjuntos acíclicos de aristas, y B contiene más bordes que A hace.

Página 338

Del teorema B.2 se deduce que un bosque que tiene k bordes contiene exactamente $| V | - k$ árboles. (A demuestre esto de otra manera, comience con $| V |$ árboles, cada uno formado por un único vértice y sin aristas. Entonces, cada borde que se agrega al bosque reduce el número de árboles en uno).
G_A contiene $| V | - | A |$ árboles y bosque G_B contiene $| V | - | B |$ arboles

Dado que el bosque G_B tiene menos árboles que el bosque G_A , el bosque G_B debe contener algún árbol T cuyo vértices están en dos árboles diferentes en bosque G_A . Además, dado que T está conectado, debe contener un borde (u, v) de tal manera que los vértices u y v son en diferentes árboles en el bosque G_A . Desde el borde (u, v) conecta vértices en dos árboles diferentes en el bosque G_A , el borde (u, v) se puede agregar al bosque G_A sin crear un ciclo. Por tanto, M_G satisface la propiedad de intercambio, completando la prueba que M_G es un matroide.

Dado un matroide $M = (S , \ell)$, llamamos a un elemento $x \notin A$ una **extensión** de $A \ell$ si se puede sumar x a A conservando la independencia; es decir, x es una extensión de A si $A \{ x \} \ell$. Como un ejemplo, considere un gráfico matroid M_G . Si A es un conjunto independiente de aristas, entonces la arista e es una extensión de A si y solo si e no está en A y la adición de e a A no crea un ciclo.

Si A es un subconjunto independiente en un matroide M , decimos que A es **máximo** si no tiene extensiones. Es decir, *una* es máxima si no está contenida en cualquier subconjunto independiente más grande de M . los La siguiente propiedad suele ser útil.

Teorema 16.6

Todos los subconjuntos máximos independientes en una matroide tienen el mismo tamaño.

Prueba Supongamos lo contrario que A es un subconjunto independiente máximo de M y existe otro más grande máxima subconjunto independiente B de M . Entonces, la propiedad de intercambio implica que A es extensible a un conjunto independiente mayor $A \{ x \}$ para algún $x B - A$, contradiciendo la suposición de que A es máxima.

Como ilustración de este teorema, considere una matroide gráfica M_G para una conexión no dirigida gráfico G . Cada subconjunto independiente máximo de M_G debe ser un árbol libre con exactamente $| V | - 1$ bordes que conecta todos los vértices de G . Un árbol de este tipo se llama un árbol de expansión de G .

Decimos que un matroide $M = (S , \ell)$ se pondera si hay una función de ponderación asociada w que asigna un peso estrictamente positivo $w (x)$ para cada elemento $x S$. La función de peso w se extiende a subconjuntos de S por suma:

para cualquier AS . Por ejemplo, si dejamos que $w (e)$ denote la longitud de un borde e en una matriz gráfica M_G , entonces $w (A)$ es la longitud total de los bordes en conjunto borde A .

Página 339

Algoritmos codiciosos en una matroide ponderada

Muchos problemas para los que un enfoque codicioso proporciona soluciones óptimas se pueden formular en términos de encontrar un subconjunto independiente de peso máximo en un matroide ponderado. Es decir, somos dado un matroide ponderado $M = (S , \ell)$, y deseamos encontrar un conjunto independiente $A \ell$ tal que $w (A)$ se maximiza. Llamamos a un subconjunto que es independiente y tiene el máximo posible ponderar un subconjunto óptimo de la matroide. Porque el peso $w (x)$ de cualquier elemento $x S$ es positivo, un subconjunto óptimo es siempre un subconjunto independiente máximo; siempre ayuda a hacer *Lo* más grande posible.

Por ejemplo, en el problema del árbol de expansión mínimo , se nos da una conexión no dirigida gráfico $G = (V, E)$ y una función de longitud w tal que $w (e)$ es la longitud (positiva) de la arista e . (Nosotros Utilice el término "longitud" aquí para referirse a los pesos de los bordes originales del gráfico, reservando el término "peso" para referirse a los pesos en el matroide asociado). Se nos pide que busquemos un subconjunto de los bordes que conecta todos los vértices juntos y tiene una longitud total mínima. Para ver esto como un problema de encontrar un subconjunto óptimo de una matroide, considere la matroide ponderada M_G con función de peso w', donde $w' (e) = w_0 - w (e)$ y w_0 es mayor que la longitud máxima de cualquier borde. En este matroide ponderado, todos los pesos son positivos y un subconjunto óptimo es un árbol de expansión de longitud total mínima en el gráfico original. Más específicamente, cada máximo subconjunto independiente A corresponde a un árbol de expansión, y desde

$$w' (A) = (| V | - 1) w_0 - w (A)$$

para cualquier subconjunto independiente máximo A , un subconjunto independiente que maximiza la cantidad $w' (A)$ debe minimizar $w (A)$. Por tanto, cualquier algoritmo que pueda encontrar un subconjunto A óptimo en un matroid arbitrario puede resolver el problema del árbol de expansión mínimo.

El capítulo 23 proporciona algoritmos para el problema del árbol de expansión mínimo, pero aquí damos una algoritmo codicioso que funciona para cualquier matroide ponderado. El algoritmo toma como entrada un matroide ponderado $M = (S , \ell)$ con una función de peso positiva asociada w , y devuelve un óptima subconjunto A . En nuestro pseudocódigo, denotamos los componentes de M por $S [M]$ y $\ell [M]$ y la función de peso por w . El algoritmo es codicioso porque considera cada elemento $x S$ en gire con el fin de monotónicamente peso decreciente y de inmediato lo añade al conjunto A ser acumulado si $A \{ x \}$ es independiente.

```
Codicioso ( M, w )
1 A ← Ø
2 ordenar S [ M ] en orden decreciente monótona por peso w
3 para cada x S [ M ], tomado en orden decreciente monótona por peso w ( x )
4 hacer si A                    { x } ℓ [ M ]
5                       luego A ← A             { x }
6 devuelve A
```

Los elementos de S se consideran a su vez, en orden de peso decreciente monótonamente. Si el

elemento x siendo considerado se puede añadir a A mientras se mantiene *una* independencia 's, lo es. De lo contrario, se descarta x . Dado que el conjunto vacío es independiente por la definición de una matroide, y dado que x se suma a A solo si $A \cup \{ x \}$ es independiente, el subconjunto A es siempre independiente, por inducción. Por lo tanto, codicioso siempre devuelve un subconjunto independiente A . Veremos en un momento en que A es un subconjunto de peso máximo posible, de modo que A es un subconjunto óptimo.

Página 340

El tiempo de ejecución de GREEDY es fácil de analizar. Deje n denotan $| S |$. La fase de clasificación de Codicia toma tiempo $O (n \lg n)$. La línea 4 se ejecuta exactamente n veces, una para cada elemento de S . Cada ejecución de la línea 4 requiere una verificación de si el conjunto $A \cup \{ x \}$ es independiente o no . Si cada una de estas comprobaciones lleva el tiempo $O (f (n))$, todo el algoritmo se ejecuta en el tiempo $O (n \lg n + nf (n))$.

Ahora demostramos que GREEDY devuelve un subconjunto óptimo.

Lema 16.7: (Las matroides exhiben la propiedad de elección codiciosa)

Suponga que $M = (S , \ell)$ es una matroide ponderada con función de ponderación w y que S se clasifica en orden decreciente monótona por peso. Sea x el primer elemento de S tal que $\{ x \}$ es independiente, si existe tal x . Si x existe, entonces existe un subconjunto óptimo A de S que contiene x .

Prueba Si noexistetal x , entonces el único subconjunto independiente es el conjunto vacío y hemos terminado.

De lo contrario, sea B cualquier subconjunto óptimo no vacío. Suponga que $x \notin B$; de lo contrario, dejamos $A = B$ y terminamos.

Ningún elemento de B tiene un peso mayor que $w (x)$. Para ver esto, observe que $y \in B$ implica que $\{ y \}$ es independiente, ya que $B \in \ell$ yℓ es hereditario. Por tanto, nuestra elección de x asegura que $w (x) \geq w (y)$ para cualquier $y \in B$.

Construya el conjunto A *de la* siguiente manera. Empiece con $A = \{ x \}$. Por la elección de x, A es independiente. Usando la propiedad de intercambio, busque repetidamente un nuevo elemento de B que pueda agregarse a A hasta $| A | = | B |$ preservando al mismo tiempo la independencia de *una* . Entonces, $A = B - \{ y \} \cup \{ x \}$ para algún $y \in B$, y entonces

$$\begin{aligned} w (A) &= w (B) - w (y) + w (x) \\ &\geq w (B). \end{aligned}$$

Debido a que B es óptimo, A también debe ser óptimo, y debido a que $x \in A$, el lema está probado.

A continuación, mostramos que si un elemento no es una opción inicialmente, no puede ser una opción más adelante.

Lema 16.8

Sea $M = (S , \ell)$ cualquier matroide. Si x es un elemento de S que es una extensión de algún subconjunto A de S , entonces x es también una extensión de Ø.

Prueba Como x es una extensión de A , tenemos que $A \cup \{ x \}$ es independiente. Dado queℓ es hereditario, $\{ x \}$ debe ser independiente. Por tanto, x es una extensión de Ø.

Página 341

Corolario 16.9

Sea $M = (S , \ell)$ cualquier matroide. Si x es un elemento de S tal que x no es una extensión de Ø, entonces x no es una extensión de cualquier subconjunto independiente A de S .

Prueba Este corolario es simplemente el contrapositivo del Lema 16.8 .

El corolario 16.9 dice que cualquier elemento que no pueda usarse inmediatamente nunca podrá usarse. Por lo tanto, GREEDY no puede cometer un error pasando por encima de cualquier elemento inicial en S que sea no una extensión de Ø, ya que nunca se pueden utilizar.

Lema 16.10: Las matroides exhiben la propiedad de subestructura óptima

Sea x el primer elemento de S elegido por GREEDY para la matroide ponderada $M = (S , \ell)$. los El problema restante de encontrar un subconjunto independiente de peso máximo que contenga x *se* reduce a encontrar un subconjunto independiente de peso máximo del matroide ponderado $M' = (S', \ell)$, donde

$S' = \{ y S : \{ x, y \} \ell\}$,
$\ell' = \{ BS - \{ x \}: B \{ x \} \ell\}$, y

la función de ponderación de M' es la función de ponderación de M , restringida a S'. (Llamamos a M' el contracción de M por el elemento x .)

Prueba Si A es cualquier subconjunto independiente de peso máximo de M *que* contiene x , entonces $A' = A - \{ x \}$ es un subconjunto independiente de M'. Por el contrario, cualquier subconjunto independiente A' de M' produce un subconjunto independiente $A = A' \{ x \}$ de M . Como tenemos en ambos casos que $w (A) = w (A') + w (x)$, una solución de peso máximo en M *que* contiene x produce una solución de peso máximo en M', y viceversa.

Teorema 16.11: (Corrección del algoritmo codicioso en matroides

Si $M = (S , \ell)$ es un matroide ponderado con la función de peso w , entonces GREEDY (M, w) devuelve un subconjunto óptimo.

Prueba Por Corolario 16.9 , cualquier elemento que se pasan sobre un principio porque no están Las extensiones de Ø pueden olvidarse, ya que nunca pueden ser útiles. Una vez que el primer elemento x se selecciona, Lemma 16.7 implica que GREEDY no err hace mediante la adición de x a A , puesto que hay existe un subconjunto óptimo que contiene x . Finalmente, el Lema 16.10 implica que el resto El problema es encontrar un subconjunto óptimo en el matroide M' que es la contracción de M por x . Después de que el procedimiento GREEDY establece A en $\{ x \}$, todos los pasos restantes se pueden interpretar como actuando en el matroide $M' = (S', \ell')$, porque B es independiente en M' si y solo si $B \{ x \}$ es independiente en M , para todos los conjuntos $B \ell'$. Así, la operación subsiguiente de GREEDY encontrará un

Página 342

subconjunto independiente de peso máximo para M', y la operación general de GREEDY encontrará un subconjunto independiente máximo peso para M .

Ejercicios 16.4-1

Demuestre que (S , ℓ_k) es una matroide, donde S es cualquier conjunto finito y ℓ_k es el conjunto de todos los subconjuntos de S de tamaño como máximo k , donde $k \leq | S |$.

Ejercicios 16.4-2:

Dada una matriz T de $m \times n$ sobre algún campo (como los reales), demuestre que (S, ℓ) es una matroide, donde S es el conjunto de columnas de T y $A \in \ell$ si y solo si las columnas de A son linealmente independiente.

Ejercicios 16.4-3:

Demuestre que si (S, ℓ) es una matroide, entonces (S, ℓ') es una matroide, donde

$\ell' = \{A' : S - A' \text{ contiene algún } A \in \ell\}$ máximo .

Es decir, los conjuntos independientes máximos de (S, ℓ') son solo los complementos del máximo conjuntos independientes de (S, ℓ).

Ejercicios 16.4-4:

Sea S un conjunto finito y sea $S_1, S_2, ..., S_k$ una partición de S en subconjuntos disjuntos no vacíos. Defina la estructura (S, ℓ) con la condición de que $\ell = \{A : |A \cap S_i| \leq 1 \text{ para } i = 1, 2, ..., k\}$. mostrar que (S, ℓ) es una matroide. Es decir, el conjunto de todos los conjuntos A que contienen como máximo un miembro en cada El bloque de la partición determina los conjuntos independientes de una matroide.

Ejercicios 16.4-5

Muestre cómo transformar la función de peso de un problema de matroide ponderado, donde el La solución óptima es un subconjunto independiente máximo de *peso mínimo* , para convertirlo en un estándar problema de la matroide ponderada. Argumenta cuidadosamente que tu transformación es correcta.

Página 343

16.5 Un problema de programación de tareas

Un problema interesante que puede resolverse utilizando matroids es el problema de optimizar programar tareas de tiempo unitario en un solo procesador, donde cada tarea tiene una fecha límite, junto con una multa que debe pagarse si se incumple el plazo. El problema parece complicado, pero se puede resolver de una manera sorprendentemente simple utilizando un codicioso algoritmo.

Una **tarea de tiempo unitario** es un trabajo, como un programa que se ejecutará en una computadora, que requiere exactamente una unidad de tiempo para completar. Dado un conjunto finito S de tareas de tiempo unitario, un **programa** para S es una permutación de S *que* especifica el orden en el que estas tareas deben ser realizado. La primera tarea del programa comienza en el momento 0 y finaliza en el momento 1, el la segunda tarea comienza en el momento 1 y termina en el momento 2, y así sucesivamente.

El problema de **programar tareas de tiempo unitario con fechas límite y sanciones para un Un solo procesador** tiene las siguientes entradas:

- un conjunto $S = \{a_1, a_2, ..., a_n\}$ de n tareas de tiempo unitario;
- un conjunto de n **plazos** enteros $d_1, d_2, ..., d_n$, tal que cada d_i satisface $1 \leq d_i \leq n$ y se supone que la tarea a_i termina en el tiempo d_i ; y
- un conjunto de n ponderaciones o **penalizaciones** no negativas $w_1, w_2, ..., w_n$, de manera que incurrimos en penalización de w_i si la tarea a_i no se termina en el tiempo d_i y no incurrimos en ninguna penalización si una tarea termina por su fecha límite.

Se nos pide que encontremos un programa para S que minimice la multa total incurrida por fechas límite incumplidas.

Considere un horario determinado. Decimos que una tarea está **atrasada** en este horario si termina después su fecha límite. De lo contrario, la tarea está al **principio** de la programación. Un horario arbitrario puede siempre se pondrá en forma **temprana-primera** , en la que las primeras tareas preceden a las tardías. A vea esto, tenga en cuenta que si alguna tarea temprana a_i sigue a una tarea tardía a_j , entonces podemos cambiar el las posiciones de a_i y a_j , y a_i todavía serán tempranas y a_j todavía llegarán tarde.

De manera similar, afirmamos que un horario arbitrario siempre se puede poner en **forma canónica** , en que las primeras tareas preceden a las tardías y las primeras se programan en orden de plazos que aumentan monótonamente. Para hacerlo, ponemos el horario en primer lugar

formar. Entonces, siempre que haya dos tareas iniciales a_i y a_j que terminen en los momentos respectivos k

y $k + 1$ en el programa tal que $d_j < d_i$, intercambiamos las posiciones de a_i y a_j. Desde una a_j

es temprano antes del intercambio, $k + 1 \leq d_j$. Por lo tanto, $k + 1 < d_i$, por lo que a_i es todavía temprano después de el intercambio. La tarea a_j se mueve más temprano en la programación, por lo que también es temprano después del intercambio.
La búsqueda de un horario óptimo se reduce así a encontrar un conjunto A de tareas que deben
Sea temprano en el horario óptimo. Una vez que *una* se determina, podemos crear el actual
programar enumerando los elementos de A en orden de fecha límite creciente monótona, luego
enumerar las tareas tardías (es decir, $S - A$) en cualquier orden, produciendo un orden canónico de las
horario óptimo.
Decimos que un conjunto A de tareas es independiente si existe un cronograma para estas tareas.
de modo que ninguna tarea llegue tarde. Claramente, el conjunto de tareas tempranas para un horario forma un

conjunto independiente de tareas. Sea ℓ el conjunto de todos los conjuntos de tareas independientes.
Considere el problema de determinar si un conjunto A de tareas dado es independiente. por
$t = 0, 1, 2, ..., n$, sea $N_t(A)$ el número de tareas en A cuya fecha límite es t o anterior.
Obsérvese que $N_0(A) = 0$ para cualquier conjunto A.

Lema 16.12

Para cualquier conjunto de tareas A, las siguientes declaraciones son equivalentes.

1. El conjunto A es independiente.
2. Para $t = 0, 1, 2, ..., n$, tenemos $N_t(A) \leq t$.
3. Si las tareas en A están programadas en orden de aumento monotónico plazos, entonces ninguna tarea llega tarde.

Prueba claramente, si $N_t(A) > t$ para algunos t, entonces no hay forma de hacer un horario sin tareas tardías para el conjunto A, porque hay más de t tareas para terminar antes del tiempo t. Por lo tanto, (1) implica (2). Si (2) se mantiene, entonces (3) debe seguir: no hay forma de "quedarse atascado" cuando

Página 344

programar las tareas en orden de plazos crecientes monótonamente, ya que (2) implica que el i^a más grande fecha límite es a lo sumo i. Finalmente, (3) implica trivialmente (1).

Usando la propiedad 2 del Lema 16.12, podemos calcular fácilmente si un conjunto dado de tareas es independiente (vea el ejercicio 16.5-2).
El problema de minimizar la suma de las penalizaciones de las tareas tardías es el mismo que problema de maximizar la suma de las penalizaciones de las primeras tareas. El seguimiento teorema asegura que podemos usar el algoritmo codicioso para encontrar un conjunto independiente A de tareas con la máxima penalización total.

Teorema 16.13

Si S es un conjunto de tareas de tiempo unitario con fechas límite, y ℓ es el conjunto de todos los conjuntos independientes de

tareas, entonces el sistema correspondiente (S, ℓ) es una matroide.

Prueba Cada subconjunto de un conjunto independiente de tareas es ciertamente independiente. Para probar el

propiedad de intercambio, suponga que B y A son conjuntos independientes de tareas y que $|B| > |A|$. Sea k el mayor t tal que $N_t(B) \leq N_t(A)$. (Tal valor de t existe, ya que $N_0(A) = N_0(B) = 0$.) Dado que $N_n(B) = |B|$ y $N_n(A) = |A|$, pero $|B| > |A|$, debemos tener que $k < n$ y

que $N_j(B) > N_j(A)$ para todo j en el rango $k + 1 \leq j \leq n$. Por tanto, B contiene más tareas

con fecha límite $k + 1$ que A hace. Sea a_i una tarea en $B - A$ con fecha límite $k + 1$. Sea $A' = A \cup \{a_i\}$.

Ahora mostramos que A' debe ser independiente usando la propiedad 2 del Lema 16.12. Para $0 \leq t \leq k$, tenemos $N_t(A') = N_t(A) \leq t$, ya que A es independiente. Para $k < t = n$, tenemos $N_t(A') \leq N_t(B) \leq t$, ya que B es independiente. Por tanto, A' es independiente, completando nuestro

prueba de que (S, ℓ) es una matroide.

Según el teorema 16.11, podemos usar un algoritmo codicioso para encontrar un peso máximo conjunto independiente de tareas A. Luego podemos crear un horario óptimo con las tareas en A como sus primeras tareas. Este método es un algoritmo eficiente para programar tareas de tiempo unitario. con plazos y sanciones para un solo procesador. El tiempo de ejecución es $O(n^2)$ usando Codicioso, ya que cada uno de los controles de independencia $O(n)$ realizados por ese algoritmo toma tiempo $O(n)$ (vea el ejercicio 16.5-2). En el problema 16-4 se ofrece una implementación más rápida. La figura 16.7 da un ejemplo de un problema de programación de tareas de tiempo unitario con fechas límite. y sanciones para un solo procesador. En este ejemplo, el algoritmo codicioso selecciona tareas

un_1 , un_2 , un_3 y un_4 , luego rechaza un_5 y un_6 , y finalmente acepta un_7 . El óptimo final el horario es

un_2 , un_4 , un_1 , un_3 , un_7 , un_5 , un_6

que tiene una penalización total incurrida de $w_5 + w_6 = 50$.

Tarea a_{yo}	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_{yo}	70	60	50	40	30	20	10

Figura 16.7: Un ejemplo del problema de programar tareas de tiempo unitario con fechas límite y sanciones para un solo procesador.

Ejercicios 16.5-1

Resuelva la instancia del problema de programación que se muestra en la Figura 16.7 , pero con cada penalización w_i reemplazado por 80 - w_i .

Ejercicios 16.5-2

Muestre cómo usar la propiedad 2 del lema 16.12 para determinar en el tiempo O (| A |) si un dado el conjunto A de tareas es independiente.

Problemas 16-1: Cambio de monedas

Considere el problema de hacer cambio por n centavos usando la menor cantidad de monedas. Suponga que el valor de cada moneda es un número entero.

a. Describe un algoritmo codicioso para realizar cambios que consta de trimestres, monedas de diez, cinco y centavos. Demuestre que su algoritmo produce un solucion optima.

si. Suponga que las monedas disponibles están en las denominaciones que son potencias de c , es decir, las denominaciones son c_0 , c_1 , ..., c_k para algunos enteros c >; 1 y $k \geq 1$. Demuestre que el algoritmo codicioso siempre produce un solucion optima.

C. Dar un conjunto de denominaciones de monedas para las que el algoritmo codicioso hace no rinde una solución óptima. Su juego debe incluir un centavo para que hay una solución para cada valor de n .

re. Dar un algoritmo de tiempo O (nk) que haga cambios para cualquier conjunto de k diferentes denominaciones de moneda, asumiendo que una de las monedas es un centavo.

Problemas 16-2: Programación para minimizar el tiempo promedio de finalización

Suponga que se le da un conjunto $S = \{ a_1 , a_2 , ..., a_n \}$ de tareas, donde la tarea a_i requiere p_i unidades de tiempo de procesamiento para completar, una vez que ha comenzado. Tienes una computadora en la que ejecutar estas tareas, y la computadora solo puede ejecutar una tarea a la vez. Deja que c_{yo} sea la finalización tiempo de la tarea a_i , es decir, el momento en el que la tarea a_i completa el procesamiento. Tu meta es minimizar el tiempo medio de finalización, es decir, minimizar . Por ejemplo, suponga que hay dos tareas, un_1 y un_2 , con $p_1 = 3$ $yp_2 = 5$, y considere el horario

Página 346

en el que un_2 corre primero, seguido de un_1. Entonces $c_2 = 5$, $c_1 = 8$, y el tiempo promedio de finalización es (5 + 8) / 2 = 6.5.

a. Dar un algoritmo que programe las tareas para minimizar la tiempo medio de finalización. Cada tarea debe ejecutarse de forma no preventiva, que es decir, una vez que se inicia la tarea a_i, debe ejecutarse continuamente durante p_i unidades de tiempo. Demuestre que su algoritmo minimiza el tiempo medio de finalización, e indique el tiempo de ejecución de su algoritmo.

si. Supongamos ahora que no todas las tareas están disponibles a la vez. Es decir, cada tarea tiene un tiempo de liberación r_i antes del cual no está disponible para ser procesada. Supongamos también que permitimos la preferencia, de modo que una tarea puede suspenderse y reiniciarse más tarde. Por ejemplo, una tarea a_i con tiempo de procesamiento $p_i = 6$ puede comenzar a ejecutarse en el tiempo 1 y ser se adelanta en el momento 4. Luego se puede reanudar en el momento 10, pero se puede adelantar en el momento 11 y finalmente reanudar en el momento 13 y completar en el momento 15. La tarea $a_{i\,se}$ ha ejecutado por un total de 6 unidades de tiempo, pero su tiempo de ejecución ha dividido en tres partes. Decimos que el tiempo de finalización de una_i es 15. Proporcione un algoritmo que programe las tareas para minimizar el tiempo medio de finalización en este nuevo escenario. Demuestra que tu El algoritmo minimiza el tiempo medio de finalización y establece el tiempo de ejecución de su algoritmo.

Problemas 16-3: Subgrafos acíclicos

a. Sea $G = (V, E)$ una gráfica no dirigida. Usando la definición de un matroide, demuestre que (E, ℓ) es una matroide, donde $A \in \ell$ si y solo si A es un subgrupo acíclico de E.

si. La matriz de incidencia para un gráfico no dirigido $G = (V, E)$ es a $|V| \times |E|$ matriz M tal que $M_{ve} = 1$ si la arista e es incidente en el vértice v, y $M_{ve} = 0$ en caso contrario. Argumenta que un conjunto de columnas de M es linealmente independiente sobre el campo de los enteros módulo 2 si y sólo si el el conjunto de aristas correspondiente es acíclico. Luego, usa el resultado de Ejercicio 16.4-2 para proporcionar una prueba alternativa de que (E, ℓ) del inciso a) es matroid.

C. Suponga que se asocia un peso no negativo $w(e)$ con cada borde en un gráfico no dirigido $G = (V, E)$. Dar un algoritmo eficiente a encuentre un subconjunto acíclico de E de peso total máximo.

re. Sea $G(V, E)$ un grafo dirigido arbitrario, y dejemos que (E, ℓ) se defina así que $A \in \ell$ si y solo si A no contiene ningún ciclo dirigido. Dar un ejemplo de un grafo dirigido G tal que el sistema asociado (E, ℓ) no es un matroide. Especificar qué condición definitoria para una matroide no aguanta.

mi. La matriz de incidencia para un gráfico dirigido $G = (V, E)$ es a $|V| \times |E|$ matriz M tal que $M_{ve} = -1$ si la arista e sale del vértice v, $M_{ve} = 1$ si la arista e entra en el vértice v, y $M_{ve} = 0$ en caso contrario. Argumente que si un conjunto de columnas de M es linealmente independiente, entonces el conjunto correspondiente de bordes no contiene un ciclo dirigido.

F. El ejercicio 16.4-2 nos dice que el conjunto de conjuntos linealmente independientes de las columnas de cualquier matriz M forman una matroide. Explique cuidadosamente por qué los resultados de las partes (d) y (e) no son contradictorios. ¿Cómo puede fallar ser una correspondencia perfecta entre la noción de un conjunto de aristas ser acíclico y la noción del conjunto asociado de columnas de la la matriz de incidencia es linealmente independiente?

Página 347

Problemas 16-4: variaciones de programación

Considere el siguiente algoritmo para el problema de la Sección 16.5 de programación de unidades de tiempo tareas con plazos y sanciones. Deje que todos los n intervalos de tiempo estén inicialmente vacíos, donde el intervalo de tiempo i es el intervalo de tiempo de longitud unitaria que finaliza en el tiempo i. Consideramos las tareas en orden de pena decreciente monótona. Al considerar la tarea a_j, si existe un intervalo de tiempo en o antes de un_j's plazo d_j que todavía está vacío, asignar un_j a la última tal ranura, llenándolo. Si no existe tal ranura, asigne la tarea a_j a la última de las ranuras aún sin llenar.

a. Argumenta que este algoritmo siempre da una respuesta óptima.
si. Utilice el bosque de conjuntos disjuntos rápidos presentado en la Sección 21.3 para implementar el algoritmo de manera eficiente. Suponga que el conjunto de tareas de entrada tiene ya se han clasificado en orden decreciente monótona por penalización. Analice el tiempo de ejecución de su implementación.

Notas del capítulo

Se puede encontrar mucho más material sobre algoritmos codiciosos y matroides en Lawler [196]. y Papadimitriou y Steiglitz [237] .
El algoritmo codicioso apareció por primera vez en la literatura de optimización combinatoria en un 1971 artículo de Edmonds [85] , aunque la teoría de las matroides se remonta a 1935 artículo de Whitney [314] .
Nuestra prueba de la exactitud del algoritmo codicioso para el problema de selección de actividad es basado en el de Gavril [112] . El problema de la programación de tareas se estudia en Lawler [196] , Horowitz y Sahni [157] y Brassard y Bratley [47] .
Los códigos de Huffman se inventaron en 1952 [162]; Lelewer y Hirschberg [200] encuestas técnicas de compresión de datos conocidas desde 1987.

Una extensión de la teoría matroide a la teoría greedoide fue promovida por Korte y Lovász [189 , 190 , 191 , 192], quienes generalizan enormemente la teoría aquí presentada.

Capítulo 17: Análisis amortizado

Visión general

En un análisis amortizado , el tiempo necesario para realizar una secuencia de operaciones de estructura de datos se promedia sobre todas las operaciones realizadas. El análisis amortizado se puede utilizar para demostrar que el costo promedio de una operación es pequeño, si se promedia sobre una secuencia de operaciones, incluso aunque una sola operación dentro de la secuencia puede resultar costosa. El análisis amortizado difiere del análisis de casos promedio en que la probabilidad no está involucrada; un análisis amortizado garantiza el *rendimiento medio de cada operación en el peor de los casos* .

Las tres primeras secciones de este capítulo cubren las tres técnicas más comunes utilizadas en análisis amortizado. La sección 17.1 comienza con un análisis agregado, en el que determinamos un límite superior $T(n)$ sobre el costo total de una secuencia de n operaciones. El costo promedio por la operación es entonces $T(n) / n$. Tomamos el costo promedio como el costo amortizado de cada operación, por lo que que todas las operaciones tienen el mismo costo amortizado.

La sección 17.2 cubre el método contable, en el que determinamos un costo amortizado de cada operación. Cuando hay más de un tipo de operación, cada tipo de operación puede tener un Diferente costo amortizado. El método contable sobrecarga algunas operaciones al principio del secuencia, almacenando el recargo como "crédito prepago" en objetos específicos en la estructura de datos. El crédito se utiliza más adelante en la secuencia para pagar las operaciones que se cobran menos de lo que se realmente cuesta.

Página 348

La sección 17.3 analiza el método potencial, que es como el método contable en el sentido de que determinar el costo amortizado de cada operación y puede sobrecargar las operaciones desde el principio compensar los recargos más tarde. El método potencial mantiene el crédito como el "potencial energía "de la estructura de datos como un todo en lugar de asociar el crédito con el individuo objetos dentro de la estructura de datos.

Usaremos dos ejemplos para examinar estos tres métodos. Uno es una pila con el adicional operación MULTIPOP, que hace estallar varios objetos a la vez. El otro es un contador binario que cuenta desde 0 mediante la operación única INCREMENT.

Al leer este capítulo, tenga en cuenta que los cargos asignados durante una amortización Los análisis son solo para fines de análisis. No necesitan ni deben aparecer en el código. Si, por ejemplo, se asigna un crédito a un objeto x cuando se usa el método contable, no hay es necesario asignar una cantidad adecuada a algún atributo de *crédito* [x] en el código.

La información sobre una estructura de datos particular obtenida al realizar un análisis amortizado puede ayudar a optimizar el diseño. En la Sección 17.4 , por ejemplo, usaremos el método potencial analizar una mesa que se expande y contrae dinámicamente.

17.1 Análisis agregado

En el análisis agregado , mostramos que para todo n , una secuencia de n operaciones toma el *peor de los casos* tiempo $T(n)$ en total. En el peor de los casos, el costo promedio, o costo amortizado , por operación es

por lo tanto $T(n)/n$. Tenga en cuenta que este costo amortizado se aplica a cada operación, incluso cuando hay varios tipos de operaciones en la secuencia. Los otros dos métodos que estudiaremos en este capítulo, el método contable y el método potencial, pueden asignar diferentes amortizaciones costos a diferentes tipos de operaciones.

Operaciones de pila

En nuestro primer ejemplo de análisis agregado, analizamos las pilas que se han aumentado con un nueva operación. La sección 10.1 presentó las dos operaciones fundamentales de pila, cada una de las cuales toma $O(1)$ tiempo:

PUSH (S , x) empuja objeto x en la pila S .

POP (S) abre la parte superior de la pila S y devuelve el objeto reventado.

Dado que cada una de estas operaciones se ejecuta en $O(1)$ tiempo, consideremos que el costo de cada una es 1. El El costo total de una secuencia de n operaciones PUSH y POP es por lo tanto n , y la ejecución real el tiempo para n operaciones es por tanto $\Theta(n)$.

Ahora agregamos la operación de pila MULTIPOP (S , k), que elimina los k objetos superiores de la pila S , o saca toda la pila si contiene menos de k objetos. En el siguiente pseudocódigo, el La operación STACK-EMPTY devuelve TRUE si no hay objetos actualmente en la pila, y FALSO de lo contrario.

```
MULTIPOP ( S , k )
1 mientras no STACK-EMPTY ( S ) y k ≠ 0
2 hacer POP ( S )
```

Página 349

```
3                k → k - 1
```

La figura 17.1 muestra un ejemplo de MULTIPOP.

Figura 17.1: La acción de MULTIPOP en una pila S , mostrada inicialmente en *(a)* . Los 4 mejores objetos aparecen en MULTIPOP (S , 4), cuyo resultado se muestra en *(b)* . La siguiente operación es MULTIPOP (S , 7), que vacía la pila, que se muestra en *(c),* ya que había menos de 7 Objetos restantes.

¿Cuál es el tiempo de ejecución de MULTIPOP (S , k) en una pila de objetos s ? El tiempo de ejecución real es lineal en el número de operaciones POP realmente ejecutadas y, por tanto, basta con analizar MULTIPOP en términos de los costos abstractos de 1 cada uno para PUSH y POP. El número de iteraciones del **mientras** bucle es el número MIN (s , k) de los objetos hechos estallar de la pila. Para cada iteración del bucle, se realiza una llamada a POP en la línea 2. Por lo tanto, el costo total de MULTIPOP es min (s , k), y el tiempo de ejecución real es una función lineal de este costo.

Analicemos una secuencia de n operaciones PUSH, POP y MULTIPOP en un apilar. El costo del peor caso de una operación MULTIPOP en la secuencia es $O(n)$, ya que la pila el tamaño es como máximo n . Por tanto, el tiempo del peor caso de cualquier operación de pila es $O(n)$, y por tanto un secuencia de n costos de operaciones $O(n^2)$, ya que podemos tener $O(n)$ costos de operaciones MULTIPOP $O(n)$ cada uno. Si bien este análisis es correcto, el resultado $O(n^2)$, obtenido al considerar el En el peor de los casos, el costo de cada operación individualmente no es ajustado.

Usando el análisis agregado, podemos obtener un mejor límite superior que considere la totalidad secuencia de n operaciones. De hecho, aunque una sola operación MULTIPOP puede resultar costosa, cualquier secuencia de n operaciones PUSH, POP y MULTIPOP en una pila inicialmente vacía puede costo como máximo $O(n)$. ¿Por qué? Cada objeto se puede hacer estallar como máximo una vez por cada vez que se empuja. Por lo tanto, la cantidad de veces que se puede llamar a POP en una pila no vacía, incluidas las llamadas dentro de MULTIPOP, es como máximo el número de operaciones PUSH, que es como máximo n . Para cualquier valor de n , cualquier secuencia de n operaciones PUSH, POP y MULTIPOP toma un total de $O(n)$ hora. El costo promedio de una operación es $O(n)/n = O(1)$. En el análisis agregado, asignamos la el costo amortizado de cada operación será el costo promedio. En este ejemplo, por lo tanto, los tres

las operaciones de pila tienen un costo amortizado de O (1).

Recalcamos nuevamente que aunque acabamos de mostrar que el costo promedio, y por lo tanto tiempo de ejecución, de una operación de pila es O (1), no se involucró ningún razonamiento probabilístico. Nosotros en realidad mostró un límite de O (n) en el *peor de los casos* en una secuencia de n operaciones. Dividiendo este total el costo por n arrojó el costo promedio por operación, o el costo amortizado.

Incrementar un contador binario

Como otro ejemplo de análisis agregado, considere el problema de implementar un k -bit contador binario que cuenta hacia arriba desde 0. Usamos una matriz A [0 k -1] de bits, donde *longitud* [A] = k , como contador. Un número binario x que se almacena en el contador tiene su

Página 350

bit de orden en A [0] y su bit de orden más alto en A [k - 1], de modo que . Inicialmente, x = 0 y así A [i] = 0 para i = 0, 1, ..., k - 1. Para sumar 1 (módulo 2^k) al valor en el contador, usamos el siguiente procedimiento.

```
INCREMENTO (A)
1 i ← 0
2 mientras que i < longitud [ A ] y A [ i ] = 1
3 hacer A [ i ] ← 0
4                   yo ← yo + 1
5 si i < longitud [ A ]
6 luego A [ i ] ← 1
```

La figura 17.2 muestra lo que le sucede a un contador binario cuando se incrementa 16 veces, comenzando con el valor inicial 0 y terminando con el valor 16. Al comienzo de cada iteración del **while** bucle en las líneas 2-4, deseamos agregar un 1 en la posición i . Si A [i] = 1, agregar 1 cambia el bit a 0 en la posición i y produce un acarreo de 1, que se agregará a la posición i + 1 en la siguiente iteración del lazo. De lo contrario, el ciclo termina, y luego, si $i < k$, sabemos que A [i] = 0, de modo que sumando un 1 en la posición i , cambiando el 0 a 1, se trata en la línea 6. El costo de cada INCREMENTO la operación es lineal en el número de bits invertidos.

Figura 17.2: Contador binario de 8 bits cuando su valor va de 0 a 16 en una secuencia de 16 Operaciones de INCREMENTO. Los bits que se voltean para alcanzar el siguiente valor están sombreados. El costo corriente para voltear bits se muestra a la derecha. Observe que el costo total nunca es más del doble del número total de operaciones INCREMENT.

Como en el ejemplo de la pila, un análisis superficial produce un límite que es correcto pero no ajustado. UNA la ejecución única de INCREMENT lleva tiempo Θ (k) en el peor de los casos, en el que la matriz A contiene todos. Por lo tanto, una secuencia de n operaciones INCREMENT en un contador inicialmente cero lleva tiempo O (nk) en el peor de los casos.

Página 351

Podemos ajustar nuestro análisis para obtener un costo del peor caso de $O(n)$ para una secuencia de n INCREMENT, observando que no todos los bits cambian cada vez que se llama INCREMENT. Como La figura 17.2 muestra que $A[0]$ cambia cada vez que se llama INCREMENT. El siguiente orden más alto bit, $A[1]$, cambia sólo cada dos veces: una secuencia de n operaciones INCREMENT en un

El contador cero hace que $A[1]$ gire $\lceil n/2 \rceil$ veces. De manera similar, el bit $A[2]$ se voltea solo cada cuarta vez, o $\lceil n/4 \rceil$ veces en una secuencia de n INCREMENTOS. En general, para $i = 0, 1, ..., \lceil \lg n \rceil$, bit $A[i]$ voltea $\lceil n/2^i \rceil$ veces en una secuencia de n operaciones de INCREMENTO en un contador inicialmente cero. Para $i > \lceil \lg n \rceil$, el bit $A[i]$ nunca cambia en absoluto. Por tanto, el número total de giros en la secuencia es

por la ecuación (A.6) . El tiempo del peor de los casos para una secuencia de n operaciones INCREMENT en un inicialmente el contador cero es por tanto $O(n)$. El costo promedio de cada operación, y por lo tanto el costo amortizado por operación, es $O(n)/n = O(1)$.

Ejercicios 17.1-1

Si el conjunto de operaciones de pila incluye una operación MULTIPUSH, que empuja k elementos a la pila, ¿se mantendría el límite $O(1)$ del costo amortizado de las operaciones de pila?

Ejercicios 17.1-2

Muestre que si se incluyera una operación DECREMENTO en el ejemplo del contador de k bits, n las operaciones pueden costar hasta $\Theta(nk)$ tiempo.

Ejercicios 17.1-3

Se realiza una secuencia de n operaciones sobre una estructura de datos. La i- ésima operación cuesta i si i es un potencia exacta de 2, y 1 en caso contrario. Utilice un análisis agregado para determinar el costo amortizado por operación.

17.2 El método contable

En el *método contable* de análisis amortizado, asignamos diferentes cargos a diferentes operaciones, con algunas operaciones cobradas más o menos de lo que realmente cuestan. La cantidad que Cargar una operación se denomina *costo amortizado* . Cuando el costo amortizado de una operación excede

Página 352

su costo real, la diferencia se asigna a objetos específicos en la estructura de datos como *crédito* . El crédito se puede usar más adelante para ayudar a pagar operaciones cuyo costo amortizado sea menor que su costo real. Por lo tanto, se puede ver el costo amortizado de una operación dividido entre sus costo real y crédito que se deposita o se agota. Este método es muy diferente al análisis agregado, en el que todas las operaciones tienen el mismo costo amortizado.

Hay que elegir cuidadosamente los costos amortizados de las operaciones. Si queremos análisis con

costos amortizados para mostrar que en el peor de los casos el costo promedio por operación es pequeño, el total
El costo amortizado de una secuencia de operaciones debe ser un límite superior en el costo real total de
la secuencia. Además, como en el análisis agregado, esta relación debe ser válida para todas las secuencias.
de operaciones. Si denotamos el costo real de la i- ésima operación por c_i y el costo amortizado de
la i ésima operación por, requerimos

(17,1)

para todas las secuencias de n operaciones. El crédito total almacenado en la estructura de datos es la diferencia
entre el costo total amortizado y el costo real total, o . Por desigualdad
(17.1), el crédito total asociado con la estructura de datos debe ser no negativo en todo momento. Si
se permitió que el crédito total llegara a ser negativo (como resultado de la
operaciones con la promesa de reembolsar la cuenta más adelante), luego los costos totales amortizados
incurrido en ese momento estaría por debajo de los costos reales totales incurridos; para la secuencia de
operaciones hasta ese momento, el costo total amortizado no sería un límite superior en el total
costo real. Por lo tanto, debemos tener cuidado de que el crédito total en la estructura de datos nunca se convierta en
negativo.

Operaciones de pila

Para ilustrar el método contable del análisis amortizado, volvamos al ejemplo de la pila.
Recuerde que los costos reales de las operaciones fueron

EMPUJAR 1,
POPULAR 1,
MULTIPOP min (k , s),

donde k es el argumento proporcionado a MULTIPOP y s es el tamaño de la pila cuando se llama. Dejar
asignamos los siguientes costes amortizados:

EMPUJAR 2,
POPULAR 0,
MULTIPOP 0.

Tenga en cuenta que el costo amortizado de MULTIPOP es una constante (0), mientras que el costo real es
variable. Aquí, los tres costos amortizados son O (1), aunque en general los costos amortizados de
las operaciones consideradas pueden diferir asintóticamente.

Ahora demostraremos que podemos pagar por cualquier secuencia de operaciones de pila cobrando el
costos amortizados. Suponga que usamos un billete de un dólar para representar cada unidad de costo. Empezamos con un

Página 353

pila vacía. Recuerde la analogía de la Sección 10.1 entre la estructura de datos de la pila y una pila
de platos en una cafetería. Cuando empujamos un plato en la pila, usamos 1 dólar para pagar el
costo del empuje y nos queda un crédito de 1 dólar (de los 2 dólares cobrados), que
poner encima del plato. En cualquier momento, cada plato de la pila tiene un dólar de crédito en
eso.

El dólar almacenado en el plato es un pago anticipado por el costo de sacarlo de la pila. Cuando
ejecutamos una operación POP, no cobramos nada a la operación y pagamos su costo real utilizando el
crédito almacenado en la pila. Para hacer estallar un plato, sacamos el dólar de crédito del plato y lo usamos para
pagar el costo real de la operación. Por lo tanto, al cargar un poco más la operación PUSH,
no necesitamos cargar nada a la operación POP.

Además, tampoco necesitamos cobrar nada a las operaciones MULTIPOP. Para hacer estallar el primer plato,
sacamos el dólar de crédito del plato y lo usamos para pagar el costo real de una operación POP.
Para hacer estallar un segundo plato, nuevamente tenemos un dólar de crédito en el plato para pagar el POP.
operación, y así sucesivamente. Por lo tanto, siempre hemos cobrado lo suficiente por adelantado para pagar MULTIPOP
operaciones. En otras palabras, dado que cada plato de la pila tiene 1 dólar de crédito y el
pila siempre tiene un número no negativo de platos, nos hemos asegurado de que la cantidad de crédito sea
siempre no negativo. Por lo tanto, para *cualquier* secuencia de n operaciones PUSH, POP y MULTIPOP,
el costo total amortizado es un límite superior del costo real total. Dado que el total amortizado
el costo es O (n), también lo es el costo real total.

Incrementar un contador binario

Como otra ilustración del método contable, analizamos la operación INCREMENTO en un contador binario que comienza en cero. Como observamos anteriormente, el tiempo de ejecución de esta operación es proporcional al número de bits invertidos, que usaremos como nuestro costo para este ejemplo. Usemos una vez más un billete de un dólar para representar cada unidad de costo (la inversión de un bit en este ejemplo).

Para el análisis amortizado, carguemos un costo amortizado de 2 dólares para establecer un bit en 1. Cuando un bit está configurado, usamos 1 dólar (de los 2 dólares cobrados) para pagar la configuración real del bit, y colocamos el otro dólar en el bit como crédito para usarlo más tarde cuando volvamos el bit a 0. En cualquier momento, cada 1 en el mostrador tiene un dólar de crédito y, por lo tanto, no necesitamos cargue cualquier cosa para restablecer un bit a 0; solo pagamos el reinicio con el billete de un dólar en el bit.

Ahora se puede determinar el costo amortizado de INCREMENTO. El costo de restablecer los bits dentro del **tiempo** de bucle es pagado por los dólares de los bits que se restablecen. Como máximo, se establece un bit, en la línea 6 de INCREMENTO, y por lo tanto el costo amortizado de una operación de INCREMENTO es como máximo 2 dólares. El número de unos en el contador nunca es negativo y, por tanto, la cantidad de el crédito es siempre no negativo. Por lo tanto, para n operaciones de INCREMENTO, el costo total amortizado es $O(n)$, que limita el costo real total.

Ejercicios 17.2-1

Se realiza una secuencia de operaciones de pila en una pila cuyo tamaño nunca excede k. Después cada k operaciones, se realiza una copia de toda la pila con fines de respaldo. Muestre que el costo de n operaciones de pila, incluida la copia de la pila, es $O(n)$ mediante la asignación de costos a las diversas operaciones de pila.

Ejercicios 17.2-2

Rehacer Ejercicio 17,1-3 utilizando un método de contabilidad de análisis.

Ejercicios 17.2-3

Supongamos que deseamos no solo incrementar un contador sino también restablecerlo a cero (es decir, hacer que todos los bits en él 0). Muestre cómo implementar un contador como una matriz de bits para que cualquier secuencia de n Las operaciones de INCREMENTO y REINICIO toman tiempo $O(n)$ en un contador inicialmente cero. (*Pista:* Mantenga un puntero en el orden superior 1.)

17.3 El método potencial

En lugar de representar el trabajo prepago como crédito almacenado con objetos específicos en los datos estructura, el método potencial de análisis amortizado representa el trabajo prepago como "potencial energía ", o simplemente" potencial ", que se puede liberar para pagar operaciones futuras. El potencial es asociado con la estructura de datos como un todo en lugar de con objetos específicos dentro de los datos estructura.

El método potencial funciona de la siguiente manera. Comenzamos con una estructura de datos inicial D_0 en la que n se realizan las operaciones. Para cada $i = 1, 2, \ldots, n$, dejamos que c_i sea el costo real del i- ésimo operación y D_i es la estructura de datos que resulta después de aplicar la operación i a los datos estructura D_{i-1}. Una función potencial Φ asigna cada estructura de datos D_i a un número real $\Phi(D_i)$, cuál es el potencial asociado con la estructura de datos D_i. El costo amortizado del i th operación con respecto a la función potencial Φ se define por

(17,2)

El costo amortizado de cada operación es, por lo tanto, su costo real más el aumento en el potencial debido a la operación. Por la ecuación (17.2), el costo total amortizado de las n operaciones es

(17,3)

La segunda igualdad se deriva de la ecuación (A.9) , ya que los términos $\Phi (D_i)$ telescopio.

Si podemos definir una función potencial Φ de modo que $\Phi (D_n) \geq \Phi (D_0)$, entonces el costo total amortizado es un límite superior del costo real total . En la práctica, no siempre sabemos cómo se pueden realizar muchas operaciones. Por lo tanto, si requerimos que $\Phi (D_i) \geq \Phi (D_0)$ para todo i , luego garantizamos, como en el método contable, que pagamos por adelantado. A menudo es conveniente

Página 355

definir $\Phi (D_0)$ como 0 y luego mostrar que $\Phi (D_i) \geq 0$ para todo i . (Consulte el ejercicio 17.3-1 para manera fácil de manejar casos en los que $\Phi (D_0) \neq 0$.)

Intuitivamente, si la diferencia de potencial $\Phi (D_i) - \Phi (D_{i-1})$ de la i- ésima operación es positiva, entonces el El costo amortizado representa un sobrecoste a la i- ésima operación, y el potencial de los datos aumenta la estructura. Si la diferencia de potencial es negativa, entonces el costo amortizado representa un recargo a la i- ésima operación, y el costo real de la operación se paga con la disminución en el potencial.

Los costos amortizados definidos por las ecuaciones (17.2) y (17.3) dependen de la elección del función potencial Φ. Diferentes funciones potenciales pueden producir diferentes costos amortizados pero aún así Ser límites superiores a los costos reales. A menudo se pueden hacer concesiones al elegir un función potencial; la mejor función potencial a utilizar depende de los límites de tiempo deseados.

Operaciones de pila

Para ilustrar el método potencial, volvemos una vez más al ejemplo de las operaciones de pila PUSH, POP y MULTIPOP. Definimos la función potencial Φ en una pila como el número de objetos en la pila. Para la pila vacía D_0 con la que comenzamos, tenemos $\Phi (D_0) = 0$. Dado que el número de objetos en la pila nunca es negativo, la pila D_i que resulta después del i- ésimo operación tiene potencial no negativo, y por lo tanto

$$\Phi (D_i) \geq 0$$
$$= \Phi (D_0).$$

El costo total amortizado de n operaciones con respecto a Φ por lo tanto representa un límite superior sobre el costo real.

Calculemos ahora los costos amortizados de las diversas operaciones de pila. Si la i- ésima operación en una pila que contiene s objetos es una operación PUSH, entonces la diferencia de potencial es

$$\Phi (D_{yo}) - \Phi (D_{yo-1}) = (s + 1) - s$$
$$= 1$$

Por la ecuación (17.2) , el costo amortizado de esta operación PUSH es

$$= do_{yo} + \Phi (D_{yo}) - \Phi (D_{yo} -1)$$
$$= 1 + 1$$
$$= 2$$

Suponga que la i- ésima operación en la pila es MULTIPOP (S , k) y que $' = \min (k , s)$ objetos se sacan de la pila. El costo real de la operación es $'$ y la diferencia de potencial es

$$\Phi (D_{yo}) - \Phi (D_{yo-1}) - - ' .$$

Así, el costo amortizado de la operación MULTIPOP es

$$= do_{yo} + \Phi (D_{yo}) - \Phi (D_{yo-1})$$

Página 356

= ′ - ′
= 0.

De manera similar, el costo amortizado de una operación POP ordinaria es 0.

El costo amortizado de cada una de las tres operaciones es $O(1)$, y por lo tanto el costo total amortizado de una secuencia de n operaciones es $O(n)$. Como ya hemos argumentado que $\Phi(D_i) \geq \Phi(D_0)$, el total El costo amortizado de n operaciones es un límite superior del costo real total. El costo del peor de los casos de n operaciones es por tanto $O(n)$.

Incrementar un contador binario

Como otro ejemplo del método potencial, volvemos a considerar el incremento de un contador binario. Esta vez, definimos el potencial del contador después de la i- ésima operación INCREMENT como b_i , el número de ′ s en el contador después de la i- ésima operación.

Calculemos el costo amortizado de una operación INCREMENTO. Suponga que el i th La operación INCREMENT restablece t_i bits. El costo real de la operación es, por lo tanto, como máximo t_i + 1, ya que además de restablecer los bits t_i , establece como máximo un bit en 1. Si $b_i = 0$, entonces el i ésimo la operación restablece todos los k bits, por lo que $b_{i-1} = t_i = k$. Si $b_i > 0$, entonces $b_i = b_{i-1} - t_i + 1$. En cualquier caso, $b_i \leq b_{i-1} - t_i + 1$, y la diferencia de potencial es

$$\Phi(D_{yo}) - \Phi(D_{yo-1}) \leq (\textit{segundo}_{yo-1} - t_{yo} + 1) - \textit{segundo}_{yo-1}$$
$$= 1 - t_i .$$

Por tanto, el coste amortizado es

= $do_{yo} + \Phi(D_{yo}) - \Phi(D_{yo-1})$
$\leq (t_{yo} + 1) + (1 - t_{yo})$
= 2.

Si el contador comienza en cero, entonces $\Phi(D_0) = 0$. Dado que $\Phi(D_i) \geq 0$ para todo i , el costo total amortizado de una secuencia de n operaciones INCREMENT es un límite superior en el costo real total, por lo que el costo del peor caso de n operaciones INCREMENT es $O(n)$.

El método potencial nos brinda una manera fácil de analizar el contador incluso cuando no se inicia en cero. Inicialmente hay b_0 ′ s, y después de n operaciones INCREMENT hay b_n 1, donde 0 $\leq b_0, b_n \leq k$. (Recuerde que k es el número de bits en el contador). Podemos reescribir la ecuación (17.3) como

(17,4)

Tenemos ≤ 2 para todo $1 \leq i \leq n$. Dado que $\Phi(D_0) = b_0$ y $\Phi(D_n) = b_n$, el costo real total de n Las operaciones INCREMENT son

Página 357

Tenga en cuenta en particular que dado que $b_0 \leq k$, siempre que $k = O(n)$, el costo real total es $O(n)$. En otra

En palabras, si ejecutamos al menos $n = \Omega(k)$ operaciones de INCREMENTO, el costo real total es $O(n)$, no importa qué valor inicial contenga el contador.

Ejercicios 17.3-1

Supongamos que tenemos una función potencial Φ tal que $\Phi(D_i) \geq \Phi(D_0)$ para todo i, pero $\Phi(D_0) \neq = 0$. Demuestre que existe una función potencial Φ' tal que $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ para todo $i \geq 1$, y los costos amortizados usando Φ' son los mismos que los costos amortizados usando Φ.

Ejercicios 17.3-2

Volver a realizar ejercicio 17,1-3 utilizando un método potencial del análisis.

Ejercicios 17.3-3

Considere una estructura de datos de min-heap binaria ordinaria con n elementos que admiten la instrucciones INSERT y EXTRACT-MIN en $O(\lg n)$ en el peor de los casos. Dar un potencial función Φ tal que el costo amortizado de INSERT sea $O(\lg n)$ y el costo amortizado de EXTRACT-MIN es $O(1)$ y muestra que funciona.

Ejercicios 17.3-4

¿Cuál es el costo total de ejecutar n de las operaciones de pila PUSH, POP y MULTIPOP? asumiendo que la pila comienza con s_0 objetos y termina con s_n objetos?

Ejercicios 17.3-5

Suponga que un contador comienza en un número con b 1 en su representación binaria, en lugar de en 0. Demuestre que el costo de realizar n operaciones INCREMENTO es $O(n)$ si $n = \Omega(b)$. (No haga suponga que b es constante.)

Página 358

Ejercicios 17.3-6

Muestre cómo implementar una cola con dos pilas ordinarias (ejercicio 10.1-6) para que el El costo amortizado de cada operación ENQUEUE y DEQUEUE es $O(1)$.

Ejercicios 17.3-7

Diseñe una estructura de datos para soportar las siguientes dos operaciones para un conjunto S de enteros:

INSERT (S , x) inserta x en conjunto S .

-DELETE LARGER la mitad (S) borra los mayores $\lceil S/2 \rceil$ elementos de S .

Explique cómo implementar esta estructura de datos para que cualquier secuencia de m operaciones se ejecute en

$O(m)$ tiempo.

17.4 Tablas dinámicas

En algunas aplicaciones, no sabemos de antemano cuántos objetos se almacenarán en una tabla. Podríamos asignar espacio para una mesa, solo para descubrir más tarde que no es suficiente. La mesa debe luego reasignar con un tamaño mayor, y todos los objetos almacenados en la tabla original deben ser copiado en la nueva tabla más grande. Del mismo modo, si se han eliminado muchos objetos del mesa, puede valer la pena reasignar la mesa con un tamaño más pequeño. En esta sección, estudiamos este problema de expandir y contraer dinámicamente una mesa. Utilizando el análisis amortizado, deberá mostrar que el costo amortizado de inserción y eliminación es solo $O(1)$, aunque el El costo real de una operación es grande cuando desencadena una expansión o una contracción. Además, Veremos cómo garantizar que el espacio no utilizado en una tabla dinámica nunca supere un fracción constante del espacio total.

Suponemos que la tabla dinámica admite las operaciones TABLE-INSERT y TABLE-ELIMINAR. TABLE-INSERT inserta en la tabla un elemento que ocupa un solo **espacio**, es decir, un espacio para un artículo. Del mismo modo, TABLE-DELETE se puede considerar como eliminar un elemento de la mesa, liberando así una ranura. Los detalles del método de estructuración de datos utilizado para organizar la mesa no es importante; podríamos usar una pila (Sección 10.1), un montón (Capítulo 6) o un hash tabla (Capítulo 11). También podríamos usar una matriz o colección de matrices para implementar el objeto almacenamiento, como hicimos en la Sección 10.3.

Nos resultará conveniente utilizar un concepto introducido en nuestro análisis de hash (capítulo 11). Definimos el **factor de carga** $\alpha(T)$ de una tabla no vacía T como el número de elementos almacenados en la mesa dividida por el tamaño (número de espacios) de la mesa. Asignamos una mesa vacía (una sin elementos) tamaño 0, y definimos su factor de carga como 1. Si el factor de carga de una dinámica tabla está delimitada debajo por una constante, el espacio no utilizado en la tabla nunca es más que un fracción constante de la cantidad total de espacio.

Página 359

Comenzamos analizando una tabla dinámica en la que solo se realizan inserciones. Nosotros entonces considere el caso más general en el que se permiten tanto las inserciones como las eliminaciones.

17.4.1 Expansión de la tabla

Supongamos que el almacenamiento de una tabla se asigna como una matriz de ranuras. Una mesa se llena cuando todos se han utilizado ranuras o, de manera equivalente, cuando su factor de carga es 1. [1] En algún software entornos, si se intenta insertar un elemento en una tabla completa, no hay alternativa sino abortar con un error. Sin embargo, asumiremos que nuestro entorno de software, como muchos modernos, proporciona un sistema de gestión de memoria que puede asignar y liberar bloques de almacenaje bajo pedido. Por lo tanto, cuando un elemento se inserta en una tabla completa, podemos **expandir** la tabla asignando una nueva mesa con más espacios que la mesa anterior. Porque siempre necesitamos el tabla para residir en la memoria contigua, debemos asignar una nueva matriz para la tabla más grande y luego copie los elementos de la tabla anterior a la tabla nueva.

Una heurística común es asignar una nueva tabla que tenga el doble de espacios que la anterior. Si solo se realizan inserciones, el factor de carga de una tabla es siempre al menos 1/2 y, por lo tanto, el La cantidad de espacio desperdiciado nunca excede la mitad del espacio total en la tabla.

En el siguiente pseudocódigo, asumimos que T es un objeto que representa la tabla. El campo *table* [T] contiene un puntero al bloque de almacenamiento que representa la tabla. El campo *num* [T] contiene el número de elementos de la tabla y el *tamaño* del campo [T] es el número total de espacios en la mesa. Inicialmente, la tabla está vacía: *num* [T] = *tamaño* [T] = 0.

```
INSERCIÓN DE MESA ( T , x )
 1 si el tamaño [ T ] = 0
 2 luego asigne la tabla [ T ] con 1 espacio
 3                tamaño [ T ] ← 1
 4 si num [ T ] = talla [ T ]
 5 luego asigne una nueva tabla con 2 espacios de tamaño [ T ]
 6                inserte todos los elementos de la tabla [ T ] en la nueva tabla
 7                mesa libre [ T ]
 8                tabla [ T ] → nueva-tabla
 9                tamaño [ T ] → 2 · tamaño [ T ]
10 inserte x en la tabla [ T ]
11 num [ T ] → num [ T ] + 1
```

Observe que aquí tenemos dos procedimientos de "inserción": el procedimiento TABLE-INSERT en sí y la inserción elemental en una tabla en las líneas 6 y 10. Podemos analizar el tiempo de ejecución de TABLE-INSERT en términos de número de inserciones elementales asignando un costo de 1 a cada inserción elemental. Suponemos que el tiempo de ejecución real de TABLE-INSERT es lineal en el tiempo para insertar elementos individuales, de modo que la sobrecarga para asignar una tabla inicial en la línea 2 es constante y los gastos generales para asignar y liberar almacenamiento en las líneas 5 y 7 es dominado por el costo de transferir artículos en la línea 6. Llamamos al evento en el que el **entonces** La cláusula de las líneas 5-9 se ejecuta como expansión .

Analicemos una secuencia de n operaciones TABLE-INSERT en una tabla inicialmente vacía. Qué ¿Cuál es el costo c_i de la i ésima operación? Si hay espacio en la tabla actual (o si esta es la primera operación), entonces $c_i = 1$, ya que solo necesitamos realizar una inserción elemental en la línea 10. Si la tabla actual está llena, sin embargo, y ocurre una expansión, entonces $c_i = i$: el costo es 1 para el inserción elemental en la línea 10 más i - 1 para los elementos que deben copiarse de la tabla anterior a

Página 360

la nueva tabla en la línea 6. Si se realizan n operaciones, el costo del peor caso de una operación es $O(n)$, que conduce a un límite superior de $O(n^2)$ en el tiempo de ejecución total para n operaciones.

Este límite no es estricto, porque el costo de expandir la tabla no se soporta a menudo en el curso de n operaciones TABLE-INSERT. Específicamente, la i- ésima operación provoca una expansión sólo cuando i - 1 es una potencia exacta de 2. El costo amortizado de una operación es de hecho $O(1)$, como podemos mostrar usando análisis agregado. El costo de la i- ésima operación es

El costo total de n operaciones TABLE-INSERT es por lo tanto

ya que hay como máximo n operaciones que cuestan 1 y los costos de las operaciones restantes forman una serie geométrica. Dado que el costo total de n operaciones TABLE-INSERT es $3n$, la amortización el costo de una sola operación es 3.

Al usar el método contable, podemos tener una idea de por qué el costo amortizado de un La operación TABLE-INSERT debe ser 3. Intuitivamente, cada artículo paga 3 inserciones: insertándose en la tabla actual, moviéndose cuando la tabla se expande, y mover otro elemento que ya se ha movido una vez cuando se expande la tabla. por Por ejemplo, suponga que el tamaño de la tabla es m inmediatamente después de una expansión. Entonces el el número de elementos de la tabla es $m / 2$ y la tabla no contiene crédito. Cobramos 3 dólares por cada inserción. La inserción elemental que se produce inmediatamente cuesta 1 dólar. Otro El dólar se coloca como crédito en el artículo insertado. El tercer dólar se coloca como crédito en uno de los $m / 2$ elementos ya en la tabla. El llenado de la mesa requiere $m / 2 - 1$ inserciones adicionales y, por lo tanto, para cuando la tabla contenga m elementos y esté llena, cada elemento tiene un dólar para pagar su reinserción durante la expansión.

El método potencial también se puede utilizar para analizar una secuencia de n TABLE-INSERT operaciones, y lo usaremos en la Sección 17.4.2 para diseñar una operación TABLE-DELETE que también tiene un costo amortizado $O(1)$. Comenzamos definiendo una función potencial Φ que es 0 inmediatamente después de una expansión, pero aumenta al tamaño de la mesa cuando la mesa está llena, de modo que la próxima expansión puede pagarse con el potencial. La función

(17,5)

es una posibilidad. Inmediatamente después de una expansión, tenemos $num[T] = tamaño[T] / 2$, y por lo tanto $\Phi(T) = 0$, como se desee. Inmediatamente antes de una expansión, tenemos $num[T] = tamaño[T]$, y por lo tanto $\Phi(T) = num[T]$, como se desee. El valor inicial del potencial es 0, y dado que la tabla es siempre al menos medio lleno, $num[T] \geq tamaño[T] / 2$, lo que implica que $\Phi(T)$ siempre es no negativo. Por tanto, la suma de los costos amortizados de n operaciones TABLE-INSERT es un límite superior en la suma de los costes reales.

Página 361

Para analizar el costo amortizado de la i- ésima operación TABLE-INSERT, dejamos que num_i denote el número de elementos almacenados en la tabla después de la i- ésima operación, el $tamaño_i$ denota el tamaño total del tabla después de la i- ésima operación, y Φ_i denota el potencial después de la i- ésima operación. Inicialmente, nosotros tienen $num_0 = 0$, $tamaño_0 = 0$ y $\Phi_0 = 0$.

Si la i- ésima operación TABLE-INSERT no activa una expansión, entonces tenemos $tamaño_i = tamaño_{i-1}$ y el costo amortizado de la operación es

$$
\begin{aligned}
&= c_{yo} + \Phi_{yo} - \Phi_{yo-1} \\
&= 1 + (2 \cdot num_i - talla_i) - (2 \cdot num_{i-1} - talla_{i-1}) \\
&= 1 + (2 \cdot num_i - talla_i) - (2 (num_{i-1}) - talla_i) \\
&= 3.
\end{aligned}
$$

Si la i- ésima operación desencadena una expansión, entonces tenemos $tamaño_i = 2 \cdot tamaño_{i-1}$ y $tamaño_{i-1} = num_{i-1} = num_{i-1}$, lo que implica que el $tamaño_i = 2 \cdot (num_{i-1})$. Así, el costo amortizado de la operación es

$$
\begin{aligned}
&= c_{yo} + \Phi_{yo} - \Phi_{yo-1} \\
&= num_i + (2 \cdot num_i - tamaño_i) - (2 \cdot num_{i-1} - tamaño_{i-1}) \\
&= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2 (num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3.
\end{aligned}
$$

La figura 17.3 grafica los valores de num_i, $tamaño_i$ y Φ_i contra i. Observe cómo el potencial aumenta pagar por la ampliación de la mesa.

Figura 17.3: El efecto de una secuencia de n operaciones TABLE-INSERT en el número num_i de elementos de la tabla, el $tamaño$ del número i de las ranuras en la tabla y el potencial $\Phi_i = 2 \cdot num_i - tamaño_i$, cada uno medido después de la i- ésima operación. La línea delgada muestra num_i, la línea discontinua muestra $tamaño_i$, y la línea gruesa muestra Φ_i. Observe que inmediatamente antes de una expansión, el potencial se ha acumulado hasta el número de elementos de la tabla y, por lo tanto, puede pagar por mover todos los elementos elementos a la nueva mesa. Posteriormente, el potencial cae a 0, pero se incrementa inmediatamente en 2 cuando se inserta el elemento que causó la expansión.

17.4.2 Expansión y contracción de la mesa

Página 362

Para implementar una operación TABLE-DELETE, es bastante simple eliminar el elemento especificado de la mesa. Sin embargo, a menudo es deseable contraer la mesa cuando el factor de carga del La mesa se vuelve demasiado pequeña, por lo que el espacio desperdiciado no es exorbitante. La contracción de la mesa es

análogo a la expansión de la tabla: cuando el número de elementos de la tabla es demasiado bajo, asigne una tabla nueva y más pequeña y luego copie los elementos de la tabla anterior en la nueva. los El almacenamiento de la tabla antigua se puede liberar devolviéndola al sistema de gestión de memoria. Idealmente, nos gustaría conservar dos propiedades:

- el factor de carga de la tabla dinámica está acotado por debajo por una constante, y
- el costo amortizado de una operación de tabla está acotado arriba por una constante.

Suponemos que el costo se puede medir en términos de inserciones y eliminaciones elementales.

Una estrategia natural para la expansión y la contracción es duplicar el tamaño de la mesa cuando un artículo está insertado en una tabla completa y reducir a la mitad el tamaño cuando una eliminación haría que la tabla se convierta menos de la mitad. Esta estrategia garantiza que el factor de carga de la tabla nunca caiga por debajo 1/2, pero desafortunadamente, puede causar que el costo amortizado de una operación sea bastante grande. Considere el siguiente escenario. Realizamos n operaciones en una tabla T, donde n es una exacta potencia de 2. Las primeras n / 2 operaciones son inserciones, que según nuestro análisis anterior cuestan un total de $\Phi(n)$. Al final de esta secuencia de inserciones, $num[T] = tamaño[T] = n/2$. Para el segundo n / 2 operaciones, realizamos la siguiente secuencia:

Yo, D, D, yo, yo, D, D, yo, yo, ...,

donde I representa una inserción y D representa una eliminación. La primera inserción provoca una Ampliación de la mesa al tamaño n. Las dos supresiones siguientes provocan una contracción de la mesa volver a la talla n / 2. Dos inserciones más provocan otra expansión, y así sucesivamente. El costo de cada expansión y contracción es $\Theta(n)$, y hay $\Theta(n)$ de ellas. Por lo tanto, el costo total de la n operaciones es $\Theta(n^2)$ y el costo amortizado de una operación es $\Theta(n)$.

La dificultad con esta estrategia es obvia: después de una expansión, no realizamos lo suficiente eliminaciones para pagar una contracción. Asimismo, después de una contracción, no realizamos lo suficiente inserciones para pagar una expansión.

Podemos mejorar esta estrategia permitiendo que el factor de carga de la tabla caiga por debajo de 1/2. Específicamente, continuamos duplicando el tamaño de la tabla cuando se inserta un elemento en una tabla completa, pero reducimos a la mitad el tamaño de la tabla cuando una eliminación hace que la tabla se llene menos de 1/4, en lugar de más de la mitad como antes. Por lo tanto, el factor de carga de la tabla está acotado a continuación por la constante 1/4. La idea es que después de una expansión, el factor de carga de la tabla sea 1/2. Por tanto, la mitad de los elementos en la tabla debe eliminarse antes de que pueda ocurrir una contracción, ya que la contracción no ocurre a menos que el factor de carga caiga por debajo de 1/4. Asimismo, después de una contracción, el factor de carga de la mesa también es 1/2. Por lo tanto, el número de elementos de la tabla debe duplicarse mediante inserciones antes de que pueda ocurrir una expansión, ya que la expansión ocurre solo cuando el factor de carga exceder 1.

Omitimos el código para TABLE-DELETE, ya que es análogo a TABLE-INSERT. Es conveniente asumir para el análisis, sin embargo, que si el número de elementos en la tabla cae a 0, se libera el almacenamiento de la mesa. Es decir, si $num[T] = 0$, entonces $size[T] = 0$.

Página 363

Ahora podemos usar el método potencial para analizar el costo de una secuencia de n TABLE-INSERT y operaciones TABLE-DELETE. Comenzamos definiendo una función potencial Φ que es 0 inmediatamente después de una expansión o contracción y aumenta a medida que el factor de carga aumenta a 1 o disminuye a 1/4. Denotemos el factor de carga de una mesa T no vacía por $\alpha(T) = num[T] / tamaño[T]$. Como para una tabla vacía, $num[T] = tamaño[T] = 0$ y $\alpha[T] = 1$, siempre tenemos $num[T] = \alpha(T) \cdot tamaño[T]$, si la mesa está vacía o no. Usaremos como nuestra función potencial

(17,6)

Observe que el potencial de una tabla vacía es 0 y que el potencial nunca es negativo. Así, el costo total amortizado de una secuencia de operaciones con respecto a Φ es un límite superior en el costo real de la secuencia.

Antes de proceder con un análisis preciso, nos detenemos a observar algunas propiedades del función potencial. Observe que cuando el factor de carga es 1/2, el potencial es 0. Cuando la carga factor es 1, tenemos $tamaño[T] = num[T]$, lo que implica $\Phi(T) = num[T]$, y por lo tanto el potencial

puede pagar una expansión si se inserta un elemento. Cuando el factor de carga es 1/4, tenemos un *tamaño* [*T*] = 4 · *num* [*T*], lo que implica Φ (*T*) = *num* [*T*], por lo que el potencial puede pagar una contracción si se elimina un elemento. La figura 17.4 ilustra cómo se comporta el potencial para una secuencia de operaciones.

Figura 17.4: El efecto de una secuencia de *n* operaciones TABLE-INSERT y TABLE-DELETE en el número num_i de elementos en la tabla, el *tamaño* del número $_i$ de ranuras en la tabla, y el potencial cada uno medido después de la *i*- ésima operación. El delgado la línea muestra num_i , la línea discontinua muestra el $tamaño_i$, y la línea gruesa muestra Φ_i . Darse cuenta de inmediatamente antes de una expansión, el potencial se ha acumulado hasta el número de elementos en el mesa y, por lo tanto, puede pagar por mover todos los elementos a la nueva mesa. Igualmente, inmediatamente antes de una contracción, el potencial se ha acumulado hasta el número de elementos en el mesa.

Para analizar una secuencia de *n* operaciones TABLE-INSERT y TABLE-DELETE, dejamos c_i denotar el costo real de la *i*- ésima operación, denotar su costo amortizado con respecto a Φ, num_i denota el número de elementos almacenados en la tabla después de la *i*- ésima operación, $tamaño_i$ denota el tamaño total de la tabla después de la *i*- ésima operación, α_i denota el factor de carga de la tabla después de la *i*- ésima operación, y Φ_i denota el potencial después de la *i*- ésima operación. Inicialmente, $num_0 = 0$, $tamaño_0 = 0$, $\alpha_0 = 1$ y Φ0 = 0.

Página 364

Comenzamos con el caso en el que la *i*- ésima operación es TABLE-INSERT. El análisis es idéntico al de la expansión de la tabla en la Sección 17.4.1 si $\alpha_{i-1} \geq 1/2$. Ya sea que la mesa se expanda o no, el el costo amortizado de la operación es como máximo 3. Si $\alpha_{i-1} < 1/2$, la tabla no se puede expandir como resultado de la operación, ya que la expansión ocurre solo cuando $\alpha_{i-1} = 1$. Si $\alpha_i < 1/2$ también, entonces el El costo amortizado de la *i*- ésima operación es

$$
\begin{aligned}
&= c_{yo} + \Phi_{yo} - \Phi_{yo-1} \\
&= 1 + (tamaño_i / 2 - num_i) - (tamaño_{i-1} / 2 - num_i - 1) \\
&= 1 + (tamaño_i / 2 - num_i) - (tamaño_i / 2 - (num_i - 1)) \\
&= 0.
\end{aligned}
$$

Si $\alpha_{i-1} < 1/2$ pero $\alpha_i \geq 1/2$, entonces

$$
\begin{aligned}
&= c_{yo} + \Phi_{yo} - \Phi_{yo-1} \\
&= 1 + (2 \cdot num_i - talla_i) - (talla_{i-1} / 2 - num_{i-1}) \\
&= 1 + (2 (num_{i-1} + 1) - tamaño_{i-1}) - (tamaño_{i-1} / 2 - num_{i-1}) \\
&= 3 \cdot núm_{i-1} - 3/2; tamaño_{i-1} + 3 \\
&= 3\alpha_{i-1} tamaño_{i-1} - 3/2 \ tamaño_{i-1} + 3 \\
&< 3/2 \ tamaño_{i-1} - 3/2 \ tamaño_{i-1} + 3 \\
&= 3.
\end{aligned}
$$

Por lo tanto, el costo amortizado de una operación TABLE-INSERT es como máximo 3.

Pasamos ahora al caso en el que la *i*- ésima operación es TABLE-DELETE. En este caso, $num_i = num_{i-1} - 1$. Si $a_{i-1} < 1/2$, entonces debemos considerar si la operación causa una contracción. Si se no, entonces $tamaño_i = tamaño_{i-1}$ y el costo amortizado de la operación es

$$
\begin{aligned}
&= c_{yo} + \Phi_{yo} - \Phi_{yo-1} \\
&= 1 + (tamaño_i / 2 - num_i) - (tamaño_{i-1} / 2 - num_{i-1})
\end{aligned}
$$

$= 1 + (tamaño_i / 2 - num_i) - (tamaño_i / 2 - (num_i + 1))$
$= 2.$

Si $\alpha_i - 1 < 1/2$ y la i- ésima operación desencadena una contracción, entonces el costo real de la
La operación es $c_i = num_i + 1$, ya que eliminamos un elemento y movemos num_i elementos. Tenemos $talla_i / 2 = tamaño_{i-1} / 4 = num_{i-1} = num_i + 1$, y el costo amortizado de la operación es

$= c_{yo} + \Phi_{yo} - \Phi_{yo-1}$
$= (num_i + 1) + (tamaño_i / 2 - num_i) (tamaño_{i-1} / 2 - num_{i-1}) = (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1))$
$= 1.$

Cuando la i- ésima operación es TABLE-DELETE y $\alpha_{i-1} \geq 1/2$, el costo amortizado también es
delimitado por encima de una constante. El análisis se deja como ejercicio 17.4-2 .

En resumen, dado que el costo amortizado de cada operación está acotado arriba por una constante, la
el tiempo real para cualquier secuencia de n operaciones en una tabla dinámica es $O (n)$.

Página 365

Ejercicios 17.4-1

Supongamos que deseamos implementar una tabla hash dinámica de direcciones abiertas. ¿Por qué podríamos
considerar que la tabla está llena cuando su factor de carga alcanza algún valor α que es estrictamente menor que
1? Describa brevemente cómo hacer que la inserción en una tabla hash dinámica de dirección abierta se ejecute en
de forma que el valor esperado del costo amortizado por inserción sea O (1). ¿Por qué es lo esperado
¿El valor del costo real por inserción no es necesariamente O (1) para todas las inserciones?

Ejercicios 17.4-2

Demuestre que si $\alpha_{i-1} \geq 1/2$ y la i- ésima operación en una tabla dinámica es TABLE-DELETE, entonces el
el costo amortizado de la operación con respecto a la función potencial (17.6) está acotado arriba
por una constante.

Ejercicios 17.4-3

Suponga que en lugar de contraer una mesa reduciendo a la mitad su tamaño cuando su factor de carga cae por debajo de
1/4, lo contraemos multiplicando su tamaño por 2/3 cuando su factor de carga cae por debajo de 1/3. Utilizando
la función potencial

$\Phi (T) = | 2 \cdot num [T] - tamaño [T] |$,

mostrar que el costo amortizado de un TABLE-DELETE que utiliza esta estrategia está acotado por encima
por una constante.

Problemas 17-1: contador binario con inversión de bits

El Capítulo 30 examina un algoritmo importante llamado Transformada Rápida de Fourier o FFT. los
El primer paso del algoritmo FFT realiza una permutación de inversión de bits en una matriz de entrada A [0 n
- 1] cuya longitud es $n = 2^k$ para algún entero no negativo k . Esta permutación intercambia elementos
cuyos índices tienen representaciones binarias que son inversas entre sí.

Podemos expresar cada índice a como una secuencia de k bits a_{k-1} , a_{k-2} , ..., a_0 , donde . Nosotros
definir

rev k (a_{k-1} , a_{k-2} , ..., a_0) = a_0 , a_1 , ..., a_{k-1} ;

así,

Página 366

Por ejemplo, si $n = 16$ (o, de manera equivalente, $k = 4$), entonces $\text{rev}_k(3) = 12$, ya que el 4-bit
La representación de 3 es 0011, que cuando se invierte da 1100, la representación de 4 bits de 12.

a. Dada una función rev_k que se ejecuta en $\Phi(k)$ tiempo, escriba un algoritmo para realizar el bit-permutación de inversión en una matriz de longitud $n = 2_k$ en tiempo $O(nk)$.

Podemos utilizar un algoritmo basado en un análisis amortizado para mejorar el tiempo de ejecución del
permutación de inversión de bits. Mantenemos un "contador de bits invertidos" y un procedimiento BIT-
INCREMENTO INVERTIDO que, cuando se le da un valor de contador invertido de bit a, produce
$\text{rev}_k(\text{rev}_k(a) + 1)$. Si $k = 4$, por ejemplo, y el contador de bit invertido comienza en 0, entonces sucesivos
las llamadas a BIT-REVERSED-INCREMENT producen la secuencia

0000, 1000, 0100, 1100, 0010, 1010, ... = 0, 8, 4, 12, 2, 10, ...

si. Suponga que las palabras en su computadora almacenan valores de k bits y que en unidad de tiempo, su
La computadora puede manipular los valores binarios con operaciones como desplazarse a la izquierda o
derecho por cantidades arbitrarias, bit a bit-AND, bit a bit-OR, etc. Describa un
implementación del procedimiento BIT-REVERSED-INCREMENT que permite la
permutación inversa en una matriz de n elementos que se realizará en un total de $O(n)$ tiempo.

C. Suponga que puede desplazar una palabra hacia la izquierda o hacia la derecha solo un bit en la unidad de tiempo. Es todavía
¿Es posible implementar una permutación de inversión de bits en tiempo $O(n)$?

Problemas 17-2: Hacer una búsqueda binaria dinámica

La búsqueda binaria de una matriz ordenada requiere tiempo de búsqueda logarítmica, pero el tiempo para insertar una nueva
El elemento es lineal en el tamaño de la matriz. Podemos mejorar el tiempo de inserción manteniendo
varias matrices ordenadas.

Específicamente, suponga que deseamos admitir SEARCH e INSERT en un conjunto de n elementos.

Vamos $k = \lceil \lg(n + 1) \rceil$, y dejar que la representación binaria de n ser $n_{k-1}, n_{k-2}, ..., n_0$. Tenemos k
matrices ordenadas $A_0, A_1, ..., A_{k-1}$, donde para $i = 0, 1, ..., k - 1$, la longitud de la matriz A_i es 2_i. Cada arreglo
está lleno o vacío, dependiendo de si $n_i = 1$ o $n_i = 0$, respectivamente. El número total
de elementos contenidos en todas las k matrices es por lo tantoAunque cada matriz individual es
ordenados, no existe una relación particular entre los elementos de diferentes matrices.

a. Describa cómo realizar la operación de BÚSQUEDA para esta estructura de datos. Analizar su
tiempo de ejecución en el peor de los casos.

si. Describe cómo insertar un nuevo elemento en esta estructura de datos. Analiza su peor caso
y tiempos de ejecución amortizados.

C. Analice cómo implementar DELETE.

Página 367

Problemas 17-3: Árboles amortizados con equilibrio de peso

Considere un árbol de búsqueda binario ordinario aumentado agregando a cada nodo *x* el *tamaño* del campo [*x*] dando el número de claves almacenadas en el subárbol enraizado en *x* . Sea α una constante en el rango 1/2 ≤ α <1. Decimos que un nodo *x* dado está **α-balanceado** si *tamaño* [*izquierda* [*x*]] ≤ α · *tamaño* [*x*]

y

tamaño [*derecha* [*x*]] ≤ α · *tamaño* [*x*].

El árbol en su conjunto está **equilibrado en α** si cada nodo del árbol está en equilibrio α. El seguimiento G. Varghese sugirió un enfoque amortizado para mantener árboles con peso equilibrado.

a. Un árbol 1/2 equilibrado es, en cierto sentido, tan equilibrado como puede ser. Dado un nodo *x* en un árbol de búsqueda binaria arbitraria, muestra cómo reconstruir el subárbol enraizado en *x* para que se vuelve 1/2 equilibrado. Su algoritmo debería ejecutarse en el tiempo Θ (*tamaño* [*x*]), y puede usar *O* (*tamaño* [*x*]) almacenamiento auxiliar.
si. Demuestre que realizar una búsqueda en un árbol de búsqueda binaria balanceada α de *n* nodos toma *O* (lg *n*) momento del peor de los casos.

Para el resto de este problema, suponga que la constante α es estrictamente mayor que 1/2.
Suponga que INSERT y DELETE se implementan como de costumbre para una búsqueda binaria de *n* nodos árbol, excepto que después de cada operación de este tipo, si algún nodo del árbol ya no tiene equilibrio α, entonces el subárbol enraizado en el nodo más alto del árbol se "reconstruye" para que se convierta en 1/2 equilibrado.

Analizaremos este esquema de reconstrucción utilizando el método potencial. Para un nodo *x* en un binario árbol de búsqueda *T* , definimos

Δ (*x*) = | *tamaño* [*izquierda* [*x*]] - *tamaño* [*derecha* [*x*]] | ,

y definimos el potencial de *T* como

donde *c* es una constante suficientemente grande que depende de α.

C. Argumente que cualquier árbol de búsqueda binaria tiene potencial no negativo y que un árbol tiene potencial 0.
re. Suponga que *m* unidades de potencial pueden pagar la reconstrucción de un subárbol de *m* nodos. Que tan grande debe ser *c* en términos de α para que tome *O* (1) tiempo amortizado para reconstruir un subárbol que no es α-balanceada?
mi. Muestre que insertar un nodo o eliminar un nodo de un árbol balanceado α de *n* nodos costos *O* (lg *n*) tiempo amortizado.

Problemas 17-4: El costo de reestructurar árboles rojo-negros

Página 368

Hay cuatro operaciones básicas en árboles rojo-negro que realizan **modificaciones estructurales** : nodo inserciones, eliminaciones de nodos, rotaciones y modificaciones de color. Hemos visto que RB-INSERT y RB-DELETE usan solo *O* (1) rotaciones, inserciones de nodos y eliminaciones de nodos para mantener la propiedades rojo-negro, pero pueden hacer muchas más modificaciones de color.

a. Describa un árbol rojo-negro legal con *n* nodos de modo que llame a RB-INSERT para agregar el (*n* + 1) st nodo causa modificaciones de color Ω (lg *n*). Luego describe un árbol rojo-negro legal con *n* nodos para los cuales llamar a RB-DELETE en un nodo particular causa color Ω (lg *n*) modificaciones.

Aunque el número de modificaciones de color en el peor de los casos por operación puede ser logarítmico, demostrará que cualquier secuencia de *m* operaciones RB-INSERT y RB-DELETE en un El árbol rojo-negro vacío provoca modificaciones estructurales *O* (*m*) en el peor de los casos.

si. Algunos de los casos manejados por el bucle principal del código de ambos RB-INSERT-FIXUP y RB-DELETE-FIXUP están terminando : una vez encontrados, hacen que el bucle se terminar después de un número constante de operaciones adicionales. Para cada uno de los casos de RB-INSERT-FIXUP y RB-DELETE-FIXUP, especifique cuáles están terminando y que no son. (*Sugerencia:* observe las figuras 13.5 , 13.6 y 13.7 .)

Primero analizaremos las modificaciones estructurales cuando solo se realicen inserciones. Deja T ser un árbol rojo-negro, y definir $\Phi(T)$ para ser el número de nodos rojos en T . Suponga que 1 unidad de potencial puede pagar las modificaciones estructurales realizadas por cualquiera de los tres casos de RB-INSERTAR-FIJAR.

C. Deje que T' es el resultado de la aplicación de la caja 1 de RB-insert-corrección a T . Argumenta que $\Phi(T') = \Phi(T) - 1$.

re. La inserción de nodos en un árbol rojo-negro mediante RB-INSERT se puede dividir en tres partes. Enumere las modificaciones estructurales y los posibles cambios resultantes de las líneas 1 a 16 de RB-INSERT, de casos no terminales de RB-INSERT-FIXUP, y de casos de terminación de RB-INSERT-FIXUP.

mi. Utilizando el inciso d), argumente que el número amortizado de modificaciones estructurales realizadas por cualquier llamada de RB-INSERT es $O(1)$.

Ahora deseamos probar que hay $O(m)$ modificaciones estructurales cuando hay ambos inserciones y eliminaciones. Definamos, para cada nodo x ,

Ahora redefinimos el potencial de un árbol rojo-negro T como

Página 369

y sea T' el árbol que resulta de aplicar cualquier caso no final de RB-INSERT-Fixup o RB-delete-corrección a T .

F. Muestre que $\Phi(T') \le \Phi(T) - 1$ para todos los casos no terminales de RB-INSERT-FIXUP. Argumentan que el número amortizado de modificaciones estructurales realizadas por cualquier convocatoria de RB-INSERT-FIXUP es $O(1)$.

gramo. Muestre que $\Phi(T') \le \Phi(T) - 1$ para todos los casos no terminales de RB-DELETE-FIXUP. Argumentan que el número amortizado de modificaciones estructurales realizadas por cualquier convocatoria de RB-DELETE-FIXUP es $O(1)$.

h. Complete la prueba de que, en el peor de los casos, cualquier secuencia de m RB-INSERT y RB-Las operaciones DELETE realizan modificaciones estructurales de $O(m)$.

[1] En algunas situaciones, como una tabla hash de dirección abierta, es posible que deseemos considerar una tabla para estar completo si su factor de carga es igual a alguna constante estrictamente menor que 1. (Vea el ejercicio 17.4-1).

Notas del capítulo

El análisis agregado fue utilizado por Aho, Hopcroft y Ullman [5] . Tarjan [293] examina el métodos contables y potenciales de análisis amortizado y presenta varias aplicaciones. Él atribuye el método contable a varios autores, entre ellos MR Brown, RE Tarjan, S. Huddleston y K. Mehlhorn. Atribuye el método potencial a DD Sleator. El termino "amortizado" se debe a DD Sleator y RE Tarjan.

Las funciones potenciales también son útiles para probar límites inferiores para ciertos tipos de problemas. por cada configuración del problema, definimos una función potencial que mapea la configuración a un número real. Luego determinamos el potencial Φ_{init} de la configuración inicial, el potencial Φ_{final} de la configuración final, y el cambio máximo en el potencial $\Delta\Phi_{\text{máximo}}$ debido a

cualquier paso. Por lo tanto, el número de pasos debe ser al menos $| \Phi_{final} - \Phi_{init} | / | \Delta\Phi_{max} |$. Ejemplos de
El uso de funciones potenciales para probar límites inferiores en la complejidad de E / S aparecen en trabajos de
Cormen [71] , Floyd [91] y Aggarwal y Vitter [4] . Krumme, Cybenko y
Venkataraman [194] aplicó funciones potenciales para demostrar límites más bajos en los chismes :
comunicar un elemento único de cada vértice en un gráfico a todos los demás vértices.

Parte V: Estructuras de datos avanzadas

Lista de capítulos

Capítulo 18: Árboles B
Capítulo 19: Montones binomiales
Capítulo 20: Montones de Fibonacci
Capítulo 21: Estructuras de datos para conjuntos disjuntos

Introducción

Página 370

Esta parte vuelve al examen de estructuras de datos que soportan operaciones en conjuntos dinámicos.
pero a un nivel más avanzado que la Parte III . Dos de los capítulos, por ejemplo, hacen extensos
uso de las técnicas de análisis amortizado que vimos en el Capítulo 17 .

El capítulo 18 presenta los árboles B, que son árboles de búsqueda equilibrados diseñados específicamente para
almacenados en discos magnéticos. Debido a que los discos magnéticos funcionan mucho más lentamente que los aleatorios
memoria de acceso, medimos el rendimiento de los árboles B no solo por la cantidad de cómputo
el tiempo que consumen las operaciones de conjuntos dinámicos, sino también la cantidad de accesos al disco que se realizan.
Para cada operación de árbol B, el número de accesos al disco aumenta con la altura del árbol B,
que se mantiene bajo por las operaciones del árbol B.

Los capítulos 19 y 20 dan implementaciones de montones fusionables, que respaldan las operaciones
INSERT, MINIMUM, EXTRACT-MIN y UNION. [1] La operación UNION une, o
se fusiona, dos montones. Las estructuras de datos de estos capítulos también admiten las operaciones DELETE
y TECLA DE DISMINUCIÓN.

Los montones binomiales, que aparecen en el Capítulo 19 , apoyan cada una de estas operaciones en $O(\lg n)$
en el peor de los casos, donde n es el número total de elementos en el montón de entrada (o en las dos entradas
montones juntos en el caso de UNION). Cuando la operación UNION debe ser apoyada,
Los montones binomiales son superiores a los montones binarios presentados en el Capítulo 6 , porque toma $\Theta(n)$
es hora de unir dos montones binarios en el peor de los casos.

Los montones de Fibonacci, en el capítulo 20 , mejoran los montones binomiales, al menos en un sentido teórico.
Usamos límites de tiempo amortizados para medir el rendimiento de los montones de Fibonacci. los
Las operaciones INSERT, MINIMUM y UNION toman solo $O(1)$ tiempo real y amortizado en
Montones de Fibonacci, y las operaciones EXTRACT-MIN y DELETE toman $O(\lg n)$ amortizado
hora. La ventaja más significativa de los montones de Fibonacci, sin embargo, es que DISMINUYE-KEY
toma solo $O(1)$ tiempo amortizado. El bajo tiempo amortizado de la operación DISMINUCIÓN-CLAVE
Es por eso que los montones de Fibonacci son componentes clave de algunos de los algoritmos asintóticamente más rápidos
hasta la fecha para problemas de gráficos.

Finalmente, el Capítulo 21 presenta estructuras de datos para conjuntos disjuntos. Tenemos un universo de n elementos
que se agrupan en conjuntos dinámicos. Inicialmente, cada elemento pertenece a su propio conjunto de singleton.
La operación UNION une dos conjuntos, y la consulta FIND-SET identifica el conjunto que un determinado
elemento está en este momento. Al representar cada conjunto mediante un árbol enraizado simple, obtenemos
operaciones sorprendentemente rápidas: una secuencia de m operaciones se ejecuta en $O(m\,\alpha(n))$ tiempo, donde $\alpha(n)$ es
una función de crecimiento increíblemente lento: $\alpha(n)$ es como máximo 4 en cualquier aplicación imaginable. los
El análisis amortizado que demuestra que este límite de tiempo es tan complejo como simple la estructura de datos.

Los temas cubiertos en esta parte no son de ninguna manera los únicos ejemplos de datos "avanzados"
estructuras. Otras estructuras de datos avanzadas incluyen las siguientes:

- Árboles dinámicos , introducidos por Sleator y Tarjan [281] y discutidos por Tarjan [292] ,
  mantener un bosque de árboles de raíces inconexas. Cada borde de cada árbol tiene un costo real.
  Los árboles dinámicos admiten consultas para encontrar padres, raíces, costos de borde y el mínimo

costo de borde en una ruta desde un nodo hasta una raíz. Los árboles pueden manipularse cortando bordes, actualizando todos los costos de borde en una ruta desde un nodo hasta una raíz, vinculando una raíz en otro árbol, y convertir un nodo en la raíz del árbol en el que aparece. Una implementación de árboles dinámicos da un límite de tiempo amortizado O (lg n) para cada operación; un mas

la implementación complicada produce O (lg n) límites de tiempo en el peor de los casos. Los árboles dinámicos son utilizado en algunos de los algoritmos de flujo de red asintóticamente más rápidos.

- Splay trees , desarrollados por Sleator y Tarjan [282] y discutidos por Tarjan [292] , son una forma de árbol de búsqueda binaria en la que se ejecutan las operaciones estándar del árbol de búsqueda en O (lg n) tiempo amortizado. Una aplicación de splay trees simplifica los árboles dinámicos.
- Las estructuras de datos persistentes permiten consultas y, a veces, también actualizaciones sobre versiones de una estructura de datos. Driscoll, Sarnak, Sleator y Tarjan [82] presentes técnicas para hacer que las estructuras de datos vinculadas sean persistentes con solo un tiempo y costo de espacio. El problema 13-1 da un ejemplo sencillo de un conjunto dinámico persistente.
- Varias estructuras de datos permiten una implementación más rápida de las operaciones del diccionario. (INSERT, DELETE y SEARCH) para un universo restringido de claves. Tomando ventaja de estas restricciones, pueden lograr mejores resultados asintóticos en el peor de los casos tiempos de ejecución que las estructuras de datos basadas en la comparación. Una estructura de datos inventada por van Emde Boas [301] admite las operaciones MINIMUM, MAXIMUM, INSERT, BORRAR, BÚSQUEDA, EXTRACCIÓN MIN, EXTRACCIÓN MAX, PREDECESOR y SUCESOR en el peor de los casos O (lg lg n), sujeto a la restricción de que el universo de claves es el conjunto {1, 2, ..., n }. Fredman y Willard introdujeron árboles de fusión [99] , que fueron la primera estructura de datos que permitió operaciones de diccionario más rápidas cuando El universo está restringido a números enteros. Mostraron cómo implementar estas operaciones en O (lg n / lg lg n) tiempo. Varias estructuras de datos posteriores, incluida la búsqueda exponencial árboles [16] , también han mejorado los límites en algunos o todos los diccionarios operaciones y se mencionan en las notas del capítulo a lo largo de este libro.
- Las estructuras de datos de gráficos dinámicos admiten varias consultas al tiempo que permiten la estructura de un gráfico para cambiar mediante operaciones que insertan o eliminan vértices o aristas. Los ejemplos de las consultas que son compatibles incluyen la conectividad de vértice [144] , el borde conectividad, árboles de expansión mínimos [143] , biconnectividad y cierre transitivo [142] .

Las notas del capítulo a lo largo de este libro mencionan estructuras de datos adicionales.

[1] Como en el problema 10-2 , hemos definido un montón fusionable para admitir MINIMUM y EXTRACT-MIN, por lo que también podemos referirnos a él como min-heap fusionable . Alternativamente, si soportado MAXIMUM y EXTRACT-MAX, sería un max-heap fusionable . A no ser que especificamos lo contrario, los montones fusionables serán por defecto min-montones fusionables.

Capítulo 18: Árboles B

Visión general

Los árboles B son árboles de búsqueda equilibrados diseñados para funcionar bien en discos magnéticos u otros acceder a dispositivos de almacenamiento secundarios. Los árboles B son similares a los árboles rojo-negro (Capítulo 13), pero son mejores para minimizar las operaciones de E / S del disco. Muchos sistemas de bases de datos utilizan árboles B o variantes de árboles B, para almacenar información.

Los árboles B difieren de los árboles rojo-negro en que los nodos del árbol B pueden tener muchos hijos, desde un puñado a miles. Es decir, el "factor de ramificación" de un árbol B puede ser bastante grande, aunque suele estar determinada por las características de la unidad de disco utilizada. Los árboles B son similares al rojo-negro árboles en que cada árbol B de n nodos tiene una altura O (lg n), aunque la altura de un árbol B puede ser considerablemente menor que el de un árbol rojo-negro porque su factor de ramificación puede ser mucho mayor.

Por lo tanto, los árboles B también se pueden usar para implementar muchas operaciones de conjuntos dinámicos en el tiempo O (lg n).

Los árboles B generalizan los árboles de búsqueda binarios de forma natural. La figura 18.1 muestra un árbol B simple. Si un nodo x del árbol B interno contiene n [x] claves, entonces x tiene n [x] + 1 hijos. Las claves en el nodo x se utilizan como puntos de división que separan el rango de claves manejadas por x en n [x] + 1 subrangos, cada uno manejado por un hijo de x . Al buscar una clave en un árbol B, hacemos un Decisión de la vía (n [x] + 1) basada en comparaciones con las claves n [x] almacenadas en el nodo x . los la estructura de los nodos foliares difiere de la de los nodos internos; examinaremos estas diferencias en la Sección 18.1 .

Figura 18.1: Un árbol B cuyas claves son las consonantes del inglés. Un nodo interno x que contiene n [x] claves tiene n [x] + 1 hijos. Todas las hojas están a la misma profundidad en el árbol. los linfáticos ligeramente sombreadas se examinan en busca de la letra R .

La sección 18.1 da una definición precisa de árboles B y demuestra que la altura de un árbol B crece sólo logarítmicamente con el número de nodos que contiene. La sección 18.2 describe cómo busque una clave e inserte una clave en un árbol B, y la Sección 18.3 trata sobre la eliminación. antes de Para continuar, sin embargo, debemos preguntarnos por qué las estructuras de datos diseñadas para funcionar en un disco magnético se evalúan de manera diferente a las estructuras de datos diseñadas para trabajar en el acceso aleatorio principal memoria.

Estructuras de datos en almacenamiento secundario

Hay muchas tecnologías diferentes disponibles para proporcionar capacidad de memoria en una computadora. sistema. La *memoria principal* (o *memoria principal*) de un sistema informático normalmente consta de chips de memoria de silicio. Esta tecnología suele ser dos órdenes de magnitud más cara. por bit almacenado que la tecnología de almacenamiento magnético, como cintas o discos. La mayoría de las computadoras los sistemas también tienen *almacenamiento secundario* basado en discos magnéticos; la cantidad de tal secundario el almacenamiento a menudo excede la cantidad de memoria primaria en al menos dos órdenes de magnitud.

La figura 18.2 (a) muestra una unidad de disco típica. La unidad consta de varios *platos* , que giran a una velocidad constante alrededor de un *eje* común . La superficie de cada plato está cubierta con un material magnetizable. Cada bandeja es leída o escrita por una *cabeza* al final de un *brazo* . los los brazos están físicamente unidos o "agrupados" juntos, y pueden mover la cabeza hacia o lejos del eje. Cuando una cabeza determinada está estacionaria, la superficie que pasa por debajo se llama *pista* . Los cabezales de lectura / escritura están alineados verticalmente en todo momento y, por lo tanto, el conjunto de las pistas debajo de ellos se accede simultáneamente. La figura 18.2 (b) muestra un conjunto de pistas, que se conoce como *cilindro* .

Página 373

Figura 18.2: *(a)* Una unidad de disco típica. Está compuesto por varios platos que giran alrededor de un huso. Cada bandeja se lee y escribe con una cabeza al final de un brazo. Los brazos están agrupados juntos para que muevan la cabeza al unísono. Aquí, los brazos giran alrededor de un

eje de pivote. Una pista es la superficie que pasa por debajo del cabezal de lectura / escritura cuando está parado.
(b) Un cilindro consta de un conjunto de pistas encubiertas.

Aunque los discos son más baratos y tienen mayor capacidad que la memoria principal, son mucho más
mucho más lento porque tienen partes móviles. Hay dos componentes en la mecánica.
movimiento: rotación del plato y movimiento del brazo. En el momento de escribir estas líneas, los discos básicos giran a
velocidades de 5400 a 15 000 revoluciones por minuto (RPM), siendo 7200 RPM la máxima
común. Aunque 7200 RPM puede parecer rápido, una rotación toma 8,33 milisegundos, que es
casi 5 órdenes de magnitud más largos que los tiempos de acceso de 100 nanosegundos que se encuentran comúnmente
para memoria de silicio. En otras palabras, si tenemos que esperar una rotación completa para que un elemento en particular
bajo el cabezal de lectura / escritura, podríamos acceder a la memoria principal casi 100,000 veces durante
ese lapso! En promedio, tenemos que esperar solo media rotación, pero aún así, la diferencia en
Los tiempos de acceso a la memoria de silicio frente a los discos son enormes. Mover los brazos también requiere
hora. Al momento de escribir este artículo, los tiempos de acceso promedio para los discos básicos están en el rango de 3 a 9
milisegundos.

Para amortizar el tiempo de espera de los movimientos mecánicos, los discos acceden no solo
un artículo pero varios a la vez. La información se divide en varias **páginas** de igual tamaño
bits que aparecen consecutivamente dentro de los cilindros, y cada lectura o escritura de disco es de uno o más
páginas enteras. Para un disco típico, una página puede tener de 2^{11} a 2^{14} bytes de longitud. Una vez que la lectura / escritura
la cabeza está colocada correctamente y el disco ha girado al principio de la página deseada,
leer o escribir un disco magnético es completamente electrónico (aparte de la rotación del disco),
y se pueden leer o escribir grandes cantidades de datos rápidamente.

A menudo, se necesita más tiempo para acceder a una página de información y leerla de un disco que lo que se necesita.
para que la computadora examine toda la información leída. Por esta razón, en este capítulo
mire por separado los dos componentes principales del tiempo de ejecución:

- el número de accesos al disco, y
- el tiempo de la CPU (informática).

El número de accesos al disco se mide en términos del número de páginas de información que
deben leerse o escribirse en el disco. Observamos que el tiempo de acceso al disco no es constante;
depende de la distancia entre la pista actual y la pista deseada y también de la
estado rotacional del disco. No obstante, utilizaremos el número de páginas leídas o escritas como
Aproximación de primer orden del tiempo total empleado en acceder al disco.

Página 374

En una aplicación típica de árbol B, la cantidad de datos manejados es tan grande que no todos los datos
encajar en la memoria principal a la vez. Los algoritmos del árbol B copian las páginas seleccionadas del disco a la
memoria según sea necesario y vuelva a escribir en el disco las páginas que han cambiado. Algoritmos de árbol B
están diseñados para que solo haya un número constante de páginas en la memoria principal en cualquier momento; así,
el tamaño de la memoria principal no limita el tamaño de los árboles B que se pueden manejar.

Modelamos las operaciones de disco en nuestro pseudocódigo de la siguiente manera. Sea x un puntero a un objeto. Si el
El objeto está actualmente en la memoria principal de la computadora, entonces podemos referirnos a los campos del objeto.
como de costumbre: *tecla* [x], por ejemplo. Sin embargo, si el objeto al que hace referencia x reside en el disco, entonces
Debe realizar la operación DISK-READ (x) para leer el objeto x en la memoria principal antes de que podamos
consulte sus campos. (Suponemos que si x ya está en la memoria principal, entonces DISK-READ (x)
no requiere acceso al disco; es un "no-op.") De manera similar, la operación DISK-WRITE (x) se usa para
guarde los cambios que se hayan realizado en los campos del objeto x . Es decir, el patrón típico de
trabajar con un objeto es el siguiente:

- $x \leftarrow$ un puntero a algún objeto
- LECTURA DE DISCO (x)
- operaciones que acceden y / o modifican los campos de x
- DISK-WRITE (x) ▷ Omitido si no se cambiaron campos de x .
- otras operaciones que acceden pero no modifican campos de x

El sistema puede mantener solo un número limitado de páginas en la memoria principal a la vez. Nosotros
asumirá que las páginas que ya no están en uso se vacían de la memoria principal por el sistema; nuestra B-
Los algoritmos de árbol ignorarán este problema.

Dado que en la mayoría de los sistemas, el tiempo de ejecución de un algoritmo de árbol B está determinado principalmente por
número de operaciones DISK-READ y DISK-WRITE que realiza, es sensato utilizar estas
operaciones de manera eficiente al hacer que lean o escriban tanta información como sea posible. Así, un

El nodo del árbol B suele ser tan grande como una página de disco completa. El número de hijos de un nodo de árbol B puede tener, por lo tanto, está limitado por el tamaño de una página de disco.

Para un árbol B grande almacenado en un disco, a menudo se utilizan factores de ramificación entre 50 y 2000, dependiendo del tamaño de una clave en relación con el tamaño de una página. Un gran factor de ramificación reduce drásticamente tanto la altura del árbol como el número de accesos al disco necesarios para encuentra cualquier clave. La figura 18.3 muestra un árbol B con un factor de ramificación de 1001 y una altura 2 que puede almacenar más de mil millones de claves; sin embargo, dado que el nodo raíz se puede mantener permanentemente en la memoria, solo se requieren *dos* accesos al disco como máximo para encontrar cualquier clave en este árbol.

Figura 18.3: Un árbol B de altura 2 que contiene más de mil millones de claves. Cada nodo interno y hoja contiene 1000 llaves. Hay 1001 nudos en la profundidad 1 y más de un millón de hojas en la profundidad 2. Dentro de cada nodo x *se* muestra $n[x]$, el número de claves en x.

18.1 Definición de árboles B

Página 375

Para simplificar las cosas, asumimos, como hemos hecho con los árboles de búsqueda binaria y los árboles rojo-negro, que cualquier "información de satélite" asociada con una clave se almacena en el mismo nodo que la clave. En práctica, uno podría almacenar con cada tecla solo un puntero a otra página de disco que contiene la información de satélite para esa clave. El pseudocódigo de este capítulo asume implícitamente que la información de satélite asociada con una clave, o el puntero a dicha información de satélite, viaja con la llave siempre que la llave se mueve de un nodo a otro. Una variante común en un B-árbol, conocido como B $_+$ - árbol , almacena toda la información por satélite en las hojas y tiendas sólo claves y punteros secundarios en los nodos internos, maximizando así el factor de ramificación del nodos internos.

Un árbol B T es un árbol enraizado (cuya raíz es la $raíz[T]$) que tiene las siguientes propiedades:

1. Cada nodo x tiene los siguientes campos:
   a. $n[x]$, el número de claves almacenadas actualmente en el nodo x,
   si. las $n[x]$ claves mismas, almacenadas en orden no decreciente, de modo que la $clave_1[x] \le tecla_2[x] \le \cdots \le tecla_{n[x]}[x]$,
   C. $hoja[x]$, un valor booleano que es VERDADERO si x es una hoja y FALSO si x es una nodo interno.
2. Cada nodo interno x también contiene $n[x] + 1$ punteros $c_1[x], c_2[x], ..., c_{n[x]+1}[x]$ a su niños. Los nodos de hojas no tienen hijos, por lo que sus campos c_i no están definidos.
3. Las claves $clave_i[x]$ separan los rangos de claves almacenadas en cada subárbol: si k_i es cualquier clave almacenado en el subárbol con root $c_i[x]$, entonces

   $$k_1 \le tecla_1[x] \le k_2 \le tecla_2[x] \le \cdots \le tecla_{n[x]}[x] \le k_{n[x]+1}.$$

4. Todas las hojas tienen la misma profundidad, que es la altura del árbol h.
5. Hay límites inferior y superior en la cantidad de claves que puede contener un nodo. Estas los límites se pueden expresar en términos de un número entero fijo $t \ge 2$ llamado grado mínimo del árbol B:
   a. Cada nodo que no sea la raíz debe tener al menos t - 1 claves. Cada interno un nodo distinto de la raíz tiene por lo menos t hijos. Si el árbol no está vacío, el root debe tener al menos una clave.
   si. Cada nodo puede contener como máximo 2 t - 1 claves. Por lo tanto, un nodo interno puede tener como máximo 2 t *de* hijos. Decimos que un nodo está lleno si contiene exactamente 2 t - 1 llaves. [1]

El árbol B más simple ocurre cuando $t = 2$. Cada nodo interno tiene 2, 3 o 4 hijos, y tenemos un árbol 2-3-4 . En la práctica, sin embargo, normalmente se utilizan valores mucho mayores de t.

La altura de un árbol B

El número de accesos al disco necesarios para la mayoría de las operaciones en un árbol B es proporcional al altura del árbol B. Ahora analizamos la altura del peor caso de un árbol B.

Teorema 18.1

Si $n \geq 1$, entonces para cualquier árbol B de clave n T de altura h y grado mínimo $t \geq 2$,

Página 376

Prueba Si un árbol B tiene una altura h , la raíz contiene al menos una clave y todos los demás nodos contienen en menos t - 1 teclas. Por lo tanto, hay al menos 2 nodos en la profundidad 1, al menos 2 t nodos en la profundidad 2, al menos 2 t^2 nodos en la profundidad 3, y así sucesivamente, hasta que en la profundidad h haya al menos 2 t^{h-1} nodos. Figura 18.4 ilustra un árbol de este tipo para $h = 3$. Por lo tanto, el número n de claves satisface la desigualdad

Figura 18.4: Un árbol B de altura 3 que contiene un número mínimo posible de llaves. Mostrado dentro de cada nodo x está $n[x]$.

Por álgebra simple, obtenemos $t^h \leq (n + 1) / 2$. Tomando logaritmos base- t de ambos lados demuestra la teorema.

Aquí vemos el poder de los árboles B, en comparación con los árboles rojo-negros. Aunque la altura del el árbol crece como $O(\lg n)$ en ambos casos (recuerde que t es una constante), para los árboles B la base del El logaritmo puede ser muchas veces mayor. Por lo tanto, los árboles B ahorran un factor de aproximadamente lg t sobre el rojo-negro árboles en el número de nodos examinados para la mayoría de las operaciones de árboles. Desde examinar un arbitrario nodo en un árbol generalmente requiere un acceso al disco, el número de accesos al disco es sustancialmente reducido.

Ejercicios 18.1-1

¿Por qué no permitimos un grado mínimo de $t = 1$?

Ejercicios 18.1-2

¿Para qué valores de t es el árbol de la figura 18.1 un árbol B legal?

Página 377

Ejercicios 18.1-3

Muestre todos los árboles B legales de grado mínimo 2 que representen {1, 2, 3, 4, 5}.

Ejercicios 18.1-4

En función del grado mínimo t , ¿cuál es el número máximo de teclas que se pueden almacenado en un árbol B de altura h ?

Ejercicios 18.1-5

Describa la estructura de datos que resultaría si cada nodo negro en un árbol rojo-negro fuera absorber sus hijos rojos, incorporando a sus hijos con los suyos.

[1] Otra variante común en un árbol B, conocida como árbol **B *** , requiere que cada nodo interno sea al menos 2/3 de su capacidad, en lugar de al menos la mitad, como requiere un árbol B.

18.2 Operaciones básicas en árboles B

En este apartado presentamos el detalle de las operaciones B-TREE-SEARCH, B-TREE-CREAR y B-TREE-INSERT. En estos procedimientos, adoptamos dos convenciones:

- La raíz del árbol B siempre está en la memoria principal, por lo que se muestra un DISK-READ en la raíz. nunca requerido; se requiere un DISK-WRITE de la raíz, sin embargo, siempre que la raíz se cambia el nodo.
- Todos los nodos que se pasan como parámetros ya deben haber tenido una LECTURA DE DISCO operación realizada en ellos.

Los procedimientos que presentamos son todos algoritmos de "una pasada" que proceden hacia abajo desde la raíz del árbol, sin tener que retroceder.

Buscando un árbol B

Buscar un árbol B es muy parecido a buscar un árbol de búsqueda binario, excepto que en lugar de hacer un decisión de ramificación binaria o "bidireccional" en cada nodo, hacemos una ramificación de múltiples vías decisión según el número de hijos del nodo. Más precisamente, en cada nodo interno x , tomamos una decisión de ramificación de (n [x] + 1) vías.

B-TREE-SEARCH es una generalización sencilla del procedimiento TREE-SEARCH definido para árboles de búsqueda binaria. B-TREE-SEARCH toma como entrada un puntero al nodo raíz x

Página 378

de un subárbol y una clave k que se buscará en ese subárbol. La llamada de nivel superior es, por tanto, del formulario B-TREE-SEARCH (*raíz* [T], k). Si k está en el árbol B, B-TREE-SEARCH devuelve el par ordenado (y , i) que consta de un nodo y y un índice i tal que $clave_i [y] = k$. De lo contrario, el se devuelve el valor NIL.

```
B-BÚSQUEDA DE ÁRBOL ( x , k )
1 i ← 1
2 mientras i ≤ n [ x ] yk > tecla i [ x ]
3                 No i ← i + 1
4 si i ≤ n [ x ] y k = clave i [ x ]
5 luego devuelve ( x , i )
6 si hoja [ x ]
7 luego devuelve NIL
```

8 **más** DISK-READ (c_i [x])
9 **return** B-TREE-SEARCH (c_i [x], k)

Usando un procedimiento de búsqueda lineal, las líneas 1-3 hallan el índice más pequeño i tal que $k \leq tecla_i$ [x], o de lo contrario, establecen i en n [x] + 1. Las líneas 4-5 comprueban si hemos descubierto la clave, volviendo si tenemos. Las líneas 6-9 terminan la búsqueda sin éxito (si x es una hoja) o vuelven a busque el subárbol apropiado de x , después de realizar la DISK-READ necesaria en ese niño.

La figura 18.1 ilustra el funcionamiento de B-TREE-SEARCH; los nodos ligeramente sombreados son examinadas durante la búsqueda de la tecla R .

Como en el procedimiento TREE-SEARCH para árboles de búsqueda binaria, los nodos encontrados durante la recursividad forma un camino hacia abajo desde la raíz del árbol. El número de páginas de disco al que se accede mediante B-TREE-SEARCH es, por tanto, $\Theta(h) = \Theta(\log_t n)$, donde h es la altura del B-árbol yn es el número de claves en el árbol B. Dado que n [x] <2 t , el tiempo que tarda el **tiempo**
El bucle de las líneas 2-3 dentro de cada nodo es $O(t)$, y el tiempo total de CPU es $O(th) = O(t \log_t n)$.

Creando un árbol B vacío

Para construir un árbol B T , primero usamos B-TREE-CREATE para crear un nodo raíz vacío y luego llamamos B-TREE-INSERT para agregar nuevas claves. Ambos procedimientos utilizan un procedimiento auxiliar ALLOCATE-NODE, que asigna una página de disco para ser utilizada como un nuevo nodo en el tiempo O (1). Podemos suponer que un nodo creado por ALLOCATE-NODE no requiere DISK-READ, ya que todavía no hay información útil almacenada en el disco para ese nodo.

B-ARBOL-CREAR (T)
1 $x \leftarrow$ ALLOCATE-NODE ()
2 $hojas$ [x] $\leftarrow$ VERDADERO
3 n [x] $\leftarrow$ 0
4 ESCRITURA EN DISCO (x)
5 $raíz$ [T] $\leftarrow x$

B-TREE-CREATE requiere O (1) operaciones de disco y O (1) tiempo de CPU.

Insertar una llave en un árbol B

Insertar una llave en un árbol B es significativamente más complicado que insertar una llave en un árbol de búsqueda binaria. Al igual que con los árboles de búsqueda binaria, buscamos la posición de la hoja en la que inserte la nueva llave. Con un árbol B, sin embargo, no podemos simplemente crear un nuevo nodo hoja y insértelo, ya que el árbol resultante no sería un árbol B válido. En cambio, insertamos la nueva clave

Página 379

en un nodo hoja existente. Dado que no podemos insertar una clave en un nodo hoja que está lleno, introducir una operación que *divide* un nodo y completo (que tiene 2 claves t - 1) alrededor de su *clave mediana* $clave_t$ [y] en dos nodos que tienen t - 1 claves cada uno. La clave de la mediana se mueve hacia arriba en Y 's padre Identifique el punto divisorio entre los dos nuevos árboles. Pero si y padres 's también está lleno, debe ser dividir antes de que se pueda insertar la nueva clave y, por lo tanto, esta necesidad de dividir nodos completos puede propagarse todo el camino hasta el árbol.

Al igual que con un árbol de búsqueda binaria, podemos insertar una clave en un árbol B en una sola pasada por el árbol. de la raíz a la hoja. Para hacerlo, no esperamos a saber si realmente necesitaremos dividir un nodo completo para realizar la inserción. En cambio, mientras viajamos por el árbol en busca de la posición a la que pertenece la nueva clave, dividimos cada nodo completo al que llegamos en el camino (incluida la propia hoja). Por lo tanto, siempre que queremos dividir un nodo completo y , estamos seguros de que su el padre no está lleno.

Dividir un nodo en un árbol B

El procedimiento B-TREE-SPLIT-CHILD toma como entrada un nodo interno x *no completo* (se supone que estar en la memoria principal), un índice iy un nodo y (también se supone que está en la memoria principal) tal que $y = c_i$ [x] es un hijo *completo* de x . El procedimiento luego divide a este niño en dos y ajusta x para que tiene un hijo adicional. (Para dividir una raíz completa, primero haremos que la raíz sea un elemento secundario de un nuevo nodo raíz, para que podamos usar B-TREE-SPLIT-CHILD. El árbol crece así en altura en uno; dividir es el único medio por el cual el árbol crece).

La figura 18.5 ilustra este proceso. El nodo completo y se divide en torno a su clave mediana S , que es se lleva hasta y 's nodo padre x . Aquellas claves en y que son mayores que la clave mediana son colocado en un nuevo nodo z , que se convierte en un nuevo hijo de x .

Figura 18.5: División de un nodo con $t = 4$. El nodo y se divide en dos nodos, y y z , y el clave mediana S de y se mueve hacia arriba en Y's padres.

```
B-ÁRBOL-DIVIDIDO-NIÑO ( x , i , y )
 1 z ← ALLOCATE-NODE ()
 2 hojas [ z ] ← hoja [ y ]
 3 n [ z ] ← t - 1
 4 para j ← 1 a t - 1
 5 hacer tecla j [ z ] ← tecla j + t [ y ]
 6 si no es hoja [ y ]
 7 entonces para j ← 1 a t
 8                    hacer c j [ z ] ← c j + t [ y ]
 9 n [ y ] ← t - 1
10 para j ← n [ x ] + 1 abajo hacia i + 1
11 do c j + 1 [ x ] ← c j [ x ]
12 c i + 1 [ x ] ← z
13 para j ← n [ x ] hacia abajo i
14 hacer tecla j + 1 [ x ] ← tecla j [ x ]
15 tecla i [ x ] ← tecla i [ y ]
16 n [ x ] ← n [ x ] + 1
17 ESCRITURA EN DISCO ( y )
```

Página 380

```
18 ESCRITURA EN DISCO ( z )
19 ESCRITURA EN DISCO ( x )
```

B-TREE-SPLIT-CHILD funciona simplemente "cortando y pegando". Aquí, y es el i th hijo de xy es el nodo que se divide. El nodo y originalmente tiene 2 t hijos (2 t - 1 claves) pero es reducido a t niños (teclas t - 1) por esta operación. El nodo z "adopta" los t hijos más grandes (t - 1 claves) de y , y z *se* convierte en un nuevo hijo de x , ubicado justo después de y en la tabla de hijos de x . los la clave mediana de y *se* mueve hacia arriba para convertirse en la clave en x que separa y y z .

Las líneas 1-8 crean el nodo z y le dan las claves t - 1 más grandes y los correspondientes hijos t de y . Línea 9 ajusta el número de teclas para y . Finalmente, las líneas 10-16 insertan z como un hijo de x , mueven la tecla mediana desde y hasta x para separar y de z , y ajustar el número de teclas de x . Las líneas 17-19 escriben todas páginas de disco modificadas. El tiempo de CPU utilizado por B-TREE-SPLIT-CHILD es $\Theta (t)$, debido a los bucles en las líneas 4-5 y 7-8. (Los otros bucles se ejecutan para $O (t)$ iteraciones). El procedimiento realiza $O (1)$ operaciones de disco.

Insertar una llave en un árbol B en una sola pasada por el árbol

Insertamos una llave k en un árbol B T de altura h en una sola pasada por el árbol, requiriendo $O (h)$ accesos al disco. El tiempo de CPU requerido es $O (th) = O (t \log_t n)$. El procedimiento B-TREE-INSERT utiliza B-TREE-SPLIT-CHILD para garantizar que la recursividad nunca descienda a un nodo completo.

```
B-INSERTAR-ÁRBOL ( T , k )
 1 r ← raíz [ T ]
 2 si n [ r ] = 2 t - 1
 3 luego s ← ALLOCATE-NODE ()
 4              raíz [ T ] ← s
 5              hoja [ s ] ← FALSO
 6              n [ s ] ← 0
 7              c 1 [ s ] ← r
 8              B-ÁRBOL-DIVIDIDO-NIÑO ( s , 1, r )
 9              B-INSERTAR-ÁRBOL-NO COMPLETO ( s , k )
10 más B-INSERTAR-ÁRBOL-NO COMPLETO ( r , k )
```

Las líneas 3-9 manejan el caso en el que el nodo raíz r está lleno: la raíz está dividida y un nuevo nodo s (tener dos hijos) se convierte en la raíz. Dividir la raíz es la única forma de aumentar la altura de un árbol B. La figura 18.6 ilustra este caso. A diferencia de un árbol de búsqueda binaria, un árbol B aumenta de altura en la parte superior en lugar de en la parte inferior. El procedimiento finaliza llamando a B-TREE-INSERT-NONFULL para realizar la inserción de la clave k en el árbol enraizado en el no completo nodo raíz. B-TREE-INSERT-NONFULL se repite según sea necesario por el árbol, en todo momento garantizando que el nodo al que recurre no esté lleno llamando a B-TREE-SPLIT-CHILD según sea necesario.

Figura 18.6: Dividir la raíz con t nodo = 4. raíz r es dividida en dos, y un nuevo nodo raíz s es decir creado. La nueva raíz contiene la clave mediana de r y tiene las dos mitades de r como hijos. El árbol B crece en altura en uno cuando la raíz se divide.

Página 381

El procedimiento recursivo auxiliar B-TREE-INSERT-NONFULL inserta la clave k en el nodo x , que se supone que no está completo cuando se llama al procedimiento. El funcionamiento de B-TREE-INSERT y la operación recursiva de B-TREE-INSERT-NONFULL garantizan que este la suposición es verdadera.

```
B-INSERTAR-ÁRBOL-NO COMPLETO ( x , k )
 1 i ← n [ x ]
 2 si hoja [ x ]
 3 entonces mientras i ≥ 1 y k < tecla i [ x ]
 4                    hacer tecla i+1 [ x ] ← tecla i [ x ]
 5                          yo ← yo - 1
 6                    tecla i+1 [ x ] ← k
 7                    n [ x ] ← n [ x ] + 1
 8                    ESCRITURA EN DISCO ( x )
 9 más mientras que i ≥ 1 y k < tecla i [ x ]
10                    ¿ Yo ← yo - 1
11                yo ← yo + 1
12                LECTURA DE DISCO ( c i [ x ])
13                si n [ c i [ x ]] = 2 t - 1
14                    luego B-TREE-SPLIT-CHILD ( x , i , c i [ x ])
15                          si k > tecla i [ x ]
dieciséis                       entonces yo ← yo + 1
17                B-INSERTAR-ÁRBOL-NO COMPLETO ( c i [ x ], k )
```

El procedimiento B-TREE-INSERT-NONFULL funciona de la siguiente manera. Las líneas 3-8 manejan el caso en que x es un nodo hoja insertando la clave k en x . Si x no es un nodo hoja, entonces debemos insertar k en el nodo hoja apropiado en el subárbol enraizado en el nodo interno x . En este caso, líneas 9-11 determinar el hijo de x al que desciende la recursividad. La línea 13 detecta si el La recursividad descendería a un hijo completo, en cuyo caso la línea 14 usa B-TREE-SPLIT-CHILD para dividir ese niño en dos niños no completos, y las líneas 15-16 determinan cuál de los dos niños ahora es el correcto para descender. (Tenga en cuenta que no es necesario un DISK-READ (c_i [x]) después de la línea 16 incrementa i , ya que la recursividad descenderá en este caso a un niño que fue creado por B-TREE-SPLIT-CHILD.) El efecto neto de las líneas 13-16 es, por lo tanto, garantizar que el El procedimiento nunca recurre a un nodo completo. Luego, la línea 17 se repite para insertar k en el subárbol. La figura 18.7 ilustra los diversos casos de inserción en un árbol B.

Página 382

Figura 18.7: Inserción de claves en un árbol B. El grado mínimo t para este árbol B es 3, por lo que un nodo Puede contener como máximo 5 llaves. Los nodos que son modificados por el proceso de inserción están ligeramente sombreados. *(a)* El árbol inicial para este ejemplo. *(b)* El resultado de insertar B en el árbol inicial; esto es un inserción simple en un nodo hoja. *(c)* El resultado de insertar Q en el árbol anterior. El nodo $RSTUV$ se divide en dos nodos que contienen RS y UV , la clave T se mueve hacia la raíz y Q se inserta en el extremo izquierdo de las dos mitades (el nodo RS). *(d)* El resultado de insertar L en el árbol anterior. La raíz se divide de inmediato, ya que está llena, y el árbol B crece en altura por uno. Luego, L se inserta en la hoja que contiene JK . *(e)* El resultado de insertar F en el árbol anterior. El nodo $ABCDE$ se divide antes de que F se inserte en el extremo derecho de los dos mitades (el nodo DE).

El número de accesos al disco realizados por B-TREE-INSERT es $O(h)$ para un árbol B de altura h , ya que solo se realizan operaciones $O(1)$ DISK-READ y DISK-WRITE entre llamadas a B-TREE-INSERT-NONFULL. El tiempo total de CPU utilizado es $O(th) = O(t \log_t n)$. Dado que B-TREE-INSERT-NONFULL es recursivo de cola, se puede implementar alternativamente como un **tiempo** bucle, demostrando que la cantidad de páginas que deben estar en la memoria principal en cualquier momento es $O(1)$.

Ejercicios 18.2-1

Mostrar los resultados de insertar las claves

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

en orden en un árbol B vacío con un grado mínimo 2. Dibuje solo las configuraciones del árbol justo antes de que algún nodo deba dividirse, y también dibujar la configuración final.

Página 383

Ejercicios 18.2-2

Explique bajo qué circunstancias, si las hubiera, DISK-READ o DISK-WRITE redundante Las operaciones se realizan durante la ejecución de una llamada a B-TREE-INSERT. (UNA DISK-READ redundante es un DISK-READ para una página que ya está en la memoria. Un redundante DISK-WRITE escribe en el disco una página de información que es idéntica a la que ya está almacenada ahí.)

Ejercicios 18.2-3

Explique cómo encontrar la clave mínima almacenada en un árbol B y cómo encontrar el predecesor de un dada la clave almacenada en un árbol B.

Ejercicios 18.2-4: ⋆

Suponga que las claves {1, 2, ..., n } se insertan en un árbol B vacío con un grado mínimo 2.
¿Cuántos nodos tiene el árbol B final?

Ejercicios 18.2-5

Dado que los nodos de las hojas no requieren punteros para los niños, posiblemente podrían usar una (mayor) valor t que los nodos internos para el mismo tamaño de página de disco. Muestre cómo modificar el procedimientos para crear e insertar en un árbol B para manejar esta variación.

Ejercicios 18.2-6

Suponga que B-TREE-SEARCH se implementa para utilizar la búsqueda binaria en lugar de la búsqueda lineal dentro de cada nodo. Muestre que este cambio hace que el tiempo de CPU requerido sea O (lg n), independientemente de cómo se podría elegir t *en* función de n .

Ejercicios 18.2-7

Página 384

Supongamos que el hardware del disco nos permite elegir el tamaño de una página de disco de forma arbitraria, pero que el el tiempo que se tarda en leer la página del disco es $a + bt$, donde a y b son constantes especificadas y t es la grado mínimo para un árbol B utilizando páginas del tamaño seleccionado. Describe cómo elegir t para que para minimizar (aproximadamente) el tiempo de búsqueda del árbol B. Sugerir un valor óptimo de t para el caso en el que a = 5 milisegundos yb = 10 microsegundos.

18.3 Eliminar una clave de un árbol B

La supresión de un árbol B es análoga a la inserción, pero un poco más complicada, porque una clave puede eliminarse de cualquier nodo, no solo una hoja, y la eliminación de un nodo interno requiere que los hijos del nodo se reorganizarán. Como en la inserción, debemos evitar que la eliminación produzca una árbol cuya estructura viola las propiedades del árbol B. Así como teníamos que asegurarnos de que un nodo no se vuelve demasiado grande debido a la inserción, debemos asegurarnos de que un nodo no se vuelva demasiado pequeño durante la eliminación (excepto que se permite que la raíz tenga menos del número mínimo t - 1 de claves, aunque no se permite tener más del número máximo 2 t - 1 de llaves). Tan simple el algoritmo de inserción podría tener que realizar una copia de seguridad si un nodo en la ruta hacia donde se iba a colocar la clave insertado estaba completo, un enfoque simple para la eliminación podría tener que realizar una copia de seguridad si un nodo (que no sea la raíz) a lo largo de la ruta hacia donde se eliminará la clave tiene el número mínimo de claves.

Suponga que se solicita al procedimiento B-TREE-DELETE que elimine la clave k del subárbol enraizado en x . Este procedimiento está estructurado para garantizar que siempre que se llame a B-TREE-DELETE recursivamente en un nodo x , el número de claves en x es al menos el grado mínimo t . Tenga en cuenta que esta condición requiere una clave más que el mínimo requerido por el árbol B habitual

condiciones, por lo que a veces una clave puede tener que moverse a un nodo hijo antes de la recursividad desciende a ese nino. Esta condición reforzada nos permite eliminar una clave del árbol en una pasada hacia abajo sin tener que "retroceder" (con una excepción, que explicaremos).
La siguiente especificación para la eliminación de un árbol B debe interpretarse con el entendiendo que si alguna vez sucede que el nodo raíz *x se* convierte en un nodo interno sin claves (esta situación puede ocurrir en los casos 2c y 3b, a continuación), entonces *x* se elimina y el único hijo de *x* $c_1[x]$ se convierte en la nueva raíz del árbol, disminuyendo la altura del árbol en uno y conservando la propiedad de que la raíz del árbol contiene al menos una clave (a menos que el árbol esté vacío).

Esbozamos cómo funciona la eliminación en lugar de presentar el pseudocódigo. La figura 18.8 ilustra los diversos casos de eliminación de claves de un árbol B.

Página 385

Figura 18.8: Eliminando claves de un árbol B. El grado mínimo para este árbol B es $t = 3$, por lo que un El nodo (que no sea la raíz) no puede tener menos de 2 claves. Los nodos que se modifican son ligeramente sombreado. *(a)* El árbol B de la figura 18.7 (e). *(b)* Supresión de *F* . Este es el caso 1: eliminación simple de una hoja. *(c)* Supresión de *M* . Este es el caso 2a: el predecesor *L* de *M* se mueve hacia arriba para tomar *M* s' posición. *(d)* Supresión de *G* . Este es el caso 2c: *G* se empuja hacia abajo para hacer el nodo *DEGJK* , y luego

G se elimina de esta hoja (caso 1). *(e)* Supresión de *D* . Este es el caso 3b: la recursividad no puede desciende al nodo *CL* porque tiene solo 2 claves, por lo que *P* se empuja hacia abajo y se fusiona con *CL* y *TX* para formar *CLPTX* ; luego, *D* se elimina de una hoja (caso 1). *(e ')* Después de (d), la raíz se elimina y el árbol se encoge en uno en altura. *(f)* Supresión de *B* . Este es el caso 3a: *C* se mueve para llenar *B* 's posición y *E* se mueve para ocupar la posición de *C.*

1. Si la clave *k* es decir en el nodo *x* y *x* es una hoja, elimine la clave *k* de *x* .
2. Si la clave *k* es decir en el nodo *x* y *x* es un nodo interno, haga lo siguiente.
    a. Si el hijo *y* que precede a *k* en el nodo *x* tiene al menos *t* claves, entonces encuentre el predecesor *k ′* de *k* en el subárbol con raíz en *y* . Eliminar *k ′* de forma recursiva y reemplazar *k* por *k ′* en *x* . (Encontrar *k ′* y borrarlo se puede realizar en una sola pasar.)

Página 386

si. Simétricamente, si el hijo *z* que sigue a *k* en el nodo *x* tiene al menos *t* claves, entonces encuentre el sucesor *k ′* de *k* en el subárbol con raíz en *z* . Suprima *k ′* de forma recursiva y reemplace *k* por *k ′* en *x* . (Encontrar *k ′* y borrarlo se puede realizar en un solo pase hacia abajo.)

C. De lo contrario, si tanto *y* como *z* tienen solo *t* - 1 claves, combine *k* y todo *z* en *y* , entonces que *x* pierde tanto *k* como el apuntador a *z* , e *y* ahora contiene 2 *t* - 1 claves. Luego, libera *z* y borra de forma recursiva *k* de *y* .

3. Si la clave *k* no está presente en el nodo interno *x* , determine la raíz $c_i[x]$ del subárbol que debe contener *k* , si *k* está en el árbol. Si $c_i[x]$ solo tiene *t* - 1 teclas, ejecute paso 3a o 3b según sea necesario para garantizar que descendemos a un nodo que contiene al menos *t* llaves. Luego, termine recurriendo al hijo apropiado de *x* .
    a. Si $c_i[x]$ solo tiene *t* - 1 claves pero tiene un hermano inmediato con al menos *t* claves, dale a $c_i[x]$ una tecla extra moviendo una tecla de *x* hacia abajo a $c_i[x]$, moviendo una tecla del hermano derecho o izquierdo inmediato de $c_i[x]$ hacia arriba en *x* , y moviendo el puntero secundario apropiado del hermano a $c_i[x]$.
    si. Si $c_i[x]$ y los dos hermanos inmediatos de $c_i[x]$ tienen *t* - 1 claves, combine $c_i[x]$ con un hermano, lo que implica mover una clave de *x* hacia abajo en la nueva combinación nodo para convertirse en la clave mediana para ese nodo.

Dado que la mayoría de las claves en un árbol B están en las hojas, podemos esperar que en la práctica, la eliminación Las operaciones se utilizan con mayor frecuencia para eliminar claves de hojas. El procedimiento B-TREE-DELETE luego actúa en una pasada descendente a través del árbol, sin tener que retroceder. Al eliminar un clave en un nodo interno, sin embargo, el procedimiento hace un pase hacia abajo a través del árbol pero Es posible que tenga que volver al nodo del que se eliminó la clave para reemplazar la clave con su predecesor o sucesor (casos 2a y 2b).

Aunque este procedimiento parece complicado, solo implica operaciones de disco $O(h)$ para un árbol B de altura *h* , ya que solo $O(1)$ llamadas a DISK-READ y DISK-WRITE se realizan entre invocaciones recursivas del procedimiento. El tiempo de CPU requerido es $O(th) = O(t \log_t n)$.

Ejercicios 18.3-1

Muestre los resultados de eliminar *C* , *P* y *V* , en orden, del árbol de la figura 18.8 (f) .

Ejercicios 18.3-2

Escriba un pseudocódigo para B-TREE-DELETE.

Problemas 18-1: pilas en almacenamiento secundario

Considere implementar una pila en una computadora que tenga una cantidad relativamente pequeña de memoria primaria y una cantidad relativamente grande de almacenamiento en disco más lento. Las operaciones PUSH

Página 387

y POP son compatibles con valores de una sola palabra. La pila que deseamos apoyar puede llegar a ser mucho más grande de lo que cabe en la memoria y, por lo tanto, la mayor parte debe almacenarse en el disco.

Una implementación de pila simple, pero ineficiente, mantiene toda la pila en el disco. Mantenemos en memory un puntero de pila, que es la dirección de disco del elemento superior de la pila. Si el puntero tiene valor p , el elemento superior es la (p mod m) th palabra en la página $\lfloor p / m \rfloor$ del disco, donde m es el número de palabras por página.

Para implementar la operación PUSH, incrementamos el puntero de la pila, leemos la página apropiada en la memoria desde el disco, copie el elemento que se enviará a la palabra correspondiente en la página, y vuelva a escribir la página en el disco. Una operación POP es similar. Disminuimos el puntero de la pila, lea en la página correspondiente del disco y devuelva la parte superior de la pila. No necesitamos escribir de nuevo la página, ya que no fue modificada.

Debido a que las operaciones de disco son relativamente caras, contamos con dos costos para cualquier implementación: el número total de accesos al disco y el tiempo total de CPU. Cualquier acceso de disco a una página de m palabras incurre en cargos de acceso a un disco y Θ (m) tiempo de CPU.

a. Asintóticamente, ¿cuál es el número de accesos al disco en el peor de los casos para n operaciones de pila? utilizando esta sencilla implementación? ¿Cuál es el tiempo de CPU para n operaciones de pila? (Exprese su respuesta en términos de m y n para esto y partes posteriores).

Ahora considere una implementación de pila en la que mantenemos una página de la pila en memoria. (También mantenemos una pequeña cantidad de memoria para realizar un seguimiento de la página que se encuentra actualmente en memoria.) Podemos realizar una operación de pila solo si la página del disco relevante reside en la memoria. Si es necesario, la página actualmente en la memoria se puede escribir en el disco y la nueva página se puede leer en del disco a la memoria. Si la página del disco relevante ya está en la memoria, entonces no hay disco Se requieren accesos.

si. ¿Cuál es el número de accesos al disco en el peor de los casos necesarios para n operaciones PUSH? ¿Cuál es el tiempo de la CPU?

C. ¿Cuál es el número de accesos al disco en el peor de los casos necesarios para n operaciones de pila? Qué es el tiempo de la CPU?

Supongamos que ahora implementamos la pila manteniendo dos páginas en la memoria (además de una pequeña cantidad de palabras para la contabilidad).

re. Describir cómo administrar las páginas de la pila para que el número amortizado de accesos al disco para cualquier operación de pila es O (1 / m) y el tiempo de CPU amortizado para cualquier operación de pila es O (1).

Problemas 18-2: Unir y dividir 2-3-4 árboles

La operación de unión toma dos conjuntos dinámicos S' y S'' y un elemento x tal que para cualquier $x' \in S'$ y $x'' \in S''$, tenemos la *clave* [x'] < *clave* [x] < *clave* [x'']. Devuelve un conjunto $S = S' \cup \{ x \} \cup S''$. La escisión La operación es como una combinación "inversa": dado un conjunto dinámico S y un elemento $x \in S$, crea un conjunto S' que consta de todos los elementos en $S - \{ x \}$ cuyas claves son menores que la *clave* [x] y un conjunto S'' que consta

Página 388

de todos los elementos en $S - \{ x \}$ cuyas claves son mayores que la *clave* [x]. En este problema, investigamos cómo implementar estas operaciones en 2-3-4 árboles. Asumimos por conveniencia que los elementos constan solo de claves y que todos los valores clave son distintos.

a. Muestre cómo mantener, para cada nodo x de un árbol 2-3-4, la altura del subárbol arraigado en x como una *altura de* campo [x]. Asegúrese de que su implementación no afecte al tiempos de ejecución asintóticos de búsqueda, inserción y eliminación.

si. Muestre cómo implementar la operación de unión. Dados dos 2-3-4 árboles T' y T'' y una clave k, la unión debe ejecutarse en $O(1 + |h' - h''|)$ tiempo, donde h' y h'' son las alturas de T' y T'', respectivamente.

C. Considere la ruta p desde la raíz de un árbol 2-3-4 T a una clave dada k, el conjunto S' de claves en T que son menores que k, y el conjunto S'' de claves en T que son mayores que k. Muestre que p rompe S' en un conjunto de árboles y un juego de llaves dónde, para $i = 1, 2, ..., m$, tenemos para cualquier llave y . Cuál es el relación entre las alturas de y? Describe cómo p divide S'' en conjuntos de arboles y llaves.

re. Mostrar cómo implementar la operación de división en T. Utilice la operación de unión para ensamblar las claves en S' en un solo árbol 2-3-4 T' y las claves en S'' en un solo árbol 2-3-4 T''. El tiempo de ejecución de la operación de división debe ser $O(\lg n)$, donde n es el número de llaves en T. (*Sugerencia:* los costos de unión deberían aumentar).

Notas del capítulo

Knuth [185] , Aho, Hopcroft y Ullman [5] y Sedgewick [269] dan más discusiones sobre esquemas de árboles balanceados y árboles B. Comer [66] proporciona un estudio completo de árboles B. Guibas y Sedgewick [135] discuten las relaciones entre varios tipos de árboles balanceados esquemas, incluidos árboles rojo-negro y 2-3-4 árboles.

En 1970, JE Hopcroft inventó 2-3 árboles, un precursor de los árboles B y 2-3-4 árboles, en los que cada nodo interno tiene dos o tres hijos. Los árboles B fueron introducidos por Bayer y McCreight en 1972 [32] ; no explicaron su elección de nombre.

Bender, Demaine y Farach-Colton [37] estudiaron cómo hacer que los árboles B funcionen bien en el presencia de efectos de jerarquía de memoria. Sus algoritmos ajenos al caché funcionan de manera eficiente sin conocer explícitamente los tamaños de transferencia de datos dentro de la jerarquía de memoria.

Capítulo 19: Montones binomiales

Visión general

Este capítulo y el capítulo 20 presentan estructuras de datos conocidas como montones fusionables , que admite las siguientes cinco operaciones.

- MAKE-HEAP () crea y devuelve un nuevo montón que no contiene elementos.
- INSERT (H , X) insertos nodo x , cuya *clave* campo ya ha sido rellenado, en montón H .
- MINIMUM (H) devuelve un puntero al nodo en el montón H cuya clave es mínima.
- EXTRACT-MIN (H) elimina el nodo del montón H cuya clave es mínima, devolviendo un puntero al nodo.

- UNION (H_1 , H_2) crea y devuelve un nuevo montón que contiene todos los nodos de los montones H_1 y H_2 . Los montones H_1 y H_2 son "destruidos" por esta operación.

Además, las estructuras de datos de estos capítulos también admiten las dos operaciones siguientes.

- DECREASE-KEY (H , x , k) asigna al nodo x dentro del montón H el nuevo valor de clave k , que se supone que no es mayor que su valor clave actual. [1]
- BORRAR (H , X) Elimina nodo x del montón H .

Como muestra la tabla de la Figura 19.1 , si no necesitamos la operación UNION, el binario ordinario montones, como se usa en heapsort (Capítulo 6), funcionan bien. Las operaciones distintas de UNION se ejecutan en el peor caso de tiempo $O(\lg n)$ (o mejor) en un montón binario. Si la operación UNION debe ser compatible, sin embargo, los montones binarios funcionan mal. Concatenando las dos matrices que contienen el binario montones para fusionar y luego ejecutar MIN-HEAPIFY (ver Ejercicio 6.2-2), la UNION la operación lleva $\Theta(n)$ tiempo en el peor de los casos.

Procedimiento	Montón binario (peor caso)	Montón binomial (peor caso)	Montón de Fibonacci (amortizado)
HACER HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

INSERTAR	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MÍNIMO	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACTO-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNIÓN	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DISMINUCIÓN-LLAVE	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
ELIMINAR	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Figura 19.1: Tiempos de ejecución para operaciones en tres implementaciones de montones fusionables. los el número de elementos en el montón (s) en el momento de una operación se indica con n .

En este capítulo, examinamos "montones binomiales", cuyos límites de tiempo en el peor de los casos también se muestran en la Figura 19.1 . En particular, la operación UNION toma solo $O(\lg n)$ tiempo para fusionar dos montones binomiales con un total de n elementos.

En el capítulo 20 , exploraremos los montones de Fibonacci, que tienen límites de tiempo aún mejores para algunas operaciones. Sin embargo, tenga en cuenta que los tiempos de ejecución de los montones de Fibonacci en la figura 19.1 son límites de tiempo amortizados, no límites de tiempo por operación en el peor de los casos.

Este capítulo ignora los problemas de la asignación de nodos antes de la inserción y la liberación de nodos después supresión. Suponemos que el código que llama a los procedimientos de montón se ocupa de estos detalles.

Montones binarios, montones binomiales y montones de Fibonacci son todos ineficientes en su apoyo de la operación BUSCAR; puede llevar un tiempo encontrar un nodo con una clave determinada. Por esta razón, operaciones como DECREASE-KEY y DELETE que se refieren a un nodo dado requieren un puntero a ese nodo como parte de su entrada. Como en nuestra discusión sobre colas de prioridad en la Sección

Página 390

6.5 , cuando usamos un montón fusionable en una aplicación, a menudo almacenamos un identificador en el objeto de aplicación correspondiente en cada elemento de pila combinable, así como un identificador para correspondiente elemento mergeable-heap en cada objeto de aplicación. La naturaleza exacta de estos maneja depende de la aplicación y su implementación.

La sección 19.1 define los montones binomiales después de definir primero sus árboles binomiales constituyentes. También introduce una representación particular de montones binomiales. La sección 19.2 muestra cómo podemos Implementar operaciones en montones binomiales en los límites de tiempo dados en la Figura 19.1 .

[1] Como se mencionó en la introducción de la Parte V , nuestros montones fusionables predeterminados son min-montones, por lo que se aplican las operaciones MINIMUM, EXTRACT-MIN y DECREASE-KEY. Alternativamente, podríamos definir un *montón máximo fusionable* con las operaciones MAXIMUM, EXTRACTO-MAX y AUMENTO-CLAVE.

19.1 Árboles binomiales y montones binomiales

Un montón binomial es una colección de árboles binomiales, por lo que esta sección comienza definiendo binomial árboles y demostrando algunas propiedades clave. Luego definimos los montones binomiales y mostramos cómo se puede representar.

19.1.1 Árboles binomiales

El *árbol binomial* B_k es un árbol ordenado (consulte la Sección B.5.2) definido de forma recursiva. Como se muestra en En la figura 19.2 (a) , el árbol binomial B_0 consta de un solo nodo. El árbol binomial B_k consta de dos árboles binomiales B_{k-1} que están *vinculados entre* sí: la raíz de uno es el hijo más a la izquierda de la raíz del otro. La figura 19.2 (b) muestra los árboles binomiales B_0 a B_4 .

Figura 19.2: *(a)* La definición recursiva del árbol binomial B_k . Los triángulos representan arraigados subárboles. *(b)* Los árboles binomiales B_0 a B_4 . Se muestran las profundidades de los nodos en B_4 . *(c)* Otra forma de mirar el árbol binomial B_k .

Algunas propiedades de los árboles binomiales vienen dadas por el siguiente lema.

Página 391

Lema 19.1: (Propiedades de los árboles binomiales)

Para el árbol binomial B_k ,

1. hay 2_k nodos,
2. la altura del árbol es k ,
3. hay exactamente nodos en la profundidad i para i = 0, 1, ..., k , y
4. la raíz tiene grado k , que es mayor que el de cualquier otro nodo; además si *yo* el los hijos de la raíz se numeran de izquierda a derecha por k - 1, k - 2, ..., 0, el hijo i es el raíz de un subárbol B_i .

Prueba La prueba es por inducción en k . Para cada propiedad, la base es el árbol binomial B_0 . Verificar que cada propiedad sea válida para B_0 es trivial.

Para el paso inductivo, asumimos que el lema es válido para B_{k-1} .

1. El árbol binomial B_k consta de dos copias de B_{k-1} , por lo que Bk tiene $2_{k-1} + 2_{k-1} = 2_k$ nodos.
2. Debido a la forma en que las dos copias de B_{k-1} están vinculadas para formar B_k , la la profundidad máxima de un nodo en B_k es uno mayor que la profundidad máxima en B_{k-1} . Por el hipótesis inductiva, esta profundidad máxima es (k - 1) + 1 = k .
3. Sea D (k , i) el número de nodos a la profundidad i del árbol binomial B_k . Dado que B_k está compuesto de dos copias de B_{k-1} enlazadas, un nodo a la profundidad i en B_{k-1} aparece en B_k una vez en profundidad i y una vez a profundidad i + 1. En otras palabras, el número de nodos a profundidad i en B_k es el número de nodos a la profundidad i en B_{k-1} más el número de nodos a la profundidad i - 1 en B_{k-1} . Así,

4. El único nodo con mayor grado en B_k que en B_{k-1} es la raíz, que tiene uno más niño que en B_{k-1} . Dado que la raíz de B_{k-1} tiene grado k - 1, la raíz de B_k tiene grado k . Ahora, por la hipótesis inductiva, y como muestra la Figura 19.2 (c) , de izquierda a derecha, la los hijos de la raíz de B_{k-1} son raíces de B_{k-2} , B_{k-3} , ..., B_0 . Cuando B_{k-1} está vinculado a B_{k-1} , por lo tanto, los hijos de la raíz resultante son raíces de B_{k-1} , B_{k-2} , ..., B_0 .

Corolario 19.2

El grado máximo de cualquier nodo en un árbol binomial de n nodos es lg n .

Prueba Inmediata de las propiedades 1 y 4 del Lema 19.1 .

Página 392

El término "árbol binomial" proviene de la propiedad 3 del Lema 19.1 , ya que los términos son los coeficientes binomiales. El ejercicio 19.1-3 proporciona una mayor justificación del término.

19.1.2 Montones binomiales

Un montón binomial *H* es un conjunto de árboles binomiales que satisface el siguiente montón binomial propiedades .

1. Cada árbol binomial en *H* obedece a la propiedad min-heap : la clave de un nodo es mayor que o igual a la clave de su padre. Decimos que cada uno de estos árboles está ordenado por min-montones .
2. Para cualquier entero no negativo *k* , hay como máximo un árbol binomial en *H* cuya raíz tiene grado *k* .

La primera propiedad nos dice que la raíz de un árbol ordenado en min-heap contiene la clave más pequeña en el árbol.

La segunda propiedad implica que un *n* -node binomial montón *H* se compone de a lo sumo ⌊lg *n* ⌋ + 1 árboles binomiales. Para ver por qué, observe que la representación binaria de *n* tiene ⌊lg *n* ⌋ + 1 bits, digamos $b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \ldots, b_0$, así que eso . Por la propiedad 1 del Lema 19.1 , por lo tanto, El árbol binomial B_i aparece en *H* si y solo si el bit $b_i = 1$. Por lo tanto, el montón binomial *H* contiene como máximo ⌊Lg *n* ⌋ + 1 árboles binomiales.

La figura 19.3 (a) muestra un montón binomial *H* con 13 nodos. La representación binaria de 13 es 1101, y *H* consta de árboles binomiales B_3 , B_2 y B_0 con orden mínimo de montón , que tienen 8, 4 y 1 nodos respectivamente, para un total de 13 nodos.

Figura 19.3: Un montón binomial *H* con *n* = 13 nodos. *(a)* El montón consta de árboles binomiales B_0 , B_2 y B_3 , que tienen 1, 4 y 8 nodos respectivamente, totalizando *n* = 13 nodos. Desde cada uno El árbol binomial tiene un orden mínimo de pila, la clave de cualquier nodo no es menor que la clave de su padre. También se muestra la lista de raíces, que es una lista vinculada de raíces en orden creciente. *(b)* A más detallado representación de binomial montón *H* . Cada árbol binomial se almacena en la izquierda-representación del niño, del hermano derecho, y cada nodo almacena su grado.

Representando montones binomiales

Página 393

Como se muestra en la Figura 19.3 (b) , cada árbol binomial dentro de un montón binomial se almacena en el lado izquierdo. representación de niño, hermano derecho de la Sección 10.4 . Cada nodo tiene un campo *clave* y cualquier otro información satelital requerida por la aplicación. Además, cada nodo *x* contiene punteros *p* [*x*] a su padre, *hijo* [*x*] a su hijo más a la izquierda, y *hermano* [*x*] al hermano de *x* inmediatamente a es lo correcto. Si el nodo *x* es una raíz, entonces *p* [*x*] = NULO. Si el nodo *x* no tiene hijos, entonces *hijo* [*x*] = NIL,

y si x es el hijo situado más a la derecha de su padre, entonces *hermano* [x] = NIL. Cada nodo x también contiene el campo *grado* [x], que es el número de hijos de x .

Como también muestra la figura 19.3 , las raíces de los árboles binomiales dentro de un montón binomial son organizados en una lista vinculada, a la que nos referimos como la lista raíz . Los grados de las raíces estrictamente aumenta a medida que recorremos la lista raíz. Por la segunda propiedad binomial-heap, en un n -nodo

montón binomial los grados de las raíces son un subconjunto de {0, 1, ..., ⌊lg n ⌋}. El campo *hermano* tiene un significado diferente para raíces que para no raíces. Si x es una raíz, el *hermano* [x] apunta a la siguiente root en la lista raíz. (Como de costumbre, *hermano* [x] = NULO si x es la última raíz en la lista de raíces).

Se accede a un montón binomial H dado mediante el *encabezado de* campo [H], que es simplemente un puntero al primera raíz en la lista raíz de H . Si el montón binomial H no tiene elementos, entonces *cabecera* [H] = NIL.

Ejercicios 19.1-1

Suponga que x es un nodo en un árbol binomial dentro de un montón binomial y suponga que *hermano* [x] ≠ NULO. Si x no es una raíz, ¿cómo se compara el *grado* [*hermano* [x]] con el *grado* [x]? ¿Qué tal si x es ¿una raíz?

Ejercicios 19.1-2

Si x es un nodo no raíz en un árbol binomial dentro de un montón binomial, ¿cómo se compara el *grado* [x] al *grado* [p [x]]?

Ejercicios 19.1-3

Suponga que etiquetamos los nodos del árbol binomial B_k en binario mediante una caminata posterior, como en la Figura 19.4 . Considere un nodo x etiquetado l en profundidad i , y sea $j = k - i$. Demuestre que x tiene j 1 en su binario representación. ¿Cuántas k- cadenas binarias hay que contienen exactamente j 1? Muestra que el El grado de x es igual al número de unos a la derecha del 0 situado más a la derecha en el binario. representación de l .

Página 394

Figura 19.4: El árbol binomial B_4 con nodos etiquetados en binario por una caminata postorder.

19.2 Operaciones en montones binomiales

En esta sección, mostramos cómo realizar operaciones en montones binomiales en los límites de tiempo se muestra en la Figura 19.1 . Solo mostraremos los límites superiores; los límites inferiores se dejan como Ejercicio 19.2-10 .

Creando un nuevo montón binomial

Para hacer un montón binomial vacío, el procedimiento MAKE-BINOMIAL-HEAP simplemente asigna y devuelve un objeto H , donde *head* [H] = NIL. El tiempo de ejecución es Θ (1).

Encontrar la clave mínima

El procedimiento BINOMIAL-HEAP-MINIMUM devuelve un puntero al nodo con el clave mínimo en un n -node binomial montón H . Esta implementación asume que no hay claves con valor ∞. (Vea el ejercicio 19.2-5 .)

```
BINOMIAL-HEAP-MINIMUM ( H )
1 y ← NIL
2 x ← cabeza [ H ]
3 min ← ∞
4 mientras que x ≠ NIL
5 hacer si la tecla [ x ] < min
6                    luego min ← tecla [ x ]
7                          y ← x
8               x ← hermano [ x ]
9 volver y
```

Dado que un montón binomial tiene un orden mínimo de montón, la clave mínima debe residir en un nodo raíz. los

El procedimiento BINOMIAL-HEAP-MINIMUM comprueba todas las raíces, cuyo número como máximo⌊lg n ⌋ + 1, guardando el mínimo actual en *min* y un puntero al mínimo actual en y . Cuando llamado el montón binomial de la Figura 19.3 , BINOMIAL-HEAP-MINIMUM devuelve un puntero al nodo con clave 1.

Debido a que hay como máximo ⌊lg n ⌋ + 1 raíces para verificar, el tiempo de ejecución de BINOMIAL-HEAP-MÍNIMO es O (lg n).

Uniendo dos montones binomiales

La operación de unir dos montones binomiales es utilizada como subrutina por la mayoría de los restantes operaciones. El procedimiento BINOMIAL-HEAP-UNION vincula repetidamente árboles binomiales cuyas las raíces tienen el mismo grado. El siguiente procedimiento vincula el árbol B_{k-1} enraizado en el nodo y al

Página 395

B_{k-1} árbol enraizado en el nodo z ; es decir, hace que z sea el padre de y . El nodo z se convierte así en la raíz de un B_k árbol.

```
ENLACE BINOMIAL ( y , z )
1 p [ y ] ← z
2 hermano [ y ] ← hijo [ z ]
3 niño [ z ] ← y
4 grados [ z ] ← grado [ z ] + 1
```

El procedimiento BINOMIAL-LINK marcas nodo y la nueva cabeza de la lista enlazada de nodos z 's niños en el tiempo O (1). Funciona porque la representación del hijo izquierdo y del hermano derecho de cada árbol binomial coincide con la propiedad de ordenación del árbol: en un árbol B_k , el hijo más a la izquierda del árbol root es la raíz de un árbol B_{k-1} .

El siguiente procedimiento une los montones binomiales H_1 y H_2 , devolviendo el montón resultante. Eso destruye las representaciones de H_1 y H_2 en el proceso. Además de BINOMIAL-LINK, el El procedimiento utiliza un procedimiento auxiliar BINOMIAL-HEAP-MERGE que fusiona las listas raíz de H_1 y H_2 en una sola lista enlazada que se ordena por grado en monotónicamente creciente orden. El procedimiento BINOMIAL-HEAP-MERGE, cuyo pseudocódigo dejamos como Ejercicio 19.2-1 , es similar al procedimiento MERGE de la Sección 2.3.1 .

```
UNIÓN-HEAP-BINOMIAL ( H 1 , H 2 )
 1 H ← HACER-BINOMIAL-HEAP ()
 2 cabezales [ H ] ← BINOMIAL-HEAP-MERGE ( H 1 , H 2 )
 3 libera los objetos H 1 y H 2 pero no las listas a las que apuntan
 4 si cabeza [ H ] = NIL
 5 luego regrese H
 6 anterior - x ← NULO
 7 x ← cabeza [ H ]
 8 siguiente - x ← hermano [ x ]
 9 mientras sigue - x ≠ NIL
10 hacer si ( grado [ x ] ≠ grado [ siguiente - x ]) o
                    ( hermano [ siguiente - x ] ≠ NULO y grado [ hermano [ siguiente - x ]] =
grado [ x ])
11                  luego prev - x ← x                                  ▹ Casos 1 y
2
12                        x ← siguiente - x                             ▹ Casos 1 y
2
```

```
13          si no, tecla [ x ] ≤ tecla [ siguiente - x ]
14              luego hermano [ x ] ← hermano [ siguiente - x ]                ▸ Caso 3
15                  ENLACE BINOMIAL ( siguiente - x , x )                      ▸ Caso 3
dieciséis           de lo contrario, si prev - x = NIL                         ▸ Caso 4
17                      luego dirígete [ H ] ← siguiente - x                   ▸ Caso 4
18                      más hermano [ anterior - x ] ← siguiente - x           ▸ Caso 4
19                  ENLACE BINOMIAL ( x , siguiente - x )                      ▸ Caso 4
20                  x ← siguiente - x                                          ▸ Caso 4
21      siguiente - x ← hermano [ x ]
22 regreso H
```

La figura 19.5 muestra un ejemplo de BINOMIAL-HEAP-UNION en el que los cuatro casos dados en se produce el pseudocódigo.

Página 396

Figura 19.5: La ejecución de BINOMIAL-HEAP-UNION. *(a)* Montones binomiales H_1 y H_2 . *(si)* El montón binomial H es la salida de BINOMIAL-HEAP-MERGE (H_1 , H_2). Inicialmente, x es la primera raíz en la lista de raíz H . Debido a que tanto x como *next-x* tienen grado 0 y la *clave* [x] < *key* [*next-x*], se aplica el caso 3. *(c)* Después de que ocurre el vínculo, x es la primera de tres raíces con el mismo grado, entonces se aplica el caso 2. *(d)* Después de que todos los punteros bajen una posición en la lista raíz, caso 4 se aplica, ya que x es la primera de dos raíces de igual grado. *(e)* Después de que se produce el vínculo, caso 3 aplica. *(f)* Después de otro enlace, se aplica el caso 1, porque x tiene grado 3 y *next-x* tiene grado 4. Esta iteración del *tiempo* de bucle es la última, ya que después de los punteros se mueven una posición hacia abajo en la lista raíz, *next-x* = NIL.

El procedimiento BINOMIAL-HEAP-UNION tiene dos fases. La primera fase, realizada por el llamada de BINOMIAL-HEAP-MERGE, fusiona las listas raíz de los montones binomiales H_1 y H_2 en un lista única enlazada H que está ordenada por grado en orden monotónicamente creciente. No podría ser hasta dos raíces (pero no más) de cada grado, sin embargo, por lo que la segunda fase enlaza raíces de igual grado hasta que como máximo queda una raíz de cada grado. Porque la lista enlazada H está ordenada por grado, podemos realizar todas las operaciones de enlace rápidamente.

En detalle, el procedimiento funciona de la siguiente manera. Las líneas 1-3 comienzan fusionando las listas de raíces de binomio

montones H_1 y H_2 en una sola lista de raíz H . Las listas de raíces de H_1 y H_2 se ordenan estrictamente

Página 397

grado creciente, y BINOMIAL-HEAP-MERGE devuelve una lista raíz H que está ordenada por grado monótonamente creciente. Si las listas de raíces de H_1 y H_2 tienen m raíces en total, BINOMIAL-HEAP-MERGE se ejecuta en tiempo $O(m)$ examinando repetidamente las raíces en el encabezados de las dos listas raíz y agregando la raíz con el grado más bajo a la lista raíz de salida, eliminándolo de su lista raíz de entrada en el proceso.

A continuación, el procedimiento BINOMIAL-HEAP-UNION inicializa algunos punteros en la lista raíz de H . Primero, simplemente regresa en las líneas 4-5 si está uniendo dos montones binomiales vacíos. De la línea 6 en adelante, por lo tanto, sabemos que H tiene al menos una raíz. Durante todo el procedimiento, mantenga tres punteros en la lista raíz:

- x apunta a la raíz que se está examinando actualmente,
- *prev* - x apunta a la raíz que precede a x en la lista raíz: *hermano* [*anterior* - x] = x (desde inicialmente x no tiene predecesor, comenzamos con *prev* - x establecido en NIL), y
- *siguiente* - x apunta a la raíz que sigue a x en la lista de raíces: *hermano* [x] = *siguiente* - x .

Inicialmente, hay como máximo dos raíces en la lista de raíces H de un grado dado: porque H_1 y H_2 eran montones binomiales, cada uno tenía como máximo una raíz de un grado dado. Además, BINOMIAL-HEAP-MERGE nos garantiza que si dos raíces en H tienen el mismo grado, son adyacentes en la lista raíz.

De hecho, durante la ejecución de BINOMIAL-HEAP-UNION, puede haber tres raíces de un grado dado que aparece en la lista raíz H en algún momento. Veremos en un momento cómo esto situación podría ocurrir. En cada iteración del **tiempo** de bucle de las líneas 9-21, por lo tanto, decidimos si se vinculará x y *al lado* - x en función de sus grados y, posiblemente, el grado de *hermanos* [*próxima* - x]. Un invariante del ciclo es que cada vez que iniciamos el cuerpo del ciclo, tanto x como *siguiente* - x son no NIL. (Consulte el ejercicio 19.2-4 para obtener un invariante de bucle preciso).

El caso 1, que se muestra en la Figura 19.6 (a) , ocurre cuando *grado* [x] $\neq$ *grado* [*siguiente* - x], es decir, cuando x es la raíz de un árbol B_k y el *siguiente* - x es la raíz de un árbol B_l para algunos $l > k$. Las líneas 11-12 manejan esto caso. No vinculamos x y *siguiente* - x , por lo que simplemente desplazamos los punteros una posición más hacia abajo la lista. Actualizar *siguiente* - x para apuntar al nodo que sigue al nuevo nodo x se maneja en la línea 21, que es común a todos los casos.

Figura 19.6: Los cuatro casos que ocurren en BINOMIAL-HEAP-UNION. Etiquetas *a, b, c* y *d* sirven solo para identificar las raíces involucradas; no indican los grados ni claves de estos

Página 398

raíces. En cada caso, x es la raíz de un árbol B_k y $l > k$. *(a)* Caso 1: *grado* [x] $\neq$ *grado* [*siguiente-x*]. Los punteros se mueven una posición más abajo en la lista raíz. *(b)* Caso 2: *grado* [x] = *grado* [*siguiente-x*] = *grado* [*hermano* [*siguiente-x*]]. Nuevamente, los punteros se mueven una posición más hacia abajo.

la lista, y la siguiente iteración ejecuta el caso 3 o el caso 4. *(c)* Caso 3: *grado* [*x*] = *grado* [*siguiente-x*] ≠ *grado* [*hermano* [*siguiente-x*]] y *tecla* [*x*] ≤ *tecla* [*siguiente-x*]. Eliminamos *next-x* del lista raíz y vincularla *ax* , creando un árbol B_{k+1} . *(d)* Caso 4: *grado* [*x*] = *grado* [*siguiente-x*] ≠ *grado* [*hermano* [*siguiente-x*]] y *clave* [*siguiente-x*] ≤ *clave* [*x*]. Eliminamos *x* de la lista raíz y lo vinculamos a *next-x* , de nuevo creando un árbol B_{k+1} .

El caso 2, que se muestra en la figura 19.6 (b) , ocurre cuando *x* es la primera de tres raíces de igual grado, que es cuando

grado [*x*] = *grado* [*siguiente* - *x*] = *grado* [*hermano* [*siguiente* - *x*]].

Manejamos este caso de la misma manera que el caso 1: solo hacemos marchar los punteros una posición más abajo en la lista. La siguiente iteración ejecutará el caso 3 o el caso 4 para combinar el segunda y tercera de las tres raíces de igual grado. Pruebas de línea 10 para los casos 1 y 2, y líneas 11-12 manejan ambos casos.

Los casos 3 y 4 ocurren cuando *x* es la primera de dos raíces de igual grado, es decir, cuando

grado [*x*] = *grado* [*siguiente* - *x*] ≠ *grado* [*hermano* [*siguiente* - *x*]].

Estos casos pueden ocurrir en cualquier iteración, pero uno de ellos siempre ocurre inmediatamente después caso 2. En los casos 3 y 4, vinculamos *x* y *siguiente* - *x* . Los dos casos se distinguen por si *x* o *siguiente* - *x* tiene la clave más pequeña, que determina el nodo que será la raíz después de que los dos sean vinculado.

En el caso 3, que se muestra en la Figura 19.6 (c) , *tecla* [*x*] ≤ *tecla* [*siguiente* - *x*], por lo que *siguiente* - *x* está vinculado a *x* . Línea 14 elimina *next* - *x* de la lista raíz, y la línea 15 convierte a *next* - *x en* el hijo más a la izquierda de *x* .

En el caso 4, que se muestra en la Figura 19.6 (d) , *next* - *x* tiene la clave más pequeña, por lo que *x* está vinculado a *next* - *x* . Líneas 16-18 eliminar *x* de la lista raíz; hay dos casos dependiendo de si *x* es la primera raíz en la lista (línea 17) o no (línea 18). La línea 19 luego hace que *x* sea el hijo más a la izquierda de la *siguiente* - *x* , y la línea 20 actualiza *x* para la siguiente iteración.

Siguiendo cualquiera de los casos 3 o 4 caso, la configuración para la siguiente iteración del **tiempo** de bucle es la misma. Acabamos de unir dos árboles B_k para formar un árbol B_{k+1} , al que ahora apunta *x* . Había ya cero, uno o dos otros B_{k+1} -árboles en la lista raíz resultante de BINOMIAL-HEAP-FUSION, por lo que *x* es ahora el primero de uno, dos o tres árboles B_{k+1} en la lista raíz. Si *x* es el solo uno, luego ingresamos el caso 1 en la siguiente iteración: *grado* [*x*] ≠ *grado* [*siguiente* - *x*]. Si *x* es el primero de dos, entonces ingresamos el caso 3 o el caso 4 en la siguiente iteración. Es cuando *x* es el primero de tres que ingresamos al caso 2 en la siguiente iteración.

El tiempo de ejecución de BINOMIAL-HEAP-UNION es $O(\lg n)$, donde *n* es el número total de nodos en montones binomiales H_1 y H_2 . Podemos ver esto de la siguiente manera. Deje que H_1 contenga n_1 nodos y H_2 contiene n_2 nodos, de modo que $n = n_1 + n_2$. Entonces H_1 contiene como máximo $\lfloor \lg n_1 \rfloor + 1$ raíces y H_2 contiene como máximo $\lfloor \lg n_2 \rfloor + 1$ raíces, por lo que H contiene como máximo $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \le 2\lfloor \lg n \rfloor + 2 = O(\lg n)$ raíces inmediatamente después de la llamada de BINOMIAL-HEAP-MERGE. El momento de actuar

Página 399

BINOMIAL-HEAP-MERGE es entonces $O(\lg n)$. Cada iteración del **tiempo** de bucle toma $O(1)$ el tiempo, y hay como máximo $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ iteraciones porque cada iteración avanza el apunta una posición hacia abajo en la lista raíz de H o elimina una raíz de la lista raíz. El total el tiempo es entonces $O(\lg n)$.

Insertar un nodo

El siguiente procedimiento inserta el nodo *x* en el montón binomial H , asumiendo que *x* ya ha asignado y la *clave* [*x*] ya se ha completado.

```
INSERTO BINOMIAL-HEAP ( H , x )
1 H' ← HACER-BINOMIAL-HEAP ()
2 p [ x ] ← NULO
3 niño [ x ] ← NULO
4 hermano [ x ] ← NULO
5 grados [ x ] ← 0
6 cabezas [ H' ] ← x
7 H ← BINOMIAL-HEAP-UNION ( H , H' )
```

El procedimiento simplemente crea un montón binomial de un nodo H' en tiempo $O(1)$ y lo une con el n- nodo binomial montón H en el tiempo $O(\lg n)$. La llamada a BINOMIAL-HEAP-UNION se encarga de liberando el montón binomial temporal H'. (Una implementación directa que no llama BINOMIAL-HEAP-UNION se da como ejercicio 19.2-8 .)

Extrayendo el nodo con clave mínima

El siguiente procedimiento extrae el nodo con la clave mínima del montón binomial H y devuelve un puntero al nodo extraído.

```
BINOMIAL-HEAP-EXTRACT-MIN ( H )
1 encuentre la raíz x con la clave mínima en la lista raíz de H ,
                y eliminar x de la lista raíz de H
2 H' ← HACER-BINOMIAL-HEAP ()
3 invierta el orden de la lista enlazada de hijos de x ,
                y coloque head [ H' ] para apuntar al encabezado de la lista resultante
4 H ← BINOMIAL-HEAP-UNION ( H , H' )
5 devuelve x
```

Este procedimiento funciona como se muestra en la Figura 19.7 . El montón binomial de entrada H se muestra en la Figura 19,7 (a) . La figura 19.7 (b) muestra la situación después de la línea 1: la raíz x con la clave mínima tiene sido eliminado de la lista de raíz de H . Si x es la raíz de un árbol B_k , entonces por la propiedad 4 de El lema 19.1 , los hijos de x , de izquierda a derecha, son raíces de árboles B_{k-1}, B_{k-2}, ..., B_0. Figura 19.7 (c) muestra que al invertir la lista de hijos de x en la línea 3, tenemos un montón binomial H' que contiene todos los nodos x árbol 's excepto para x sí mismo. Debido a que el árbol de x se eliminó de H en línea 1, el montón binomial que resulta de unir H y H' en la línea 4, que se muestra en la Figura 19.7 (d) , contiene todos los nodos originalmente en H excepto x . Finalmente, la línea 5 devuelve x .

Página 400

Figura 19.7: La acción de BINOMIAL-HEAP-EXTRACT-MIN. *(a)* A binomial montón H . *(si)* La raíz x con mínima de clave se elimina de la lista de raíz de H . *(c)* La lista enlazada de x 's children se invierte, dando otro binomio H'. *(d)* El resultado de unir H y H' .

Dado que cada una de las líneas 1-4 toma $O(\lg n)$ tiempo si H tiene n nodos, BINOMIAL-HEAP-EXTRACT-MIN se ejecuta en tiempo $O(\lg n)$.

Disminuir una clave

El siguiente procedimiento reduce la clave de un nodo x en un montón binomial H a un nuevo valor k . Señala un error si k es mayor que la clave actual de x .

```
TECLA BINOMIAL-HEAP-DISMINUCIÓN ( H , x , k )
 1 si k > tecla [ x ]
 2 entonces el error "la clave nueva es mayor que la clave actual"
 3 tecla [ x ] ← k
 4 y ← x
```

```
 5 z ← p [ y ]
 6 mientras z ≠ NIL y la tecla [ y ] < tecla [ z ]
 7 hacen intercambio clave [ y ] ↔ tecla [ z ]
 8              ▸ Si Y y Z tienen campos satélite, cambio de ellos, también.
 9              y ← z
10              z ← p [ y ]
```

Como se muestra en la figura 19.8 , este procedimiento disminuye una clave de la misma manera que en un binario. min-heap: "burbujeando" la clave en el montón. Después de asegurarse de que la nueva clave no mayor que la clave actual y luego asignando la nueva clave *ax* , el procedimiento sube el árbol, con *y* apuntando inicialmente al nodo *x* . En cada iteración del **tiempo** de bucle de líneas 6-10, *clave* [*y*] se compara con la tecla de *y* padres 's *z* . Si *y* es la raíz o la *clave* [*y*] ≥ *clave* [*z*], el árbol binomial es ahora min-heap-order. De lo contrario, el nodo *y* viola el orden de min-heap, por lo que su clave es intercambiado con la clave de su padre *z* , junto con cualquier otra información de satélite. los luego, el procedimiento establece *y* en *z* , subiendo un nivel en el árbol y continúa con la siguiente iteración.

Página 401

Figura 19.8: La acción de BINOMIAL-HEAP-DECREASE-KEY. *(a)* La situación simplemente antes de la línea 6 de la primera iteración de la *mientras* bucle. Se ha reducido la clave del nodo *y* a 7, que es menor que la clave de *Y* los padres 's *z* . *(b)* Se intercambian las claves de los dos nodos, y Se muestra la situación justo antes de la línea 6 de la segunda iteración. Punteros *y* y *z* se han movido sube un nivel en el árbol, pero todavía se infringe el orden de pila mínima. *(c)* Después de otro intercambio y mover los punteros *y* y *z* hasta un nivel más, nos encontramos con que fin-min heap está satisfecho, por lo que la *mientras que el* ciclo termina.

El procedimiento BINOMIAL-HEAP-DECREASE-KEY lleva $O(\lg n)$ tiempo. Por propiedad 2 de Lema 19.1 , la profundidad máxima de *x* es $\lfloor \lg n \rfloor$, por lo **que el** ciclo while de las líneas 6-10 itera en la mayoría de $\lfloor \lg n \rfloor$ veces.

Eliminar una clave

Es fácil eliminar la clave de un nodo *x* y la información del satélite del montón binomial *H* en $O(\lg n)$ hora. La siguiente implementación asume que ningún nodo actualmente en el montón binomial tiene una clave de -∞.

```
BINOMIAL-HEAP-DELETE ( H , x )
1 TECLA BINOMIAL-HEAP-DISMINUCIÓN ( H , x , -∞)
2 BINOMIAL-HEAP-EXTRACT-MIN ( H )
```

El procedimiento BINOMIAL-HEAP-DELETE hace que el nodo *x* tenga la clave mínima única en el montón binomial completo dándole una clave de -∞. (El ejercicio 19.2-6 trata con la situación en que -∞ no puede aparecer como una clave, ni siquiera temporalmente.) Luego burbujea esta clave y el información satelital asociada hasta una raíz llamando a BINOMIAL-HEAP-DECREASE-LLAVE. Esta raíz luego se elimina de *H* mediante una llamada de BINOMIAL-HEAP-EXTRACT-MIN.

El procedimiento BINOMIAL-HEAP-DELETE tarda $O(\lg n)$ tiempo.

Ejercicios 19.2-1

Página 402

Escriba el pseudocódigo para BINOMIAL-HEAP-MERGE.

Ejercicios 19.2-2

Muestre el montón binomial que resulta cuando se inserta un nodo con la clave 24 en el binomio montón que se muestra en la Figura 19.7 (d) .

Ejercicios 19.2-3

Muestre el montón binomial que resulta cuando el nodo con la clave 28 se elimina del binomio montón que se muestra en la Figura 19.8 (c) .

Ejercicios 19.2-4

Argumente la corrección de BINOMIAL-HEAP-UNION utilizando el siguiente invariante de bucle:

- Al comienzo de cada iteración del **tiempo** de bucle de las líneas 9-21, x puntos a una raíz que es uno de los siguientes:
  - la única raíz de su grado,
  - la primera de las dos únicas raíces de su grado, o
  - la primera o la segunda de las tres únicas raíces de su grado.
- Además, todas las raíces que preceden al predecesor de x en la lista de raíces tienen grados únicos en la lista raíz, y si el predecesor de x tiene un grado diferente al de x , su grado en la lista raíz también es única. Finalmente, los grados de los nodos aumentan monótonamente a medida que atravesamos la lista raíz.

Ejercicios 19.2-5

Explique por qué el procedimiento BINOMIAL-HEAP-MINIMUM podría no funcionar correctamente si las teclas puede tener el valor ∞. Vuelva a escribir el pseudocódigo para que funcione correctamente en tales casos.

Ejercicios 19.2-6

Suponga que no hay forma de representar la clave -∞. Reescribe el BINOMIAL-HEAP-DELETE procedimiento para trabajar correctamente en esta situación. Todavía debería tomar O (lg n) tiempo.

Página 403

Ejercicios 19.2-7

Discutir la relación entre insertar en un montón binomial e incrementar un binario número y la relación entre unir dos montones binomiales y sumar dos binarios números.

Ejercicios 19.2-8

A la luz del ejercicio 19.2-7 , reescriba BINOMIAL-HEAP-INSERT para insertar un nodo directamente en un montón binomial sin llamar a BINOMIAL-HEAP-UNION.

Ejercicios 19.2-9

Muestre que si las listas raíz se mantienen en orden estrictamente decreciente por grado (en lugar de estrictamente orden ascendente), cada una de las operaciones del montón binomial se puede implementar sin cambiar su tiempo de funcionamiento asintótico.

Ejercicios 19.2-10

Encuentre entradas que causen BINOMIAL-HEAP-EXTRACT-MIN, BINOMIAL-HEAP-DECREASE-KEY y BINOMIAL-HEAP-DELETE para ejecutar en $\Omega(\lg n)$ tiempo. Explique por qué tiempos de ejecución en el peor de los casos de BINOMIAL-HEAP-INSERT, BINOMIAL-HEAP-MINIMUM, y BINOMIAL-HEAP-UNION son pero no $\Omega(\lg n)$. (Vea el problema 3-5 .)

Problemas 19-1: 2-3-4 montones

El capítulo 18 introdujo el árbol 2-3-4, en el que cada nodo interno (excepto posiblemente el raíz) tiene dos, tres o cuatro hijos y todas las hojas tienen la misma profundidad. En este problema, nosotros deberá implementar *2-3-4 montones* , que admiten las operaciones de *montones* fusionables.

Los 2-3-4 montones se diferencian de los 2-3-4 árboles en las siguientes formas. En 2-3-4 montones, solo hojas almacenar claves, y cada hoja x almacena exactamente una clave en el campo *clave* [x]. No hay particular ordenamiento de las llaves en las hojas; es decir, de izquierda a derecha, las claves pueden estar en cualquier orden. Cada nodo interno x contiene un valor *pequeño* [x] que es igual a la clave más pequeña almacenada en cualquier hoja en el subárbol enraizado en x . La raíz r contiene una *altura de* campo [r] que es la altura del árbol.

Página 404

Finalmente, 2-3-4 montones están pensados para mantenerse en la memoria principal, de modo que las lecturas y escrituras del disco sean innecesario.

Implemente las siguientes operaciones de pila 2-3-4. Cada una de las operaciones de las partes (a) - (e) debe ejecutar en tiempo $O(\lg n)$ en un montón 2-3-4 con n elementos. La operación UNION en la parte (f) debería ejecutar en tiempo $O(\lg n)$, donde n es el número de elementos en los dos montones de entrada.

a. MINIMO, que devuelve un puntero a la hoja con la clave más pequeña.
si. DECREASE-KEY, que disminuye la clave de una hoja dada x a un valor dado $k \leq$ *tecla* [x].
C. INSERT, que inserta la hoja x con la tecla k .
re. DELETE, que elimina una hoja x *determinada* .
mi. EXTRACT-MIN, que extrae la hoja con la tecla más pequeña.
F. UNION, que une dos montones 2-3-4, devolviendo un solo montón 2-3-4 y destruyendo los montones de entrada.

Problemas 19-2: algoritmo de árbol de expansión mínimo que utiliza montones binomiales

El capítulo 23 presenta dos algoritmos para resolver el problema de encontrar un árbol de expansión mínimo de un gráfico no dirigido. Aquí, veremos cómo se pueden usar los montones binomiales para idear un diferente algoritmo de árbol de expansión mínimo.

Se nos da un conectado, grafo no dirigido $G = (V , E)$ con una función de ponderación $w : E \rightarrow \mathbf{R}$. Nosotros llame a $w (u , v)$ el peso de la arista (u , v). Deseamos encontrar un árbol de expansión mínimo para G : an subconjunto acíclico TE que conecta todos los vértices en V y cuyo peso total

se minimiza.

El siguiente pseudocódigo, que puede demostrarse que es correcto utilizando técnicas de la Sección 23.1 , construye un árbol de expansión T mínimo . Mantiene una partición $\{ V_i \}$ de los vértices de V y, con cada conjunto V_i , un conjunto

E_{yo} $(u , v): u V_i$ o $v V_i \}$

de aristas incidentes en vértices en V_I .

```
MST ( G )
 1 T ← Ø
 2 para cada vértice v_i                    V [ G ]
 3 ¿ V_i ← { v_i }
 4                E_i ← {( v_i , v ) E [ G ]}
 5 mientras haya más de un conjunto V_i
 6 hacen elegir cualquier conjunto V_i
 7          extraer el borde de peso mínimo ( u , v ) de E_i
 8          asumir sin pérdida de generalidad que u                V_i y v          V_j
 9          si yo ≠ j
```

Página 405

```
10          luego T ← T          {( u , v )}
11               V_i ← V_i       V_j , destruyendo V_j
12               E_yo ← E_yo     E_j
```

Describe cómo implementar este algoritmo usando montones binomiales para administrar el vértice y conjuntos de bordes. ¿Necesita cambiar la representación de un montón binomial? ¿Necesitas agregar operaciones más allá de las operaciones de pila fusionable dadas en la figura 19.1 ? Dar el tiempo de ejecución de su implementación.

Notas del capítulo

Los montones binomiales fueron introducidos en 1978 por Vuillemin [307] . Brown [49 , 50] estudió su propiedades en detalle.

Capítulo 20: Montones de Fibonacci

Visión general

En el capítulo 19 , vimos cómo los montones binomiales apoyan en el peor de los casos O (lg n) el tiempo de fusión operaciones de montón INSERT, MINIMUM, EXTRACT-MIN y UNION, más las operaciones DISMINUIR TECLA y BORRAR. En este capítulo examinaremos los montones de Fibonacci, que soportan las mismas operaciones pero tienen la ventaja de que operaciones que no involucran eliminar un elemento ejecutado en O (1) tiempo amortizado.

Desde un punto de vista teórico, los montones de Fibonacci son especialmente deseables cuando el número de Las operaciones EXTRACT-MIN y DELETE son pequeñas en relación con el número de otras operaciones realizado. Esta situación surge en muchas aplicaciones. Por ejemplo, algunos algoritmos para Los problemas de gráficos pueden llamar a DECREASE-KEY una vez por borde. Para gráficos densos, que tienen muchas aristas, el tiempo amortizado O (1) de cada llamada de DECREASE-KEY se suma a una gran

mejora con respecto al tiempo Θ (lg *n*) del peor de los casos de montones binarios o binomiales. Algoritmos rápidos para problemas tales como calcular árboles de expansión mínimos (Capítulo 23) y encontrar Las rutas más cortas de origen (Capítulo 24) hacen un uso esencial de los montones de Fibonacci.

Sin embargo, desde un punto de vista práctico, los factores constantes y la complejidad de programación de Los montones de Fibonacci los hacen menos deseables que los montones binarios (o *k* -ary) ordinarios para la mayoría aplicaciones. Por lo tanto, los montones de Fibonacci son predominantemente de interés teórico. Si mucho estructura de datos más simple con los mismos límites de tiempo amortizado que los montones de Fibonacci fueron desarrollado, también sería de uso práctico.

Como un montón binomial, un montón de Fibonacci es una colección de árboles. Montones de Fibonacci, de hecho, son vagamente basado en montones binomiales. Si nunca se invoca ni DECREASE-KEY ni DELETE en un montón de Fibonacci, cada árbol del montón es como un árbol binomial. Los montones de Fibonacci tienen más estructura relajada que los montones binomiales, sin embargo, lo que permite un mejor tiempo asintótico límites. El trabajo que mantiene la estructura se puede retrasar hasta que sea conveniente realizarlo.

Al igual que las tablas dinámicas de la Sección 17.4 , los montones de Fibonacci ofrecen un buen ejemplo de datos estructura diseñada teniendo en cuenta el análisis amortizado. La intuición y los análisis de Fibonacci Las operaciones de pila en el resto de este capítulo se basan en gran medida en el método potencial de Sección 17.3.

Página 406

La exposición en este capítulo asume que ha leído el Capítulo 19 sobre montones binomiales. los las especificaciones para las operaciones aparecen en ese capítulo, al igual que la tabla de la Figura 19.1 , que resume los límites de tiempo para operaciones en montones binarios, montones binomiales y Fibonacci muchísimo. Nuestra presentación de la estructura de los montones de Fibonacci se basa en la de binomial-heap estructura, y algunas de las operaciones realizadas en montones de Fibonacci son similares a las realizado en montones binomiales.

Al igual que los montones binomiales, los montones de Fibonacci no están diseñados para dar un soporte eficiente a la operación BUSCAR; operaciones que se refieren a un nodo dado, por lo tanto, requieren un puntero a ese nodo como parte de su entrada. Cuando usamos un montón de Fibonacci en una aplicación, a menudo almacenamos un manipular al objeto de aplicación correspondiente en cada elemento del montón de Fibonacci, así como un manipular al elemento correspondiente de Fibonacci-heap en cada objeto de aplicación.

La sección 20.1 define los montones de Fibonacci, analiza su representación y presenta el potencial función utilizada para su análisis amortizado. La sección 20.2 muestra cómo implementar el operaciones de pila fusionable y lograr los límites de tiempo amortizados que se muestran en la figura 19.1 . los las dos operaciones restantes, DECREASE-KEY y DELETE, se presentan en la Sección 20.3. Finalmente, la Sección 20.4 remata una parte clave del análisis y también explica el curioso nombre de la estructura de datos.

20.1 Estructura de los montones de Fibonacci

Como un montón binomial, un *montón de Fibonacci* es una colección de árboles ordenados en min-montones. Los árboles en Sin embargo, un montón de Fibonacci no está limitado a ser árboles binomiales. La figura 20.1 (a) muestra una ejemplo de un montón de Fibonacci.

Figura 20.1: *(a)* Un montón de Fibonacci que consta de cinco árboles ordenados por min-montón y 14 nodos. La línea discontinua indica la lista raíz. El nodo mínimo del montón es el nodo que contiene la llave 3. Los tres nodos marcados están ennegrecidos. El potencial de este Fibonacci en particular montón es $5 + 2 \cdot 3 = 11$. *(b)* Una representación más completa que muestra los punteros *p* (flechas hacia arriba), *niño* (flechas hacia abajo) e *izquierda* y *derecha* (flechas laterales). Estos detalles se omiten en el

las cifras restantes de este capítulo, ya que toda la información mostrada aquí se puede determinar de lo que aparece en el inciso a).

A diferencia de los árboles dentro de montones binomiales, que están ordenados, los árboles dentro de montones de Fibonacci son arraigado pero desordenado. Como muestra la figura 20.1 (b) , cada nodo x contiene un puntero p [x] a su

Página 407

padre y un apuntador *hijo* [x] a cualquiera de sus hijos. Los hijos de x están vinculados en una lista circular, doblemente enlazada, que llamamos la lista secundaria de x . Cada niño y en una lista de niños tiene punteros *dejaron* [Y] y *derecho* [y] a ese punto y que está a la izquierda y los hermanos derecha, respectivamente. Si el nodo y es un hijo único, luego *izquierda* [y] = *derecha* [y] = y . El orden en el que aparecen los hermanos en una lista secundaria es arbitrario.

Las listas circulares, doblemente enlazadas (ver Sección 10.2) tienen dos ventajas para su uso en Fibonacci muchísimo. Primero, podemos eliminar un nodo de una lista circular doblemente enlazada en el tiempo O (1). Segundo, dadas dos de estas listas, podemos concatenarlas (o "empalmarlas" juntas) en una circular, lista doblemente enlazada en el tiempo O (1). En las descripciones de las operaciones del montón de Fibonacci, referirse a estas operaciones de manera informal, dejando que el lector complete los detalles de su implementaciones.

Serán útiles otros dos campos en cada nodo. El número de hijos en la lista de hijos del nodo x se almacena en *grados* [x]. La *marca de* campo con valor booleano [x] indica si el nodo x ha perdido un hijo desde la última vez que x se convirtió en hijo de otro nodo. Los nodos recién creados son sin marcar, y un nodo x deja de estar marcado cada vez que se convierte en hijo de otro nodo. Hasta que veamos la operación DECREASE-KEY en la Sección 20.3 , simplemente estableceremos todas las *marcas* campos a FALSO.

Se accede a un montón de Fibonacci dado H mediante un puntero *min* [H] a la raíz de un árbol que contiene un clave mínima; este nodo se denomina nodo mínimo del montón de Fibonacci. Si un Fibonacci el montón H está vacío, entonces *min* [H] = NIL.

Las raíces de todos los árboles en un montón de Fibonacci están vinculadas entre sí usando sus *lados izquierdo* y *derecho.* punteros en una lista circular, doblemente enlazada llamada lista raíz del montón de Fibonacci. los pointer *min* [H] apunta al nodo en la lista raíz cuya clave es mínima. El orden del árboles dentro de una lista raíz es arbitrario.

Confiamos en otro atributo para un montón de Fibonacci H : el número de nodos actualmente en H es mantenido en n [H].

Función potencial

Como se mencionó, usaremos el método potencial de la Sección 17.3 para analizar el desempeño de las operaciones del montón de Fibonacci. Para un montón de Fibonacci dado H , indicamos por t (H) el número de árboles en la lista de raíz de H y por m (H) el número de nodos marcados en H . El potencial de El montón de Fibonacci H se define entonces por

(20,1)

(Obtendremos algo de intuición para esta función potencial en la sección 20.3.) Por ejemplo, el El potencial del montón de Fibonacci que se muestra en la figura 20.1 es 5 + 2 · 3 = 11. El potencial de un conjunto de Montones de Fibonacci es la suma de los potenciales de sus montones de Fibonacci constituyentes. Deberíamos Supongamos que una unidad de potencial puede pagar una cantidad constante de trabajo, donde la constante es suficientemente grande para cubrir el costo de cualquiera de los trabajos específicos de tiempo constante que podría encontrar.

Suponemos que una aplicación de montón de Fibonacci comienza sin montones. El potencial inicial, por lo tanto, es 0 y, según la ecuación (20.1) , el potencial no es negativo en todos los momentos posteriores.

Página 408

De la ecuación (17.3) , un límite superior del costo total amortizado es, por lo tanto, un límite superior de el costo real total de la secuencia de operaciones.

Grado máximo

Los análisis amortizados que realizaremos en las secciones restantes de este capítulo suponen que hay un límite superior conocido $D(n)$ en el grado máximo de cualquier nodo en un n -node Montón de Fibonacci. El ejercicio 20.2-3 muestra que cuando solo las operaciones del montón apoyado, $D(n) \leq \lfloor \lg n \rfloor$. En la Sección 20.3 , mostraremos que cuando apoyamos DISMINUCIÓN-KEY y DELETE también, $D(n) = O(\lg n)$.

20.2 Operaciones de pila fusionable

En esta sección, describimos y analizamos las operaciones de pila fusionable implementadas para Montones de Fibonacci. Si solo estas operaciones -HACER-HEAP, INSERT, MINIMUM, EXTRACT-MIN, y UNION-deben ser compatibles, cada montón de Fibonacci es simplemente una colección de árboles binomiales "desordenados". Un *árbol binomial desordenado* es como un árbol binomial, y también es definido de forma recursiva. El árbol binomial desordenado U_0 consta de un solo nodo y un árbol binomial desordenado U_k consta de dos árboles binomiales desordenados U_{k-1} para los cuales la raíz de uno se convierte en *cualquier* hijo de la raíz del otro. Lema 19.1 , que da propiedades de árboles binomiales, también se aplica a los árboles binomiales no ordenados, pero con la siguiente variación en propiedad 4 (vea el ejercicio 20.2-2):

- 4 ′. Para el árbol binomial desordenado U_k , la raíz tiene grado k , que es mayor que el de cualquier otro nodo. Los hijos de la raíz son raíces de los subárboles U_0 , U_1 ,. . . , U_{k-1} en algún orden.

Por lo tanto, si un montón de Fibonacci de n nodos es una colección de árboles binomiales desordenados, entonces $D(n) = \lg n$.

La idea clave en las operaciones de pila fusionable en montones de Fibonacci es retrasar el trabajo siempre que posible. Existe una compensación de rendimiento entre las implementaciones de las diversas operaciones. Si el número de árboles en un montón de Fibonacci es pequeño, entonces durante una operación EXTRACT-MIN podemos determinar rápidamente cuál de los nodos restantes se convierte en el nuevo nodo mínimo. Sin embargo, como vimos con los montones binomiales en el ejercicio 19.2-10 , pagamos un precio por asegurarnos de que el número de árboles es pequeño: puede tomar hasta $\Omega(\lg n)$ tiempo para insertar un nodo en un binomio montón o para unir dos montones binomiales. Como veremos, no intentamos consolidar árboles en un montón de Fibonacci cuando insertamos un nuevo nodo o unimos dos montones. Salvamos la consolidación para la operación EXTRACT-MIN, que es cuando realmente necesitamos encontrar el nuevo mínimo nodo.

Creando un nuevo montón de Fibonacci

Para hacer un montón de Fibonacci vacío, el procedimiento MAKE-FIB-HEAP asigna y devuelve el Objeto de montón de Fibonacci H , donde $n[H] = 0$ y $min[H] =$ NIL; no hay árboles en H . Como $t(H) = 0$ y $m(H) = 0$, el potencial del montón de Fibonacci vacío es $\Phi(H) = 0$. El El costo amortizado de MAKE-FIB-HEAP es, por lo tanto, igual a su costo real $O(1)$.

Insertar un nodo

El siguiente procedimiento inserta el nodo x en el montón H *de* Fibonacci , asumiendo que el nodo tiene ya ha sido asignado y esa *clave* $[x]$ ya se ha completado.

```
INSERCIÓN DE FIB-HEAP ( H , x )
 1 grado [ x ] ← 0
 2 p [ x ] ← NULO
 3 niño [ x ] ← NULO
 4 izquierda [ x ] ← x
 5 a la derecha [ x ] ← x
 6 marca [ x ] ← FALSO
 7 concatenar la lista raíz que contiene x con la lista raíz H
 8 si min [ H ] = NIL o tecla [ x ] < tecla [ min [ H ]]
 9 luego min [ H ] ← x
```

10 $n[H] \leftarrow n[H] + 1$

Después de las líneas 1-6, inicialice los campos estructurales del nodo x , haciéndolo su propio círculo, doblemente lista enlazada, la línea 7 agrega x a la lista raíz de H en $O(1)$ tiempo real. Por tanto, el nodo x *se* convierte en un árbol ordenado min-heap de un solo nodo, y por lo tanto un árbol binomial desordenado, en el Fibonacci montón. No tiene hijos y no está marcado. Las líneas 8-9 luego actualizan el puntero al mínimo nodo del montón H de Fibonacci si es necesario. Finalmente, la línea 10 incrementa $n[H]$ para reflejar la suma del nuevo nodo. La figura 20.2 muestra un nodo con la clave 21 insertada en el montón de Fibonacci de Figura 20.1 .

Figura 20.2: Insertar un nodo en un montón de Fibonacci. *(a)* A Fibonacci montón H . *(b)* Fibonacci montón H después de insertar el nodo con la clave 21. El nodo se convierte en su propio min-heap-árbol ordenado y luego se agrega a la lista raíz, convirtiéndose en el hermano izquierdo de la raíz.

A diferencia del procedimiento BINOMIAL-HEAP-INSERT, FIB-HEAP-INSERT no intenta consolidar los árboles dentro del montón de Fibonacci. Si k consecutivos FIB-HEAP-INSERT ocurren operaciones, luego se agregan k árboles de un solo nodo a la lista raíz.

Para determinar el costo amortizado de FIB-HEAP-INSERT, sea H el montón de Fibonacci de entrada y H' es el montón de Fibonacci resultante. Entonces, $t(H') = t(H) + 1$ y $m(H') = m(H)$, y el el aumento de potencial es

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1.$$

Dado que el costo real es $O(1)$, el costo amortizado es $O(1) + 1 = O(1)$.

Encontrar el nodo mínimo

El nodo mínimo de un montón de Fibonacci H viene dado por el puntero $min[H]$, por lo que podemos encontrar el nodo mínimo en tiempo real $O(1)$. Debido a que el potencial de H no cambia, el amortizado El costo de esta operación es igual a su costo real $O(1)$.

Uniendo dos montones de Fibonacci

Página 410

El siguiente procedimiento une los montones de Fibonacci H_1 y H_2 , destruyendo H_1 y H_2 en el proceso. Simplemente concatena las listas raíz de H_1 y H_2 y luego determina el nuevo nodo mínimo.

```
UNIÓN FIB-HEAP (H1 , H2 )
1 H ← MAKE-FIB-HEAP ()
2 min[H] ← min[H1 ]
3 concatenar la lista raíz de H2 con la lista raíz de H
4 si (min[H1 ] = NIL) o (min[H2 ] ≠ NIL y min[H2 ] < min[H1 ])
5 luego min[H] ← min[H2 ]
6 n[H] ← n[H1 ] + n[H2 ]
7 libera los objetos H1 y H2
8 vuelve H
```

Líneas 1-3 concatenan las listas profundas de H_1 y H_2 en una nueva lista de raíz H . Conjunto de líneas 2, 4 y 5 el nodo mínimo de H , y la línea 6 establece $n[H]$ en el número total de nodos. El Fibonacci montón objetos H_1 y H_2 son liberados en la línea 7, y la línea 8 vuelve la resultante Fibonacci montón H . Como en el procedimiento FIB-HEAP-INSERT, no se produce consolidación de árboles.

El cambio de potencial es

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\ &= 0,\end{aligned}$$

porque $t(H) = t(H_1) + t(H_2)$ y $m(H) = m(H_1) + m(H_2)$. El costo amortizado de FIB-HEAP- Por lo tanto, UNION es igual a su costo real $O(1)$.

Extrayendo el nodo mínimo

El proceso de extracción del nodo mínimo es la más complicada de las operaciones
presentado en esta sección. También es donde el trabajo tardío de consolidación de árboles en la raíz
finalmente aparece la lista. El siguiente pseudocódigo extrae el nodo mínimo. El código asume
Por conveniencia, cuando se elimina un nodo de una lista vinculada, los punteros que permanecen en la lista
se actualizan, pero los punteros en el nodo extraído no se modifican. También utiliza el auxiliar
procedimiento CONSOLIDAR, que será presentado en breve.

```
FIB-HEAP-EXTRACT-MIN ( H )
 1 z ← min [ H ]
 2 si z ≠ NULO
 3 luego para cada hijo x de z
 4                    hacer add x a la lista raíz del H
 5                          p [ x ] ← NULO
 6              eliminar z de la lista raíz de H
 7              si z = derecha [ z ]
 8                    luego min [ H ] ← NIL
 9                    más min [ H ] ← derecha [ z ]
10                          CONSOLIDAR ( H )
11              n [ H ] ← n [ H ] - 1
12 volver z
```

Como se muestra en la Figura 20.3 , FIB-HEAP-EXTRACT-MIN funciona primero haciendo una raíz de
cada uno de los hijos del nodo mínimo y eliminar el nodo mínimo de la lista raíz. Eso

Página 411

luego consolida la lista de raíces vinculando raíces de igual grado hasta que como máximo queda una raíz
de cada grado.

Figura 20.3: La acción de FIB-HEAP-EXTRACT-MIN. *(a)* A Fibonacci montón *H* . *(b)* El
situación después de que el nodo mínimo *z* se elimine de la lista raíz y sus hijos se agreguen a
la lista raíz. *(c)* - *(e)* La matriz *A* y los árboles después de cada una de las tres primeras iteraciones de *for*
bucle de las líneas 3-13 del procedimiento CONSOLIDAR. La lista raíz se procesa comenzando en
el nodo apuntado por *min* [*H*] y siguiendo los punteros *derechos* . Cada parte muestra los valores de *w*
y *x* al final de una iteración. *(f)* - *(h)* La siguiente iteración del ciclo *for* , con los valores de *w*
y *x Se* muestra al final de cada iteración del *tiempo* de bucle de líneas 6-12. La parte (f) muestra el
situación después de la primera vez a través del *tiempo* de bucle. El nodo con la clave 23 se ha vinculado a
el nodo con la clave 7, que ahora está apuntado por *x* . En la parte (g), el nodo con clave 17 ha sido

vinculado al nodo con la clave 7, que todavía está apuntado por x . En la parte (h), el nodo con clave 24 se ha vinculado al nodo con la clave 7. Dado que A [3] no señaló previamente ningún nodo , en al final de la iteración del ciclo *for* , A [3] se establece para que apunte a la raíz del árbol resultante. *(i) - (l)* El situación después de cada una de las siguientes cuatro iteraciones del ciclo *for* . *(m)* Fibonacci montón H después reconstrucción de la lista raíz a partir de la matriz A y determinación del nuevo puntero *min* [H].

Página 412

Comenzamos en la línea 1 guardando un puntero z en el nodo mínimo; este puntero se devuelve en el final. Si z = NIL, entonces el montón H *de* Fibonacci ya está vacío y hemos terminado. De lo contrario, como en el Procedimiento BINOMIAL-HEAP-EXTRACT-MIN, que elimine nodo z de H al hacer todos de z 's hijos raíces de H en las líneas 3-5 (poniéndolos en la lista raíz) y quitando z de la raíz lista en la línea 6. Si z = *derecha* [z] después de la línea 6, entonces z era el único nodo en la lista raíz y no tenía niños, por lo que todo lo que queda es hacer que el montón de Fibonacci esté vacío en la línea 8 antes de devolver z . De lo contrario, establecemos el puntero *min* [H] en la lista raíz para apuntar a un nodo que no sea z (en este caso, *derecha* [z]), que no necesariamente será el nuevo nodo mínimo cuando FIB-HEAP-EXTRACT-MIN está listo. La figura 20.3 (b) muestra el montón de Fibonacci de la figura 20.3 (a) después de la línea 9 se ha realizado.

El siguiente paso, en el que reducimos la cantidad de árboles en el montón de Fibonacci, es *consolidar* la lista raíz de H ; esto se realiza mediante la llamada CONSOLIDAR (H). Consolidando la raíz La lista consiste en ejecutar repetidamente los siguientes pasos hasta que cada raíz de la lista raíz tenga un valor de *grado* distinto .

1. Encontrar dos raíces x e y en la lista de raíz con el mismo grado, donde *la tecla* [x] ≤ *clave* [Y].
2. **Vincular** y con x : eliminar y de la lista raíz y convertir y *en* un hijo de x . Esta operación es realizado por el procedimiento FIB-HEAP-LINK. El *grado de* campo [x] se incrementa, y se borra la marca en y , si la hay.

El procedimiento CONSOLIDAR utiliza una matriz auxiliar A [0 D (n [H])]; si A [i] = y , entonces y es actualmente una raíz con *grado* [y] = i .

```
CONSOLIDAR ( H )
 1 para i ← 0 a D ( n [ H ])
 2 hacer A [ i ] ← NULO
 3 para cada nodo w en la lista raíz de H
 4 hacer x ← w
 5                 d ← grado [ x ]
 6                 mientras que A [ d ] ≠ NIL
 7                         do y ← A [ d ]  ▷ Otro nodo con el mismo grado que x .
 8                             si tecla [ x ]> tecla [ y ]
 9                                 luego intercambia x ↔ y
10                             FIB-HEAP-LINK ( H , y , x )
11                             A [ d ] ← NULO
12                             d ← d + 1
13                 A [ d ] ← x
14 min [ H ] ← NULO
15 para i ← 0 a D ( n [ H ])
16 hacer si A [ i ] ≠ NIL
17                         luego agregue A [ i ] a la lista raíz de H
18                             si min [ H ] = NIL o tecla [ A [ i ]] < tecla [ min [ H ]]
19                                 luego min [ H ] ← A [ i ]
FIB-HEAP-LINK ( H , y , x )
1 eliminar y de la lista raíz de H
2 hacer que y sea un hijo de x , grado creciente [ x ]
3 marca [ y ] ← FALSO
```

En detalle, el procedimiento CONSOLIDAR funciona de la siguiente manera. Las líneas 1-2 inicializan A haciendo cada entrada NULO. El ciclo **for** de las líneas 3-13 procesa cada raíz w en la lista de raíces. Después procesando cada raíz w , termina en un árbol enraizado en algún nodo x , que puede o no ser idéntico a w . De las raíces procesadas, ninguna otra tendrá el mismo grado que x , por lo que

Página 413

establezca la entrada de matriz A [*grado* [x]] para que apunte ax . Cuando este ciclo **for** termina, como máximo una raíz de cada grado permanecerá y la matriz A apuntará a cada raíz restante.

El **tiempo** de bucle de líneas 6-12 une repetidamente la raíz x del árbol que contiene nodo w a otro árbol cuya raíz tenga el mismo grado que x , hasta que ninguna otra raíz tenga el mismo grado. Esta **while** loop mantiene el siguiente invariante:

- Al comienzo de cada iteración del **tiempo** de bucle, d = *grado* [x].

Usamos este ciclo invariante de la siguiente manera:

- **Inicialización: la** línea 5 asegura que el invariante de ciclo se mantenga la primera vez que ingresamos al lazo.
- **Mantenimiento:** En cada iteración del **tiempo** de bucle, A [d] puntos a alguna raíz y . Porque d = *grado* [x] = *grado* [y], queremos vincular x e y . Cualquiera de x y y tiene la La clave más pequeña se convierte en el padre de la otra como resultado de la operación de enlace, por lo que las líneas 8-9 intercambian los punteros ax e y si es necesario. A continuación, vinculamos y con x mediante la llamada FIB-HEAP-LINK (H , y , x) en la línea 10. Esta llamada aumenta el *grado* [x] pero sale *grado* [y] como d . Debido a que el nodo y ya no es una raíz, el puntero a él en la matriz A es eliminado en la línea 11. Debido a que la llamada de FIB-HEAP-LINK incrementa el valor de *grado* [x], la línea 12 restaura el invariante que d = *grado* [x].
- **Terminación:** Repetimos el **mientras** bucle hasta A [d] = NIL, en cuyo caso no hay es otra raíz con el mismo grado que x .

Después de **que** termina el ciclo while, establecemos A [d] ax en la línea 13 y realizamos la siguiente iteración de el bucle **for** .

Las Figuras 20.3 (c) - (e) muestran la matriz A y los árboles resultantes después de las tres primeras iteraciones del **para el** bucle de las líneas 3-13. En la siguiente iteración del bucle **for** , se producen tres enlaces; sus resultados son mostrado en las Figuras 20.3 (f) - (h) . Las figuras 20.3 (i) - (l) muestran el resultado de las siguientes cuatro iteraciones de el bucle **for** .

Todo lo que queda es limpiar. Una vez que se completa el bucle **for** de las líneas 3-13, la línea 14 vacía el Lista de raíz, y las líneas 15-19 reconstruirlo de la matriz A . El montón de Fibonacci resultante es mostrado en la Figura 20.3 (m) . Después de consolidar la lista raíz, FIB-HEAP-EXTRACT-MIN termina decrementando n [H] en la línea 11 y devolviendo un puntero al nodo eliminado z en línea 12.

Observe que si todos los árboles del montón de Fibonacci son árboles binomiales desordenados antes de FIB-HEAP-EXTRACT-MIN se ejecuta, luego todos son árboles binomiales desordenados. Existen dos formas en que se cambian los árboles. Primero, en las líneas 3-5 de FIB-HEAP-EXTRACT-MIN, cada el hijo x de la raíz z *se* convierte en una raíz. Según el ejercicio 20.2-2 , cada árbol nuevo es en sí mismo un árbol binomial. En segundo lugar, los árboles están vinculados por FIB-HEAP-LINK solo si tienen el mismo la licenciatura. Dado que todos los árboles son árboles binomiales desordenados antes de que se produzca el vínculo, dos árboles cuyo cada raíz tiene k hijos deben tener la estructura de U_k . Por tanto, el árbol resultante tiene la estructura de U_{k+1} .

Ahora estamos listos para mostrar que el costo amortizado de extraer el nodo mínimo de un n - el montón de Fibonacci del nodo es O (D (n)). Dejemos que H denote el montón de Fibonacci justo antes de FIB-HEAP-Operación EXTRACT-MIN.

Página 414

El costo real de extraer el nodo mínimo se puede contabilizar de la siguiente manera. Una O (D (n)) La contribución proviene de que hay como máximo D (n) hijos del nodo mínimo que son procesado en FIB-HEAP-EXTRACT-MIN y del trabajo en las líneas 1-2 y 14-19 de CONSOLIDAR. Queda por analizar la contribución del bucle **for** de las líneas 3-13. los el tamaño de la lista raíz al llamar a CONSOLIDATE es como máximo D (n) + t (H) - 1, ya que consiste de los nodos originales de la lista raíz t (H), menos el nodo raíz extraído, más los hijos del nodo extraído, cuyo número como máximo D (n). Cada vez que a través del **tiempo** de bucle de líneas 6-12, una de las raíces está vinculado a otro, y por lo tanto la cantidad total de trabajo realizado en el **para** bucle es como mucho proporcional a D (n) + t (H). Por lo tanto, el trabajo real total en la extracción de El nodo mínimo es O (D (n) + t (H)).

El potencial antes de extraer el nodo mínimo es $t(H) + 2m(H)$, y el potencial después es como máximo $(D(n) + 1) + 2m(H)$, ya que como máximo $D(n) + 1$ raíces permanecen y no hay nudos quedar marcado durante la operación. El coste amortizado es, por tanto, como máximo

$$\begin{aligned} &O(D(n)) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &\quad = O(D(n)) + O(t(H)) - t(H) \\ &\quad = O(D(n)), \end{aligned}$$

ya que podemos escalar las unidades de potencial para dominar la constante oculta en $O(t(H))$. Intuitivamente, el costo de realizar cada enlace se paga mediante la reducción del potencial debido a la enlace está reduciendo el número de raíces en uno. Veremos en la sección 20.4 que $D(n) = O(\lg n)$, de modo que el costo amortizado de extraer el nodo mínimo es $O(\lg n)$.

Ejercicios 20.2-1

Muestre el montón de Fibonacci que resulta de llamar a FIB-HEAP-EXTRACT-MIN en el Montón de Fibonacci que se muestra en la Figura 20.3 (m) .

Ejercicios 20.2-2

Demuestre que el lema 19.1 es válido para árboles binomiales no ordenados, pero con la propiedad 4 ′ en lugar de propiedad 4.

Ejercicios 20.2-3

Muestre que si solo se admiten las operaciones de pila fusionable, el grado máximo $D(n)$ en un n -node Fibonacci montón es a lo sumo $\lfloor \lg n \rfloor$.

Ejercicios 20.2-4

Página 415

El profesor McGee ha ideado una nueva estructura de datos basada en montones de Fibonacci. Un montón de McGee tiene la misma estructura que un montón de Fibonacci y admite las operaciones de montón fusionable. los Las implementaciones de las operaciones son las mismas que para los montones de Fibonacci, excepto que la inserción y el sindicato realizan la consolidación como último paso. ¿Cuáles son los peores tiempos de ejecución de operaciones en montones de McGee? ¿Qué tan novedosa es la estructura de datos del profesor?

Ejercicios 20.2-5

Argumenta que cuando las únicas operaciones en las claves son comparar dos claves (como es el caso de todas las implementaciones en este capítulo), no todas las operaciones de pila fusionable pueden ejecutarse en $O(1)$ tiempo amortizado.

20.3 Disminuir una clave y eliminar un nodo

En esta sección, mostramos cómo disminuir la clave de un nodo en un montón de Fibonacci en $O(1)$ tiempo amortizado y cómo eliminar cualquier nodo de un montón de Fibonacci de n nodos en $O(D(n))$ tiempo amortizado. Estas operaciones no preservan la propiedad de que todos los árboles de Fibonacci montón son árboles binomiales desordenados. Sin embargo, estn lo suficientemente cerca como para que podamos grado máximo $D(n)$ por $O(\lg n)$. Demostrar este límite, que haremos en la Sección 20.4 ,

implica que FIB-HEAP-EXTRACT-MIN y FIB-HEAP-DELETE se ejecutan en O (lg n) amortizado hora.

Disminuir una clave

En el siguiente pseudocódigo para la operación FIB-HEAP-DECREASE-KEY, asumimos como antes de que eliminar un nodo de una lista vinculada no cambia ninguno de los campos estructurales en el nodo eliminado.

```
TECLA DE DISMINUCIÓN DE FIB-HEAP ( H , x , k )
1 si k > tecla [ x ]
2 entonces el error "la clave nueva es mayor que la clave actual"
3 tecla [ x ] ← k
4 y ← p [ x ]
5 si y ≠ NIL y tecla [ x ] < tecla [ y ]
6 luego CORTAR ( H , x , y )
7                    CORTE EN CASCADA ( H , y )
8 si tecla [ x ] < tecla [ min [ H ]]
9 luego min [ H ] ← x
CORTE ( H , x , y )
1 eliminar x de la lista secundaria de y , grado decreciente [ y ]
2 agregue x a la lista raíz de H
3 p [ x ] ← NULO
4 marca [ x ] ← FALSO
CORTE EN CASCADA ( H , y )
1 z ← p [ y ]
2 si z ≠ NULO
```

Página 416

```
3 entonces si marca [ y ] = FALSO
4                    luego marque [ y ] ← VERDADERO
5                    si no CORTAR ( H , y , z )
6                              CORTE EN CASCADA ( H , z )
```

El procedimiento FIB-HEAP-DECREASE-KEY funciona de la siguiente manera. Las líneas 1-3 garantizan que el nuevo key no es mayor que la clave actual de x y luego asigne la nueva clave ax . Si x es una raíz o si $clave$ [x] ≥ $clave$ [y], donde y es el padre de x , entonces no es necesario que ocurran cambios estructurales, ya que min-heap el orden no ha sido violado. Las líneas 4-5 prueban esta condición.

Si se ha infringido el orden de pila mínima, pueden producirse muchos cambios. Comenzamos cortando x en la línea 6. El procedimiento CUT "corta" el vínculo entre x y su padre y , haciendo que x *sea* una raíz.

Usamos los campos de *marca* para obtener los límites de tiempo deseados. Graban un pedacito del historia de cada nodo. Suponga que los siguientes eventos le han sucedido al nodo x :

1. en algún momento, x fue una raíz,
2. luego x se vinculó a otro nodo,
3. Luego, dos hijos de x fueron eliminados mediante cortes.

Tan pronto como se haya perdido el segundo hijo, cortamos x de su padre, convirtiéndolo en una nueva raíz. los la *marca de* campo [x] es VERDADERA si se han realizado los pasos 1 y 2 y se ha cortado un hijo de x . los El procedimiento CUT, por lo tanto, borra la *marca* [x] en la línea 4, ya que realiza el paso 1. (Ahora podemos ver por qué la línea 3 de FIB-HEAP-LINK borra la *marca* [y]: el nodo y está vinculado a otro nodo, por lo que se está realizando el paso 2. La próxima vez que se elimine un hijo de y , la *marca* [y] se establecerá en TRUE).

Aún no hemos terminado, porque x podría ser el segundo hijo cortado de su padre y desde el momento que y estaba vinculado a otro nodo. Por lo tanto, la línea 7 de FIB-HEAP-DECREASE-KEY realiza una operación de corte en cascada en y . Si y es una raíz, entonces la prueba en la línea 2 de CASCADING-CUT hace que el procedimiento simplemente regrese. Si y no está marcado, el procedimiento marca en la línea 4, ya que su primer hijo acaba de ser cortado, y regresa. Si y está marcado, sin embargo, tiene acaba de perder a su segundo hijo; y se corta en la línea 5, y CASCADING-CUT se llama a sí mismo recursivamente en la línea 6 en y padres 's z . El procedimiento CASCADING-CUT repite su camino hacia arriba del árbol hasta se encuentra un nodo raíz o no marcado.

Una vez que se han producido todos los cortes en cascada, las líneas 8-9 de FIB-HEAP-DECREASE-KEY terminan actualizando *min* [H] si es necesario. El único nodo cuya clave cambió fue el nodo x cuya clave clave disminuida. Por lo tanto, el nuevo nodo mínimo es el nodo mínimo original o el nodo x .

La figura 20.4 muestra la ejecución de dos llamadas de FIB-HEAP-DECREASE-KEY, comenzando con el montón de Fibonacci que se muestra en la figura 20.4 (a) . La primera llamada, que se muestra en la Figura 20.4 (b) , implica

sin cortes en cascada. La segunda llamada, que se muestra en las Figuras 20.4 (c) - (e) , invoca dos cortes en cascada.

Página 417

Figura 20.4: Dos llamadas de FIB-HEAP-DECREASE-KEY. *(a)* El montón de Fibonacci inicial. *(si)* El nodo con la clave 46 tiene su clave reducida a 15. El nodo se convierte en una raíz y su padre (con la tecla 24), que anteriormente no estaba marcado, se marca. *(c)* - *(e)* El nodo con la clave 35 tiene su clave reducida a 5. En la parte (c), el nodo, ahora con la clave 5, se convierte en una raíz. Sus padre, con la clave 26, está marcado, por lo que se produce un corte en cascada. El nodo con clave 26 se corta de su padre e hizo una raíz sin marcar en (d). Se produce otro corte en cascada, ya que el nodo con La clave 24 también está marcada. Este nodo se corta de su padre y se convierte en una raíz sin marcar en parte (mi). Los cortes en cascada se detienen en este punto, ya que el nodo con la clave 7 es una raíz. (Incluso si esto nodo no fuera una raíz, los cortes en cascada se detendrían, ya que no está marcado.) El resultado de la La operación FIB-HEAP-DECREASE-KEY se muestra en la parte (e), con min [H] apuntando al nuevo nodo mínimo.

Ahora mostraremos que el costo amortizado de FIB-HEAP-DECREASE-KEY es solo O (1). Comenzamos determinando su costo real. El procedimiento FIB-HEAP-DECREASE-KEY toma O (1) tiempo, más el tiempo para realizar los cortes en cascada. Supongamos que CASCADING-CUT es llamado recursivamente c veces desde una invocación dada de FIB-HEAP-DECREASE-KEY. Cada la llamada de CASCADING-CUT toma O (1) tiempo sin incluir las llamadas recursivas. Por lo tanto, el costo real de FIB-HEAP-DECREASE-KEY, incluidas todas las llamadas recursivas, es O (c).

A continuación, calculamos el cambio de potencial. Sea H el montón de Fibonacci justo antes de la Operación FIB-HEAP-DECREASE-KEY. Cada llamada recursiva de CASCADING-CUT, excepto para el último, corta un nodo marcado y borra el bit de marca. Luego, hay árboles t (H) + c (los árboles originales t (H), c -1 árboles producidos por cortes en cascada, y el árbol enraizado en x) y en la mayoría de los nodos marcados m (H) - c +2 (c -1 fueron desmarcados por cortes en cascada y la última llamada de CASCADING-CUT puede haber marcado un nodo). El cambio de potencial es, por tanto, como mucho

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c .$$

Por lo tanto, el costo amortizado de FIB-HEAP-DECREASE-KEY es como máximo

$$O(c) + 4 - c = O(1),$$

ya que podemos escalar las unidades de potencial para dominar la constante oculta en O (c).

Ahora puede ver por qué la función potencial se definió para incluir un término que es el doble de número de nodos marcados. Cuando un nodo marcado y es cortado por un corte en cascada, su bit de marca es despejado, por lo que el potencial se reduce en 2. Una unidad de potencial paga por el corte y el despeje

Página 418

del bit de marca, y la otra unidad compensa el aumento de potencial de la unidad debido al nodo *y* convirtiéndose en una raíz.

Eliminar un nodo

Es fácil eliminar un nodo de un montón de Fibonacci de *n* nodos en el tiempo amortizado *O* (*D* (*n*)), como es hecho por el siguiente pseudocódigo. Suponemos que no hay un valor clave de -∞ actualmente en el montón de Fibonacci.

```
FIB-HEAP-DELETE ( H , x )
1 TECLA DE DISMINUCIÓN DE FIB-HEAP ( H , x , -∞)
2 FIB-HEAP-EXTRACT-MIN ( H )
```

FIB-HEAP-DELETE es análogo a BINOMIAL-HEAP-DELETE. Hace que *x se* convierta en el nodo mínimo en el montón de Fibonacci dándole una clave única y pequeña de -∞. El nodo *x* es entonces extraído del montón de Fibonacci mediante el procedimiento FIB-HEAP-EXTRACT-MIN. los el tiempo amortizado de FIB-HEAP-DELETE es la suma del tiempo amortizado *O* (1) de FIB-HEAP-DECREASE-KEY y el tiempo de amortización *O* (*D* (*n*)) de FIB-HEAP-EXTRACT-MIN. Desde que nosotros Verá en la Sección 20.4 que *D* (*n*) = *O* (lg *n*), el tiempo amortizado de FIB-HEAP-DELETE es *O* (lg *n*).

Ejercicios 20.3-1

Suponga que se marca una raíz *x* en un montón de Fibonacci. Explica cómo *x* llegó a ser una raíz marcada. Argumenta que no importa para el análisis que *x* esté marcado, aunque no sea una raíz que primero fue vinculado a otro nodo y luego perdió un hijo.

Ejercicios 20.3-2

Justifique el tiempo amortizado *O* (1) de FIB-HEAP-DECREASE-KEY como un costo promedio por operación mediante análisis agregado.

20.4 Delimitación del grado máximo

Para demostrar que el tiempo amortizado de FIB-HEAP-EXTRACT-MIN y FIB-HEAP-DELETE es *O* (lg *n*), debemos mostrar que el límite superior *D* (*n*) en el grado de cualquier nodo de un *n* -nodo El montón de Fibonacci es *O* (lg *n*). Por el ejercicio 20.2-3 , cuando todos los árboles del montón de Fibonacci son árboles binomiales no ordenados, *D* (*n*) = ⌊lg *n* ⌋. Los cortes que ocurren en FIB-HEAP-DECREASE-KEY, sin embargo, puede hacer que los árboles dentro del montón de Fibonacci violen el binomio desordenado propiedades de los árboles. En esta sección, mostraremos que debido a que cortamos un nodo de su padre como tan pronto como pierde dos hijos, *D* (*n*) es *O* (lg *n*). En particular, mostraremos que *D* (*n*) ≤ ⌊log $_{\varphi}$ *n* ⌋, dónde .

La clave del análisis es la siguiente. Para cada nodo *x* dentro de un montón de Fibonacci, defina el tamaño (*x*) para ser el número de nodos, incluido el propio *x* , en el subárbol con raíz en *x* . (Tenga en cuenta que *x* no necesita

Página 419

estar en la lista raíz, puede ser cualquier nodo.) Mostraremos que el tamaño (*x*) es exponencial en *grado* [*x*]. Tenga en cuenta que el *grado* [*x*] siempre se mantiene como un recuento exacto de grado de *x* .

Lema 20.1

Sea *x* cualquier nodo en un montón de Fibonacci, y suponga que *grado* [*x*] = *k* . Deje $y_1, y_2, \ldots, y_k$ denotar a los hijos de *x* en el orden en que fueron vinculados *ax* , desde el más antiguo hasta el

último. Entonces, *grado* [y_1] ≥ 0 y *grado* [y_i] ≥ *i* - 2 para *i* = 2, 3,. . . , *k* .

Prueba Obviamente, *grado* [y_1] ≥ 0.

Para *i* ≥ 2, observamos que cuando Y_i estaba relacionado con *x* , todos y_1 , y_2 ,. . . , y_{i-1} eran hijos de *x* , por lo que debe haber tenido *grado* [*x*] = *i* - 1. El nodo y_i está vinculado *ax* solo si *grado* [*x*] = *grado* [y_i], por lo que también debe haber tenido *grado* [y_i] = *i* - 1 en ese momento. Desde entonces, el nodo y_i ha perdido como máximo una niño, ya que se habría cortado de *x* si hubiera perdido dos hijos. Concluimos que *grado* [y_i] ≥ *i* - 2.

Finalmente llegamos a la parte del análisis que explica el nombre "Montones de Fibonacci". Recordar de la Sección 3.2 que para *k* = 0, 1, 2,. . . , el *k*- ésimo número de Fibonacci está definido por el reaparición

El siguiente lema da otra forma de expresar F_k .

Lema 20.2

Para todos los enteros *k* ≥ 0,

Prueba La prueba es por inducción en *k* . Cuando *k* = 0,

Ahora asumimos la hipótesis inductiva de que , y tenemos

Página 420

El siguiente lema y su corolario completan el análisis. Usan la desigualdad (probado en el ejercicio 3.2-7)

$F_{k+2} \geq \varphi^k$,

donde φ es la proporción áurea, definida en la ecuación (3.22) como .

Lema 20.3

Sea *x* cualquier nodo en un montón de Fibonacci y sea *k* = *grado* [*x*]. Entonces, tamaño (*x*) ≥ $F_{k+2} \geq \varphi^k$, donde .

Prueba Sea s_k el valor mínimo posible de tamaño (*z*) sobre todos los nodos *z* tal que *grado* [*z*] = *k* . Trivialmente, $s_0 = 1$, $s_1 = 2$ y $s_2 = 3$. El número s_k tiene como máximo el tamaño (*x*), y claramente, el valor

de s_k aumenta monótonamente con k . Al igual que en el Lema 20.1 , dejar y_1 , y_2 ,. . . , y_k denotan a los niños de x en el orden en que se vincularon ax . Para calcular un límite inferior de tamaño (x), cuente uno para x *en* sí mismo y uno para el primer hijo y_1 (para cuyo tamaño (y_1) $\geq$ 1), dando

donde la última línea se sigue del Lema 20.1 (de modo que el *grado* [y_i] $\geq i$ - 2) y la monotonicidad de s_k (de modo que $s_{grado[yi]} \geq s_{i-2}$).

Ahora mostramos por inducción en k que $s_k \geq F_{k+2}$ para todo entero no negativo k . Las bases, para k = 0 y k = 1, son triviales. Para el paso inductivo, asumimos que $k \geq 2$ y que $s_i \geq F_{i+2}$ para i = 0, 1,. . . , k - 1. Tenemos

Página 421

Así, hemos demostrado que tamaño (x) $\geq s_k \geq F_{k+2} \geq \varphi_k$.

Corolario 20.4

El grado máximo D (n) de cualquier nodo en un montón de Fibonacci de n nodos es O (lg n).

Prueba Sea x cualquier nodo en unmontón de Fibonacci de n nodos, y sea k = *grado* [x]. Según el lema 20.3 , tenemos $n \geq$ tamaño (x) $\geq \varphi_k$. Tomando logaritmos base- φ nos da $k \leq \log_\varphi n$. (De hecho, como k es un número entero, $k \leq \lfloor \log_\varphi n \rfloor$.) El grado máximo D (n) de cualquier nodo es entonces O (lg n).

Ejercicios 20.4-1

El profesor Pinocho afirma que la altura de un montón de Fibonacci de n nodos es O (lg n). Muestra esa el profesor se equivoca al exhibir, para cualquier entero positivo n , una secuencia de Fibonacci-operaciones de montón que crean un montón de Fibonacci que consta de un solo árbol que es una cadena lineal de n nodos.

Ejercicios 20.4-2

Suponga que generalizamos la regla de corte en cascada para cortar un nodo x de su padre tan pronto como pierde

su k- ésimo hijo, para alguna constante entera k . (La regla de la sección 20.3 usa $k = 2$.) ¿Para qué valores de k es $D (n) = O (\lg n)$?

Problemas 20-1: implementación alternativa de eliminación

El profesor Pisano ha propuesto la siguiente variante del procedimiento FIB-HEAP-DELETE, afirmando que se ejecuta más rápido cuando el nodo que se elimina no es el nodo al que apunta $min [H]$.

Página 422

```
BORRAR PISANO ( H , x )
1 si x = min [ H ]
2 luego FIB-HEAP-EXTRACT-MIN ( H )
3 más y ← p [ x ]
4                 si y ≠ NIL
5                       luego CORTAR ( H , x , y )
6                                 CORTE EN CASCADA ( H , y )
7                 agregue la lista secundaria de x a la lista raíz de H
8                 eliminar x de la lista raíz de H
```

a. La afirmación del profesor de que este procedimiento se ejecuta más rápido se basa en parte en el supuesto que la línea 7 se puede ejecutar en $O (1)$ tiempo real. ¿Qué hay de malo en esta suposición?

si. Proporcione un buen límite superior en el tiempo real de PISANO-DELETE cuando x no es $min [H]$. Su límite debe ser en términos de $grado [x]$ y el número c de llamadas al Procedimiento CASCADING-CUT.

C. Supongamos que llamamos PISANO-DELETE (H , x), y sea H' el montón de Fibonacci que resultados. Suponiendo que el nodo x no es una raíz, limite el potencial de H' en términos de $grado [x]$, c , $t (H)$ $ym (H)$.

re. Concluya que el tiempo amortizado para PISANO-DELETE no es asintóticamente mejor que para FIB-HEAP-DELETE, incluso cuando $x \neq min [H]$.

Problemas 20-2: Más operaciones de montón de Fibonacci

Deseamos aumentar un montón H *de* Fibonacci para admitir dos nuevas operaciones sin cambiar el tiempo de ejecución amortizado de cualquier otra operación del montón de Fibonacci.

a. La operación FIB-HEAP-CHANGE-KEY (H , x , k) cambia la clave del nodo x al valor k . Dar una implementación eficiente de FIB-HEAP-CHANGE-KEY, y analizar el tiempo de ejecución amortizado de su implementación para los casos en los que k es mayor que, menor que o igual a la *tecla* $[x]$.

si. Dar una implementación eficiente de FIB-HEAP-PRUNE (H , r), que elimina min (r , $n [H]$) nodos de H . Los nodos que se eliminan deben ser arbitrarios. Analizar el tiempo de ejecución amortizado de su implementación. (*Sugerencia:* es posible que deba modificar el estructura de datos y función potencial.)

Notas del capítulo

Los montones de Fibonacci fueron introducidos por Fredman y Tarjan [98] . Su artículo también describe la aplicación de los montones de Fibonacci a los problemas de los caminos más cortos de una sola fuente, todos los pares caminos más cortos, emparejamiento bipartito ponderado y el problema del árbol de expansión mínimo.

Posteriormente, Driscoll, Gabow, Shrairman y Tarjan [81] desarrollaron "montones relajados" como un alternativa a los montones de Fibonacci. Hay dos variedades de montones relajados. Uno da lo mismo límites de tiempo amortizados a medida que Fibonacci se acumula. El otro permite que DECREASE-KEY se ejecute en $O (1)$ en el peor de los casos (no amortizado) y EXTRACT-MIN y DELETE para ejecutarse en $O (\lg n)$ tiempo del caso. Los montones relajados también tienen algunas ventajas sobre los montones de Fibonacci en paralelo algoritmos.

Página 423

Consulte también las notas del capítulo del Capítulo 6 para conocer otras estructuras de datos que admiten Operaciones DECREASE-KEY cuando la secuencia de valores devueltos por EXTRACT-MIN llama aumentan monótonamente con el tiempo y los datos son números enteros en un rango específico.

Capítulo 21: Estructuras de datos para disjuntos Conjuntos

Algunas aplicaciones implican agrupar n elementos distintos en una colección de conjuntos disjuntos. Dos Las operaciones importantes son entonces encontrar a qué conjunto pertenece un elemento dado y unir dos conjuntos. Este capítulo explora métodos para mantener una estructura de datos que admita estos operaciones.

La sección 21.1 describe las operaciones soportadas por una estructura de datos de conjuntos disjuntos y presenta un aplicación sencilla. En la Sección 21.2 , observamos una implementación de lista enlazada simple para disjuntos conjuntos. En la Sección 21.3 se ofrece una representación más eficiente utilizando árboles con raíces . El funcionamiento El tiempo que usa la representación del árbol es lineal para todos los propósitos prácticos, pero teóricamente superlineal. La sección 21.4 define y analiza una función de crecimiento muy rápido y su propia inversa de crecimiento lento, que aparece en el tiempo de ejecución de las operaciones en el árbol implementación, y luego utiliza el análisis amortizado para demostrar un límite superior en la ejecución tiempo que es apenas superlineal.

21.1 Operaciones de conjuntos disjuntos

Una *estructura de datos de conjuntos disjuntos* mantiene una colección de conjuntos dinámicos disjuntos. Cada conjunto está identificado por un *representante* , que es algún miembro del conjunto. En algunos aplicaciones, no importa qué miembro se utilice como representante; solo nos importa que si pedimos el representante de un conjunto dinámico dos veces sin modificar el conjunto entre los solicitudes, obtenemos la misma respuesta en ambas ocasiones. En otras aplicaciones, puede haber un regla preespecificada para elegir al representante, como elegir al miembro más pequeño de la set (asumiendo, por supuesto, que los elementos se pueden ordenar).

Como en las otras implementaciones de conjuntos dinámicos que hemos estudiado, cada elemento de un conjunto es representado por un objeto. Dejando que x denote un objeto, deseamos apoyar lo siguiente operaciones:

- MAKE-SET (x) crea un nuevo conjunto cuyo único miembro (y por tanto representativo) es x . Dado que los conjuntos son disjuntos, requerimos que x no esté ya en otro conjunto.
- UNIÓN (x , y) une los conjuntos dinámicos que contienen X e Y , digamos S_x y S_y , en un nuevo conjunto esa es la unión de estos dos conjuntos. Se supone que los dos conjuntos están separados antes de la operación. El representante del conjunto resultante es cualquier miembro de $S_x \quad S_y$, aunque Muchas implementaciones de UNION eligen específicamente al representante de S_x o S_y como nuevo representante. Dado que requerimos que los conjuntos de la colección estén separados, "destruimos" los conjuntos S_x y S_y , eliminándolos de la colección.
- FIND-SET (x) devuelve un puntero al representante del conjunto (único) que contiene x .

A lo largo de este capítulo, analizaremos los tiempos de ejecución de estructuras de datos de conjuntos disjuntos en términos de dos parámetros: n , el número de operaciones MAKE-SET, ym , el número total de Operaciones MAKE-SET, UNION y FIND-SET. Dado que los conjuntos son disjuntos, cada UNIÓN

Página 424

La operación reduce el número de juegos en uno. Después de n - 1 operaciones UNION, por lo tanto, solo queda un juego. Por tanto, el número de operaciones de UNION es como máximo n - 1. Tenga en cuenta también que, dado que las operaciones MAKE-SET están incluidas en el número total de operaciones m , tenemos $m \geq n$. Suponemos que las n operaciones MAKE-SET son las primeras n operaciones realizadas.

Una aplicación de estructuras de datos de conjuntos disjuntos

Una de las muchas aplicaciones de las estructuras de datos de conjuntos disjuntos surge al determinar la componentes conectados de un gráfico no dirigido (consulte la Sección B.4). Figura 21.1 (a) , por ejemplo, muestra un gráfico con cuatro componentes conectados.

Figura 21.1: *(a)* Una gráfica con cuatro componentes conectados: { *a* , *b* , *c* , *d* }, { *e* , *f* , *g* }, { *h* , *i* } y { *j* }. *(b)* La colección de conjuntos disjuntos después de que se procesa cada borde.

El procedimiento CONNECTED-COMPONENTS que sigue usa las operaciones de disjoint-set para calcular los componentes conectados de un gráfico. Una vez que CONNECTED-COMPONENTS ejecutado como un paso de preprocesamiento, el procedimiento SAME-COMPONENT responde consultas sobre si dos vértices están en el mismo componente conectado. [1] (El conjunto de vértices de un gráfico *G* se denota por *V* [*G*], y el conjunto de aristas se denota por *E* [*G*].)

```
COMPONENTES CONECTADOS ( G )
1 para cada vértice v                    V [ G ]
2 hacer MAKE-SET ( v )
3 para cada borde ( u , v ) E [ G ]
4 hacer si FIND-SET ( u ) ≠ FIND-SET ( v )
5                    luego UNION ( u , v )

MISMO COMPONENTE ( u , v )
1 si FIND-SET ( u ) = FIND-SET ( v )
2 luego devuelve VERDADERO
3 si no, devuelve FALSO
```

El procedimiento CONNECTED-COMPONENTS coloca inicialmente cada vértice *v* en su propio conjunto. Luego, para cada borde (*u* , *v*), une los conjuntos que contienen *u* y *v* . Por el ejercicio 21.1-2 , después de todo los bordes se procesan, dos vértices están en el mismo componente conectado si y solo si el los objetos correspondientes están en el mismo conjunto. Por lo tanto, CONNECTED-COMPONENTS calcula se establece de tal manera que el procedimiento SAME-COMPONENT puede determinar si dos

Página 425

los vértices están en el mismo componente conectado. La figura 21.1 (b) ilustra cómo los conjuntos disjuntos son calculados por CONNECTED-COMPONENTS.

En una implementación real de este algoritmo de componentes conectados, las representaciones de el gráfico y la estructura de datos de conjuntos disjuntos deberían hacer referencia entre sí. Es decir, un El objeto que representa un vértice contendría un puntero al objeto de conjunto disjunto correspondiente, y viceversa. Estos detalles de programación dependen del lenguaje de implementación, y nosotros no los aborde más aquí.

Ejercicios 21.1-1

Suponga que CONNECTED-COMPONENTS se ejecuta en el gráfico no dirigido *G* = (*V* , *E*), donde *V* = { *a* , *b* , *c* , *d* , *e* , *f* , *g* , *h* , *i* , *j* , *k* } y los bordes de *E* se procesan en el siguiente orden: (*d* , *i*), (*f* , *k*), (*g* , *i*), (*b* , *g*), (*a* , *h*), (*i* , *j*), (*d* , *k*), (*b* , *j*), (*d* , *f*), (*g* , *j*), (*a* , *e*), (*i* , *d*). Enumere los vértices en cada componente conectado después de cada iteración de las líneas 3-5.

Ejercicios 21.1-2

Muestre que después de que todos los bordes son procesados por CONNECTED-COMPONENTS, dos vértices son en el mismo componente conectado si y solo si están en el mismo conjunto.

Ejercicios 21.1-3

Durante la ejecución de COMPONENTES CONECTADOS en un gráfico no dirigido $G = (V , E)$ con k componentes conectados, ¿cuántas veces se llama a FIND-SET? Cuantas veces es UNION llamado? Exprese sus respuestas en términos de $| V |$, $| E |$ y k .

[1] Cuando los bordes del gráfico son "estáticos" (no cambian con el tiempo) los componentes conectados se puede calcular más rápido utilizando la búsqueda en profundidad (ejercicio 22.3-11). A veces, sin embargo, los bordes se agregan "dinámicamente" y necesitamos mantener los componentes conectados como se agrega cada borde. En este caso, la implementación dada aquí puede ser más eficiente que ejecutar una nueva búsqueda en profundidad para cada nuevo borde.

21.2 Representación de listas enlazadas de conjuntos disjuntos

Una forma sencilla de implementar una estructura de datos de conjuntos disjuntos es representar cada conjunto mediante un enlace lista. El primer objeto de cada lista enlazada sirve como representante de su conjunto. Cada objeto en el La lista enlazada contiene un miembro del conjunto, un puntero al objeto que contiene el siguiente miembro del conjunto y un puntero de nuevo al representante. Cada lista mantiene punteros *de cabeza* , al representante, y *tail* , hasta el último objeto de la lista. La figura 21.2 (a) muestra dos conjuntos. Dentro de cada lista enlazada, el

Página 426

Los objetos pueden aparecer en cualquier orden (sujeto a nuestra suposición de que el primer objeto de cada lista es el representante).

Figura 21.2: *(a)* Representaciones de lista enlazada de dos conjuntos. Uno contiene los objetos b , c , e y h , con c como representante, y el otro contiene objetos d , f , yg , con f como el representante. Cada objeto de la lista contiene un miembro de conjunto, un puntero al siguiente objeto en la lista, y un puntero al primer objeto de la lista, que es el representante. Cada lista tiene punteros de *cabeza* y *cola* al primer y último objeto, respectivamente. *(b)* El resultado de UNIÓN (e , g). El representante del conjunto resultante es f .

Con esta representación de lista enlazada, tanto MAKE-SET como FIND-SET son fáciles, requiriendo O (1) hora. Para realizar MAKE-SET (x), creamos una nueva lista enlazada cuyo único objeto es x . por FIND-SET (x), simplemente devolvemos el puntero de x al representante.

Una simple implementación de unión

La implementación más simple de la operación UNION usando la representación de conjuntos de listas vinculadas toma mucho más tiempo que MAKE-SET o FIND-SET. Como muestra la figura 21.2 (b) , realizar UNION (x , y) añadiendo x 'lista s en el extremo de Y ' s lista. Usamos el puntero de *cola* para Y 's lista para encontrar rápidamente dónde append x ' s lista. El representante del nuevo conjunto es el elemento que originalmente era el representante del conjunto que contenía y . Desafortunadamente, debemos actualizar el puntero al representante de cada objeto originalmente en la lista de x , lo que lleva tiempo lineal en la longitud de la lista de x .

De hecho, no es difcil encontrar una secuencia de m operaciones en n objetos que requiera $\Theta(n^2)$ tiempo. Supongamos que tenemos objetos $x_1, x_2, ..., x_n$. Ejecutamos la secuencia de n MAKE-SET operaciones seguidas de n - 1 operaciones UNION mostradas en la Figura 21.3, de modo que $m = 2n - 1$. Pasamos $\Theta(n)$ tiempo realizando las n operaciones MAKE-SET. Porque la i th UNION operación actualiza i objetos, el número total de objetos actualizados por todas las n - 1 operaciones UNION es

Operación	**Numero de objetos actualizado**

MAKE-SET (x_1)	1
MAKE-SET (x_2)	1
⋮	⋮
CONFIGURACIÓN (x_n	1
UNIÓN (x_1 , x_2)	1
UNIÓN (x_2 , x_3)	2
UNIÓN (x_3 , x_4)	3
⋮	⋮
UNIÓN (x_{n-1} , x_n)	n - 1

Figura 21.3: Una secuencia de 2 n - 1 operaciones en n objetos que toma $\Theta(n^2)$ tiempo o $\Theta(n)$ tiempo por operación en promedio, utilizando la representación de conjuntos de listas vinculadas y la simple implementación de UNION.

El número total de operaciones es 2 n - 1, por lo que cada operación en promedio requiere $\Theta(n)$ tiempo. Es decir, el tiempo amortizado de una operación es $\Theta(n)$.

Una heurística de unión ponderada

En el peor de los casos, la implementación anterior del procedimiento UNION requiere un promedio de $\Theta(n)$ tiempo por llamada porque es posible que agreguemos una lista más larga a una lista más corta; debemos actualice el puntero al representante de cada miembro de la lista más larga. Supongamos en cambio que cada lista también incluye la longitud de la lista (que se mantiene fácilmente) y que Siempre agregue la lista más pequeña a la más larga, con los lazos rotos arbitrariamente. Con este simple heurística de unión ponderada, una sola operación UNION todavía puede tomar $\Omega(n)$ tiempo si ambos conjuntos tienen $\Omega(n)$ miembros. Sin embargo, como muestra el siguiente teorema, una secuencia de m MAKE-SET, Las operaciones UNION y FIND-SET, n de las cuales son operaciones MAKE-SET, toman $O(m + n \lg n)$ tiempo.

Teorema 21.1

Usando la representación de lista enlazada de conjuntos disjuntos y la heurística de unión ponderada, un secuencia de m operaciones MAKE-SET, UNION y FIND-SET, n de las cuales son MAKE-SET operaciones, toma $O(m + n \lg n)$ tiempo.

Prueba Comenzamos calculando, para cada objeto en un conjunto de tamaño n, un límite superior en el número de veces se ha actualizado el puntero del objeto al representante. Considere un fijo objeto x. Sabemos que cada vez que x 's puntero del representante se ha actualizado, x debe haber comenzado en el conjunto más pequeño. La primera vez que se actualizó el puntero representativo de x, por lo tanto, el resultado el conjunto debe haber tenido al menos 2 miembros. De manera similar, la próxima vez que el puntero representativo de x fue actualizado, el conjunto resultante debe haber tenido al menos 4 miembros. Continuando, observamos que para cualquier $k \le n$, después de que el puntero representativo de x se haya actualizado $\lceil \lg k \rceil$ veces, el conjunto resultante debe tener al menos k miembros. Dado que el conjunto más grande tiene como máximo n miembros, cada objeto El puntero representativo se ha actualizado como máximo $\lceil \lg n \rceil$ veces en todas las operaciones de UNION.

Página 428

También debemos tener en cuenta la actualización de los punteros de *cabeza* y *cola* y las longitudes de la lista, que toman sólo $\Theta(1)$ tiempo por operación UNION. El tiempo total utilizado en la actualización de los n objetos es por tanto $O(n \lg n)$.

El tiempo para la secuencia completa de m operaciones sigue fácilmente. Cada MAKE-SET y FIND-La operación SET toma $O(1)$ tiempo, y hay $O(m)$ de ellos. El tiempo total para todo La secuencia es entonces $O(m + n \lg n)$.

Ejercicios 21.2-1

Escriba el pseudocódigo para MAKE-SET, FIND-SET y UNION usando la lista vinculada representación y la heurística de la unión ponderada. Suponga que cada objeto x tiene un atributo *rep* [x] apunta al representante del conjunto que contiene x y que cada conjunto S tiene atributos *cabeza* [S], *cola* [S] y *talla* [S] (que es igual a la longitud de la lista).

Ejercicios 21.2-2

Muestre la estructura de datos que resulta y las respuestas devueltas por las operaciones FIND-SET en el siguiente programa. Utilice la representación de lista vinculada con la heurística de unión ponderada.

```
 1 para i ← 1 a 16
 2 hacer MAKE-SET ( x_i )
 3 para i ← 1 a 15 por 2
 4 hacer UNIÓN ( x_i , x_{i+1} )
 5 para i ← 1 a 13 por 4
 6 hacer UNIÓN ( x_i , x_{i+2} )
 7 UNIÓN ( x_1 , x_5 )
 8 UNIÓN ( x_11 , x_13 )
 9 UNIÓN ( x_1 , x_10 )
10 CONJUNTO DE BÚSQUEDA ( x_2 )
11 CONJUNTO DE BÚSQUEDA ( x_9 )
```

Suponga que si los conjuntos que contienen x_i y x_j tienen el mismo tamaño, entonces la operación UNION (x_i , x_j) agrega la lista de x_j a la lista de x_i .

Ejercicios 21.2-3

Adapte la prueba agregada del teorema 21.1 para obtener límites de tiempo amortizados de $O(1)$ para MAKE-SET y FIND-SET y $O(\lg n)$ para UNION usando la representación de lista enlazada y la heurística de la unión ponderada.

Página 429

Ejercicios 21.2-4

Dar un límite asintótico ajustado en el tiempo de ejecución de la secuencia de operaciones en la Figura 21.3 asumiendo la representación de lista enlazada y la heurística de unión ponderada.

Ejercicios 21.2-5

Sugerir un cambio simple en el procedimiento UNION para la representación de lista enlazada que elimina la necesidad de mantener el puntero de *cola* en el último objeto de cada lista. Sea o no el se utiliza la heurística de unión ponderada, su cambio no debe cambiar el tiempo de ejecución asintótico del procedimiento UNION. (*Sugerencia: en* lugar de agregar una lista a otra, empalme juntos.)

21.3 Bosques desunidos

En una implementación más rápida de conjuntos disjuntos, representamos conjuntos por árboles enraizados, con cada nodo que contiene un miembro y cada árbol representa un conjunto. En un bosque disjunto , ilustrado en la figura 21.4 (a) , cada miembro apunta solo a su padre. La raíz de cada árbol contiene el representante y es su propio padre. Como veremos, aunque los algoritmos sencillos que utilizan esta representación no son más rápidos que los que utilizan la representación de lista enlazada, por introduciendo dos heurísticas - "unión por rango" y "compresión de ruta" - podemos lograr el La estructura de datos de conjuntos disjuntos asintóticamente más rápida conocida.

Figura 21.4: Un bosque de conjuntos disjuntos. *(a)* Dos árboles que representan los dos conjuntos de la figura 21.2. los El árbol de la izquierda representa el conjunto { *b* , *c* , *e* , *h* }, con *c* como representante, y el árbol de la right representa el conjunto { *d* , *f* , *g* }, con *f* como representante. *(b)* El resultado de UNION (*e* , *g*).

Realizamos las tres operaciones de conjuntos disjuntos de la siguiente manera. Una operación MAKE-SET simplemente crea un árbol con un solo nodo. Realizamos una operación FIND-SET siguiendo al padre punteros hasta que encontremos la raíz del árbol. Los nodos visitados en este camino hacia la raíz constituyen el camino de búsqueda . Una operación UNION, que se muestra en la Figura 21.4 (b) , causa la raíz de una árbol para señalar la raíz del otro.

Heurísticas para mejorar el tiempo de ejecución

Hasta ahora, no hemos mejorado la implementación de la lista vinculada. Una secuencia de *n* - 1 UNIÓN las operaciones pueden crear un árbol que es solo una cadena lineal de *n* nodos. Al utilizar dos heurísticas,

Página 430

sin embargo, podemos lograr un tiempo de ejecución casi lineal en el número total de operaciones *m* .

La primera heurística, unión por rango , es similar a la heurística de unión ponderada que usamos con el representación de lista enlazada. La idea es hacer que la raíz del árbol con menos nodos apunte a la raíz del árbol con más nodos. En lugar de realizar un seguimiento explícito del tamaño de la subárbol arraigado en cada nodo, utilizaremos un enfoque que facilite el análisis. Para cada nodo, mantenemos un rango que es un límite superior en la altura del nodo. En unión por rango, el La raíz con un rango más pequeño se hace para apuntar a la raíz con un rango más grande durante una UNIÓN. operación.

La segunda heurística, la compresión de rutas , también es bastante simple y muy efectiva. Como se muestra en Figura 21.5 , la usamos durante las operaciones FIND-SET para hacer que cada nodo en el punto de ruta de búsqueda directamente a la raíz. La compresión de la ruta no cambia ningún rango.

Figura 21.5: Compresión de ruta durante la operación FIND-SET. Flechas y bucles automáticos en se omiten las raíces. *(a)* Un árbol que representa un conjunto antes de ejecutar FIND-SET (a). triangulos representan subárboles cuyas raíces son los nodos que se muestran. Cada nodo tiene un puntero a su padre. *(si)* El mismo conjunto después de ejecutar FIND-SET (a). Cada nodo en la ruta de búsqueda ahora apunta directamente a la raíz.

Pseudocódigo para bosques de conjuntos disjuntos

Para implementar un bosque de conjuntos disjuntos con la heurística de unión por rango, debemos realizar un seguimiento de rangos. Con cada nodo x , mantenemos el *rango de* valor entero [x], que es un límite superior en la altura de x (el número de aristas en el camino más largo entre x y una hoja descendiente). Cuando un conjunto singleton es creado por MAKE-SET, el rango inicial del único nodo en el el árbol correspondiente es 0. Cada operación FIND-SET deja todos los rangos sin cambios. Cuando aplicando UNION a dos árboles, hay dos casos, dependiendo de si las raíces tienen igual rango. Si las raíces tienen un rango desigual, hacemos que la raíz de rango superior sea el padre de la raíz de rango inferior, pero los rangos mismos permanecen sin cambios. Si, en cambio, las raíces tienen iguales rangos, elegimos arbitrariamente una de las raíces como padre e incrementamos su rango.

Pongamos este método en pseudocódigo. Designamos al padre del nodo x por p [x]. El enlace procedimiento, una subrutina llamada por UNION, toma punteros a dos raíces como entradas.

```
HACER SET ( x )
1 p [ x ] ← x
2 rango [ x ] ← 0

UNIÓN ( x , y )
```

Página 431

```
1 ENLACE (BUSCAR-AJUSTAR ( x ), ENCONTRAR-AJUSTAR ( y ))

ENLACE ( x , y )
1 si rango [ x ]> rango [ y ]
2 luego p [ y ] ← x
3 más p [ x ] ← y
4                   si rango [ x ] = rango [ y ]
5                         luego rango [ y ] ← rango [ y ] + 1
```

El procedimiento FIND-SET con compresión de ruta es bastante simple.

```
BUSCAR-SET ( x )
1 si x ≠ p [ x ]
2 luego p [ x ] ← FIND-SET ( p [ x ])
3 devuelve p [ x ]
```

El procedimiento FIND-SET es un método de dos pasos : hace que uno pase por la ruta de búsqueda para encontrar el root, y hace una segunda pasada por la ruta de búsqueda para actualizar cada nodo para que apunte directamente a la raíz. Cada llamada de FIND-SET (x) devuelve p [x] en la línea 3. Si x es la raíz, entonces la línea 2 no se ejecuta y se devuelve p [x] = x . Este es el caso en el que la recursividad toca fondo. De lo contrario, se ejecuta la línea 2 y la llamada recursiva con el parámetro p [x] devuelve un puntero a la raíz. La línea 2 actualiza el nodo x para que apunte directamente a la raíz, y este puntero se devuelve en línea 3.

Efecto de las heurísticas sobre el tiempo de ejecución

Por separado, la unión por rango o la compresión de la ruta mejoran el tiempo de ejecución del operaciones en bosques disjuntos, y la mejora es aún mayor cuando los dos las heurísticas se utilizan juntas. Solo, la unión por rango produce un tiempo de ejecución de $O (m \lg n)$ (ver Ejercicio 21.4-4), y este límite es estrecho (consulte el ejercicio 21.3-3). Aunque no lo probaremos aquí, si hay n operaciones MAKE-SET (y por lo tanto como máximo n - 1 operaciones UNION) yf FIND-SET, la heurística de compresión de ruta por sí sola da un tiempo de ejecución en el peor de los casos de $\Theta (n + f \cdot (1 + \log_{2+f/n} n))$.

Cuando usamos unión por rango y compresión de ruta, el peor tiempo de ejecución es $O (m \alpha (n))$, donde $\alpha (n)$ es una función de crecimiento *muy* lento, que definimos en la sección 21.4 . En cualquier aplicación concebible de una estructura de datos de conjuntos disjuntos, $\alpha (n) \leq 4$; por lo tanto, podemos ver el

tiempo de funcionamiento lineal en m en todas las situaciones prácticas. En la sección 21.4 , probamos esta superior Unido.

Ejercicios 21.3-1

Realice el ejercicio 21.2-2 utilizando un bosque de conjuntos disjuntos con unión por rango y compresión de ruta.

Ejercicios 21.3-2

Escriba una versión no recursiva de FIND-SET con compresión de ruta.

Página 432

Ejercicios 21.3-3

Dé una secuencia de m operaciones MAKE-SET, UNION y FIND-SET, n de las cuales son Operaciones MAKE-SET, que toman $\Omega (m \lg n)$ tiempo cuando usamos unión por rango solamente.

Ejercicios 21.3-4: ⋆

Muestre que cualquier secuencia de operaciones m MAKE-SET, FIND-SET y LINK, donde todos los Las operaciones LINK aparecen antes que cualquiera de las operaciones FIND-SET, toma solo $O (m)$ tiempo si ambas Se utilizan la compresión de ruta y la unión por rango. ¿Qué sucede en la misma situación si solo el ¿Se utiliza la heurística de compresión de ruta?

21.4 Análisis de unión por rango con compresión de trayectoria

Como se señaló en la Sección 21.3 , el tiempo de ejecución de la unión por rango y trayectoria combinada la heurística de compresión es $O (m \alpha (n))$ para m operaciones de conjuntos disjuntos en n elementos. En esto En la sección, examinaremos la función α para ver qué tan lentamente crece. Entonces probamos esto tiempo de ejecución utilizando el método potencial de análisis amortizado.

Una función de crecimiento muy rápido y su inversa de crecimiento muy lento

Para enteros $k \geq 0$ y $j \geq 1$, definimos la función $A_k (j)$ como

donde la expresion utiliza la notación de iteración funcional dada en la Sección 3.2. Específicamente, y para $i \geq 1$. Nos referiremos al parámetro k como el nivel de la función A .

La función $A_k (j)$ aumenta estrictamente tanto con j como con k . Para ver qué tan rápido esta función crece, primero obtenemos expresiones de forma cerrada para $A_1 (j)$ y $A_2 (j)$.

Lema 21.2

Para cualquier número entero $j \geq 1$, tenemos $A_1 (j) = 2 j + 1$.

Prueba Primero usamos la inducción en i para demostrar que . Para el caso base, tenemos . Para el paso inductivo, suponga que . Luego . Finalmente, notamos que .

Página 433

Lema 21.3

Para cualquier número entero $j \geq 1$, tenemos $A_2(j) = 2^{j+1}(j+1) - 1$.

Prueba Primero usamos la inducción en i para demostrar que . Para el caso base, tenemos . Para el paso inductivo, suponga que . Luego . Finalmente, notamos ese .

Ahora podemos ver qué tan rápido crece $A_k(j)$ simplemente examinando $A_k(1)$ para los niveles $k = 0, 1, 2, 3, 4$. De la definición de $A_0(k)$ y los lemas anteriores, tenemos $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$, y $A_2(1) = 2^{1+1} \cdot (1+1) - 1 = 7$. También tenemos

$$
\begin{aligned}
A_3(1) &= \\
&= A_2(A_2(1)) \\
&= A_2(7) \\
&= 2^8 \cdot 8 - 1 \\
&= 2^{11} - 1 \\
&= 2047
\end{aligned}
$$

y

$$
\begin{aligned}
A_4(1) &= \\
&= A_3(A_3(1)) \\
&= A_3(2047) \\
&= \\
&\quad A_2(2047) \\
&= 1^{2048} \cdot 2048 - 1 \\
&> 2^{2048} \\
&= (2^4)^{512} \\
&= 16^{512} \\
&\quad 10^{80},
\end{aligned}
$$

que es el número estimado de átomos en el universo observable.

Definimos la inversa de la función $A_k(n)$, para el entero $n \geq 0$, por

$$\alpha(n) = \text{mínimo } \{ k : A_k(1) = n \}.$$

En palabras, $\alpha(n)$ es el nivel k más bajo para el cual $A_k(1)$ es al menos n. De los valores anteriores de $A_k(1)$, vemos que

Página 434

Es solo para valores de n imprácticamente grandes (mayores que $A_4(1)$, un número enorme) que $\alpha(n) > 4$, y entonces $\alpha(n) \leq 4$ para todos los propósitos prácticos.

Propiedades de los rangos

En el resto de esta sección, probamos un límite $O(m\,\alpha(n))$ en el tiempo de ejecución del operaciones de conjuntos disjuntos con unión por rango y compresión de ruta. Para probar este límite, Primero probamos algunas propiedades simples de los rangos.

Lema 21.4

Para todos los nodos x, tenemos $rango[x] \leq rango[p[x]]$, con desigualdad estricta si $x \neq p[x]$. El valor de *el rango* $[x]$ es inicialmente 0 y aumenta con el tiempo hasta que $x \neq p[x]$; a partir de ese momento, el *rango* $[x]$ no cambio. El valor de *rango* $[p[x]]$ aumenta monótonamente con el tiempo.

Prueba La prueba es una inducción sencilla sobre el número de operaciones, utilizando el implementaciones de MAKE-SET, UNION y FIND-SET que aparecen en la Sección 21.3. Nosotros déjelo como ejercicio 21.4-1 .

Corolario 21.5

A medida que seguimos el camino desde cualquier nodo hacia una raíz, los rangos de los nodos aumentan estrictamente.

Lema 21,6

Cada nodo tiene rango como máximo $n - 1$.

Prueba El rango de cada nodo comienza en 0 y aumenta solo con las operaciones LINK. Porque allí hay como máximo $n - 1$ operaciones UNION, también hay como máximo $n - 1$ operaciones LINK. Porque cada operación LINK deja todos los rangos solos o aumenta el rango de algún nodo en 1, todos los rangos son como máximo $n - 1$.

El lema 21.6 proporciona un límite débil en los rangos. De hecho, cada nodo tiene rango como máximo $\lfloor \lg n \rfloor$ (ver Ejercicio 21.4-2). Sin embargo, el límite más flexible del Lema 21.6 será suficiente para nuestros propósitos.

Página 435

Demostrando el límite de tiempo

Usaremos el método potencial de análisis amortizado (ver la Sección 17.3) para probar la $O(m\,\alpha(n))$ límite de tiempo. Al realizar el análisis amortizado, es conveniente asumir que invocamos la operación LINK en lugar de la operación UNION. Es decir, dado que los parámetros del procedimiento LINK son punteros a dos raíces, asumimos que el FIND-SET apropiado las operaciones se realizan por separado. El siguiente lema muestra que incluso si contamos el operaciones adicionales FIND-SET inducidas por llamadas UNION, el tiempo de ejecución asintótico permanece sin alterar.

Lema 21,7

Suponga que convertimos una secuencia S 'de m ' operaciones MAKE-SET, UNION y FIND-SET en una secuencia S de m operaciones MAKE-SET, LINK y FIND-SET girando cada UNION en dos operaciones FIND-SET seguidas de un LINK. Entonces, si la secuencia S *se* ejecuta en $O(m\,\alpha(n))$ tiempo, la secuencia S 'se ejecuta en tiempo $O(m'\,\alpha(n))$.

Prueba Dado que cada operación UNION en la secuencia S 'se convierte en tres operaciones en S , tener $m' \leq m \leq 3m'$. Dado que $m = O(m')$, un $O(m \alpha(n))$ límite de tiempo para la secuencia convertida S implica un límite de tiempo $O(m' \alpha(n))$ para la secuencia original S' .

En el resto de esta sección, asumiremos que la secuencia inicial de m 'MAKE-SET, Las operaciones UNION y FIND-SET se han convertido en una secuencia de m MAKE-SET, Operaciones LINK y FIND-SET. Ahora demostramos un límite de tiempo $O(m \alpha(n))$ para el secuencia y apelar al Lema 21.7 para probar el tiempo de ejecución $O(m' \alpha(n))$ del original secuencia de operaciones m' .

Función potencial

La función potencial que usamos asigna un potencial $\varphi_q(x)$ a cada nodo x en el bosque de conjuntos disjuntos después de q operaciones. Sumamos los potenciales de nodo para el potencial de todo el bosque: $\Phi_q = \Sigma_x \varphi(x)$, donde Φ_q denota el potencial del bosque después de q operaciones. El bosque está vacío antes de la primera operación, y establecemos arbitrariamente $\Phi_0 = 0$. Ningún potencial Φ_q nunca será negativo.

El valor de $\varphi_q(x)$ depende de si x es una raíz de árbol después de la q ésima operación. Si es, o si $rango[x] = 0$, entonces $\varphi_q(x) = \alpha(n) \cdot rango[x]$.

Ahora suponga que después de la q- ésima operación, x no es una raíz y que el $rango[x] \geq 1$. Necesitamos definir dos funciones auxiliares en x antes de que podamos definir $\varphi_q(x)$. Primero definimos

$nivel(x) = \max\{k : rango[p[x]] \geq A_k(rango[x])\}$.

Es decir, nivel (x) es el mayor nivel k para el que A_k , aplicado a x 'rango s, no es mayor que x ' s rango de los padres.

Afirmamos que

Página 436

(21,1)

que vemos como sigue. Tenemos

$$\begin{aligned} rango[p[x]] &\geq rango[x] + 1 \text{ (según el Lema 21.4)} \\ &= A_0(rango[x]) \text{ (por definición de } A_0(j)), \end{aligned}$$

lo que implica que el nivel (x) ≥ 0, y tenemos

$$\begin{aligned} A_{\alpha(n)}(rango[x]) &\geq A_{\alpha(n)}(1) \text{ (porque } A_k(j) \text{ es estrictamente creciente)} \\ &\geq n \qquad \text{(por la definición de } \alpha(n)) \\ &> rango[p[x]] \text{ (según el lema 21.6),} \end{aligned}$$

lo que implica que el nivel (x) $<\alpha(n)$. Tenga en cuenta que debido a que el $rango[p[x]]$ aumenta monótonamente sobre tiempo, también lo hace el nivel (x).

La segunda función auxiliar es

Es decir, iter (x) es el mayor número de veces que podemos aplicar iterativamente $un_{nivel(x)}$, aplicado inicialmente al rango de x , antes de que obtengamos un valor mayor que el rango del padre de x .

Afirmamos que

(21,2)

que vemos como sigue. Tenemos

lo que implica que iter (x) ≥ 1, y tenemos

lo que implica que iter (x) ≤ *rango* [x]. Tenga en cuenta que debido a que el *rango* [p [x]] aumenta monótonamente con el tiempo, para que iter (x) disminuya, el nivel (x) debe aumentar. Mientras el nivel (x) permanezca sin cambios, iter (x) debe aumentar o permanecer sin cambios.

Con estas funciones auxiliares en su lugar, estamos listos para definir el potencial del nodo x después de q operaciones:

Los dos lemas siguientes dan propiedades útiles de los potenciales de los nodos.

Página 437

Lema 21,8

Para cada nodo x , y para todos los recuentos de operaciones q , tenemos

$0 \le \varphi_q(x) \le \alpha(n) \cdot rango[x]$.

Prueba Si x es una raíz o *rango* [x] = 0, entonces $\varphi_q(x) = \alpha(n) \cdot rango[x]$ por definición. Ahora suponga que x no es una raíz y ese *rango* [x] ≥ 1. Obtenemos un límite inferior en $\varphi_q(x)$ maximizando el nivel (x) e iter (x). Por el límite (21.1), nivel (x) ≤ α (n) - 1, y por el límite (21.2), iter (x) ≤ *rango* [x]. Así,

$$\begin{aligned}\varphi_q(x) &\ge (\alpha(n) - (\alpha(n) - 1)) \cdot rango[x] - rango[x] \\ &= rango[x] - rango[x] \\ &= 0.\end{aligned}$$

De manera similar, obtenemos un límite superior en $\varphi_q(x)$ minimizando el nivel (x) y el iter (x). Por el límite (21.1), nivel (x) ≥ 0, y por el límite (21.2), iter (x) ≥ 1. Por lo tanto,

$$\begin{aligned}\varphi_q(x) &\le (\alpha(n) - 0) \cdot rango[x] - 1 \\ &= \alpha(n) \cdot rango[x] - 1 \\ &< \alpha(n) \cdot rango[x].\end{aligned}$$

Cambios potenciales y costos de operación amortizados

Ahora estamos listos para examinar cómo las operaciones de conjuntos disjuntos afectan los potenciales de los nodos. Con un comprensión del cambio de potencial debido a cada operación, podemos determinar cada costo amortizado de la operación.

Lema 21,9

Sea x un nodo que no es una raíz, y suponga que la q- ésima operación es un ENLACE o FIND-SET. Luego, después de la q- ésima operación, $\varphi_q(x) \le \varphi_{q-1}(x)$. Además, si *rango* [x] ≥ 1 y level (x) o iter (x) cambia debido a la q- ésima operación, entonces $\varphi_q(x) \le \varphi_{q-1}(x) - 1$. Es decir, x 's el potencial no puede aumentar, y si tiene rango positivo y cambia el nivel (x) o el iter (x), entonces El potencial de x *se* reduce al menos en 1.

Prueba Debido a que x no es una raíz, la q º operación no cambia *el rango* [x], y debido a n hace no cambia después de las n operaciones iniciales MAKE-SET, α (n) también permanece sin cambios. Por lo tanto, estos componentes de la fórmula para el potencial de x siguen siendo los mismos después de la q- ésima operación. Si *rango* [x] = 0, entonces $\varphi_q(x) = \varphi_{q-1}(x) = 0$. Ahora suponga que *rango* [x] ≥ 1.

Recuerde que el nivel (x) aumenta monótonamente con el tiempo. Si el q nivel de las hojas de operación th (x) sin cambios, entonces iter (x) aumenta o permanece sin cambios. Si tanto el nivel (x) como el iter (x) son

Página 438

sin cambios, entonces $\varphi_q(x) = \varphi_{q-1}(x)$. Si el nivel (x) no cambia e iter (x) aumenta, entonces aumenta por al menos 1, por lo que $\varphi_q(x) \le \varphi_{q-1}(x) - 1$.

Finalmente, si la q- ésima operación aumenta el nivel (x), aumenta en al menos 1, de modo que el valor de la término $(\alpha(n) - \text{nivel}(x)) \cdot \textit{rango}[x]$ cae al menos $\textit{rango}[x]$. Debido a que el nivel (x) aumentó, el valor de iter (x) podría caer, pero de acuerdo con el límite (21.2), la caída es como máximo $\textit{rango}[x] - 1$. Por tanto, el aumento de potencial debido al cambio en el iter (x) es menor que la disminución de potencial debido al cambio de nivel (x), y concluimos que $\varphi_q(x) \le \varphi_{q-1}(x) - 1$.

Nuestros tres lemas finales muestran que el costo amortizado de cada MAKE-SET, LINK y FIND-La operación SET es $O(\alpha(n))$. Recuerde de la ecuación (17.2) que el costo amortizado de cada operación es su costo real más el aumento de potencial debido a la operación.

Lema 21.10

El costo amortizado de cada operación MAKE-SET es $O(1)$.

Prueba Suponga que la q- ésima operación es MAKE-SET (x). Esta operación crea el nodo x con rango 0, de modo que $\varphi_q(x) = 0$. Ningún otro rango o potencial cambia, por lo que $\Phi_q = \Phi_{q-1}$. Señalando que el costo real de la operación MAKE-SET es $O(1)$ completa la prueba.

Lema 21.11

El costo amortizado de cada operación LINK es $O(\alpha(n))$.

Prueba Suponga que la q- ésima operación es LINK (x , y). El costo real de la operación LINK es $O(1)$. Sin pérdida de generalidad, suponga que el ENLACE hace que y sea el padre de x .

Para determinar el cambio de potencial debido al ENLACE, observamos que los únicos nodos cuyas los potenciales que pueden cambiar son x , y , y los hijos de y justo antes de la operación. Nosotros mostraremos que el único nodo cuyo potencial puede aumentar debido al LINK es y , y que su aumento está en la mayoría de $\alpha(n)$:

- Por el Lema 21.9 , cualquier nodo que es Y 's hijo justo antes de que el enlace no puede tener su aumento potencial debido al LINK.
- De la definición de $\varphi_q(x)$, vemos que, dado que x era una raíz justo antes de q th operación, $\varphi_{q-1}(x) = \alpha(n) \cdot \textit{rango}[x]$. Si $\textit{rango}[x] = 0$, entonces $\varphi_q(x) = \varphi_{q-1}(x) = 0$. De lo contrario,

$$\begin{aligned}\varphi_q(x) &= (\alpha(n) - \text{nivel}(x)) \cdot \textit{rango}[x] - \text{iter}(x) \\ &< \alpha(n) \cdot \textit{rango}[x] \text{ (por desigualdades (21.1) y (21.2))}.\end{aligned}$$

- Como esta última cantidad es $\varphi_{q-1}(x)$, vemos que el potencial de x disminuye.

Página 439

- Como y es una raíz anterior al ENLACE, $\varphi_{q-1}(y) = \alpha(n) \cdot \textit{rango}[y]$. La operación LINK

hojas *de Y* como una raíz, y cualquiera de las hojas *y* 's rango solo o que aumenta *y* ' rango s por 1. Por lo tanto, $\varphi_q(y) = \varphi_{q-1}(y)$ o $\varphi_q(y) = \varphi_{q-1}(y) + \alpha(n)$.

El aumento de potencial debido a la operación LINK, por lo tanto, es como máximo $\alpha(n)$. El amortizado El costo de la operación LINK es $O(1) + \alpha(n) = O(\alpha(n))$.

Lema 21.12

El costo amortizado de cada operación FIND-SET es $O(\alpha(n))$.

Prueba Suponga que la *q*- ésima operación es un FIND-SET y que la ruta de búsqueda contiene *s* nodos. El costo real de la operación FIND-SET es $O(s)$. Demostraremos que el potencial de ningún nodo aumenta debido al FIND-SET y que al menos max $(0, s - (\alpha(n) + 2))$ nodos en la ruta de búsqueda tienen su potencial disminución en al menos 1.

Para ver que el potencial de ningún nodo aumenta, primero apelamos al Lema 21.9 para todos los demás nodos. que la raíz. Si *x* es la raíz, entonces su potencial es $\alpha(n) \cdot rango[x]$, que no cambia.

Ahora mostramos que al menos los nodos max $(0, s - (\alpha(n) + 2))$ tienen su potencial disminución en en al menos 1. Sea *x* un nodo en la ruta de búsqueda tal que $rango[x] > 0$ y *x* se sigue en algún lugar de la ruta de búsqueda por otro nodo *y* que no es una raíz, donde $nivel(y) = nivel(x)$ justo antes de la Operación FIND-SET. (No es necesario que el nodo *y* siga *inmediatamente* a *x* en la ruta de búsqueda). la mayoría de los nodos $\alpha(n) + 2$ en la ruta de búsqueda satisfacen estas restricciones en *x* . Los que no satisfacen ellos son el primer nodo en la ruta de búsqueda (si tiene rango 0), el último nodo en la ruta (es decir, el raíz), y el último nodo *w* en la ruta para el cual $nivel(w) = k$, para cada $k = 0, 1, 2, \ldots, \alpha(n) - 1$.

Fijemos tal nodo *x* , y demostraremos que el potencial de *x* disminuye en al menos 1. Sea $k = nivel(x) = nivel(y)$. Justo antes de la compresión de ruta causada por FIND-SET, tenemos

Poniendo estas desigualdades juntas y dejando que *i* sea el valor de $iter(x)$ antes de la compresión de la ruta, tenemos

Dado que la compresión ruta hará que *X* y *Y* tienen el mismo padre, sabemos que después de ruta compresión, $rango[p[x]] = rango[p[y]]$ y que la compresión de la ruta no disminuya $rango[p[y]]$. Dado que el $rango[x]$ no cambia, después de la compresión de la ruta tenemos que . Por lo tanto, la compresión de la ruta hará que $iter(x)$ aumente (al menos a *i*

Página 440

+ 1) o $nivel(x)$ para aumentar (lo que ocurre si $iter(x)$ aumenta al menos al $rango[x] + 1$). En cualquiera caso, por el Lema 21.9 , tenemos $\varphi_q(x) \le \varphi_{q-1}(x) - 1$. Por lo tanto, el potencial de *x* disminuye en al menos 1.

El costo amortizado de la operación FIND-SET es el costo real más el cambio de potencial. El costo real es $O(s)$, y hemos demostrado que el potencial total disminuye al menos en max $(0, s - (\alpha(n) + 2))$. El costo amortizado, por lo tanto, es como máximo $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, ya que podemos escalar las unidades de potencial para dominar la constante oculto en $O(s)$.

Al juntar los lemas precedentes se obtiene el siguiente teorema.

Teorema 21.13

Una secuencia de m operaciones MAKE-SET, UNION y FIND-SET, n de las cuales son MAKE-
Las operaciones SET se pueden realizar en un bosque de conjuntos disjuntos con unión por rango y ruta
compresión en el peor de los casos $O(m \alpha(n))$.

Prueba inmediata de los Lemas 21.7 , 21.10 , 21.11 y 21.12 .

Ejercicios 21.4-1

Demuestre el lema 21.4 .

Ejercicios 21.4-2

Demuestre que cada nodo tiene rango como máximo $\lfloor \lg n \rfloor$.

Ejercicios 21.4-3

A la luz del ejercicio 21.4-2 , ¿cuántos bits son necesarios para almacenar el $rango[x]$ para cada nodo x ?

Ejercicios 21.4-4

Página 441

Utilizando el ejercicio 21.4-2 , dé una prueba simple de que las operaciones en un bosque de conjuntos disjuntos con unión por rango pero sin compresión de trayectoria en tiempo $O(m \lg n)$.

Ejercicios 21.4-5

El profesor Dante razona que debido a que los rangos de los nodos aumentan estrictamente a lo largo de un camino hacia la raíz, los niveles de los nodos deben aumentar monótonamente a lo largo del camino. En otras palabras, si $rango(x) > 0$ y $p[x]$ no es una raíz, entonces nivel $(x) \le$ nivel $(p[x])$. ¿Tiene razón el profesor?

Ejercicios 21.4-6: ⋆

Considere la función $\alpha'(n) = \min\{k : A_k(1) \ge \lg(n+1)\}$. Demuestre que $\alpha'(n) \le 3$ para todos los valores de ny , usando el ejercicio 21.4-2 , muestre cómo modificar el argumento de función potencial para demostrar que una secuencia de m operaciones MAKE-SET, UNION y FIND-SET, n de las cuales son operaciones MAKE-SET, se pueden realizar en un bosque disjoint-set con unión por rango y compresión del trayecto en el tiempo del caso más desfavorable $O(m \alpha'(n))$.

Problemas 21-1: Mínimo fuera de línea

El **problema mínimo fuera de línea** nos pide que mantengamos un conjunto dinámico T de elementos de la dominio $\{1, 2, ..., n\}$ bajo las operaciones INSERT y EXTRACT-MIN. Se nos da un

secuencia *S* de *n* INSERT *ym* EXTRACT-MIN llamadas, donde cada tecla en {1, 2, ..., *n* } es insertado exactamente una vez. Deseamos determinar qué clave devuelve cada EXTRACT-MIN llamada. Específicamente, deseamos completar una matriz *extraída* [1 *m*], donde para *i* = 1, 2, ..., *m* , *extraído* [*i*] es la clave devuelta por la *i*- ésima llamada EXTRACT-MIN. El problema está "fuera de línea" en el sentido de que se nos permite procesar toda la secuencia *S* antes de determinar cualquiera de los llaves devueltas.

a. En la siguiente instancia del problema mínimo fuera de línea, cada INSERT es representado por un número y cada EXTRACTO-MIN está representado por la letra E:

4, 8, Mi, 3, Mi, 9, 2, 6, Mi, Mi, Mi, 1, 7, Mi, 5.

Complete los valores correctos en la matriz *extraída* .

Para desarrollar un algoritmo para este problema, dividimos la secuencia *S* en homogéneos subsecuencias. Es decir, representamos *S* por

Yo $_1$, E, Yo $_2$, E, Yo $_3$, ..., Yo $_m$, E, Yo $_{m+1}$,

Página 442

donde cada E representa una sola llamada EXTRACT-MIN y cada I $_j$ representa un (posiblemente vacío) secuencia de llamadas INSERT. Para cada subsecuencia I $_j$, inicialmente colocamos las claves insertado por estas operaciones en un conjunto K_j , que está vacío si I $_j$ está vacío. Luego hacemos el siguiendo.

```
FUERA DE LÍNEA-MÍNIMO ( m , n )
1 para i ← 1 a n
2 do determinar j tal que i                                   K j
3                 si j ≠ m + 1
4                     luego extrajo [ j ] ← i
5                             sea l el valor más pequeño mayor que j
                                                              para qué conjunto K l existe
6                             K l ← K j        K l , destruyendo K j
7 retorno extraído
```

si. Argumenta que la matriz *extraída* devuelta por OFF-LINE-MINIMUM es correcta.

C. Describir cómo implementar OFF-LINE-MINIMUM de manera eficiente con un conjunto de datos disjuntos estructura. Establezca un límite estricto en el peor tiempo de ejecución de su implementación.

Problemas 21-2: Determinación de la profundidad

En el problema de la determinación de la profundidad , mantenemos un bosque de árboles enraizados menores de tres operaciones:

- MAKE-TREE (*v*) crea un árbol cuyo único nodo es *v* .
- FIND-DEPTH (*v*) devuelve la profundidad del nodo *v* dentro de su árbol.
- GRAFT (*r* , *v*) hace que el nodo *r* , que se supone que es la raíz de un árbol, se convierta en el hijo del nodo *v* , que se supone que está en un árbol diferente de *r* pero puede o no en sí mismo ser una raíz.

a. Supongamos que usamos una representación de árbol similar a un bosque de conjuntos disjuntos: *p* [*v*] es el padre del nodo *v* , excepto que *p* [*v*] = *v* si *v* es una raíz. Si implementamos GRAFT (*r* , *v*) configurando *p* [*r*] ← *v* y FIND-DEPTH (*v*) siguiendo la búsqueda ruta hasta la raíz, devolviendo un recuento de todos los nodos que no sean *v* encontrados, muestran que el peor tiempo de ejecución de una secuencia de *m* MAKE-TREE, Las operaciones FIND-DEPTH y GRAFT son is (m_2).

Al utilizar las heurísticas de unión por rango y de compresión de ruta, podemos reducir el peor de los casos tiempo de ejecución. Usamos el bosque disjoint-set , donde cada conjunto S_i (que en sí mismo es un árbol) corresponde a un árbol T_i en el bosque. La estructura de árbol dentro de un conjunto S_i , sin embargo, no corresponde necesariamente al de T_i . De hecho, la implementación de S_i no registra la exacta relaciones padre-hijo, pero sin embargo nos permite determinar la profundidad de cualquier nodo en T_i .

La idea clave es mantener en cada nodo *v* una "pseudodistancia" *d* [*v*], que se define de manera que la suma de las pseudodistancias a lo largo del camino desde *v* hasta la raíz de su conjunto S_i es igual a la profundidad

de v en T_i . Es decir, si la ruta de v a su raíz en S_i es v_0 , v_1 , ..., v_k , donde $v_0 = v$ y v_k es S_i 's raíz, entonces la profundidad de v en T_i es .

si. Dar una implementación de MAKE-TREE.

Página 443

C. Muestre cómo modificar FIND-SET para implementar FIND-DEPTH. Tu implementación debe realizar la compresión de la ruta, y su tiempo de ejecución debe ser lineal en la longitud de la ruta de búsqueda. Asegúrese de que su implementación actualice las pseudodistancias correctamente.

re. Muestre cómo implementar GRAFT (r , v), que combina los conjuntos que contienen r y v , por modificando los procedimientos UNION y LINK. Asegúrese de que su implementación actualiza las pseudodistancias correctamente. Tenga en cuenta que la raíz de un conjunto S_i no es necesariamente la raíz del árbol correspondiente T_i .

mi. Dar un límite estricto en el peor de los casos de tiempo de ejecución de una secuencia de m MAKE-TREE, FIND-DEPTH e GRAFT, n de las cuales son operaciones MAKE-TREE.

Problemas 21-3: algoritmo de ancestros menos comunes fuera de línea de Tarjan

El antepasado menos común de dos nodos u y v en un árbol con raíz T es el nodo w que es un ancestro de ambos U y V y que tiene la mayor profundidad en T . En el menos común fuera de línea problema de antepasados , se nos da un árbol enraizado T y un conjunto arbitrario $P = \{\{ u , v \}\}$ de pares desordenados de nodos en T , y deseamos determinar el antepasado menos común de cada par en P .

Para resolver el problema de ancestros menos comunes fuera de línea, el siguiente procedimiento realiza una caminata de árbol de T con la llamada inicial LCA (*raíz* [T]). Se supone que cada nodo es de color BLANCO antes de la caminata.

```
LCA ( u )
 1 MAKE-SET ( u )
 2 antepasado [FIND-SET (u)] ← u
 3 para cada hijo v de u en T
 4 hacer LCA ( v )
 5                  UNIÓN ( u , v )
 6                  ancestro [FIND-SET ( u )] ← u
 7 colores [ u ] ← NEGRO
 8 para cada nodo v tal que { u , v } P
 9 hacer si color [ v ] = NEGRO
10                      luego imprime "El menor ancestro común de"
                                    u "y" v "es" ancestro [FIND-SET ( v )]
```

a. Argumentan que la línea 10 se ejecuta exactamente una vez para cada par { u , v } P .

si. Argumente que en el momento de la llamada LCA (u), el número de conjuntos en los datos de conjuntos disjuntos estructura es igual a la profundidad de u en T .

C. Demuestre que LCA imprime correctamente el ancestro común mínimo de u y v para cada par { u , v } P .

re. Analizar el tiempo de ejecución de LCA, asumiendo que usamos la implementación del estructura de datos de conjuntos disjuntos en la Sección 21.3.

Notas del capítulo

Muchos de los resultados importantes para las estructuras de datos de conjuntos disjuntos se deben al menos en parte a RE Tarjan. Utilizando un análisis agregado, Tarjan [290 , 292] dio el primer límite superior ajustado en términos de la inversa que crece muy lentamente de la función de Ackermann. (La función $A_k (j)$ dada

Página 444

en la sección 21.4 es similar a la función de Ackermann, y la función $\alpha(n)$ es similar a la inverso. Tanto $\alpha(n)$ como son a lo sumo 4 para todos los valores concebibles de m y n.) Una $O(m \lg^* n)$ el límite superior fue probado anteriormente por Hopcroft y Ullman [5 , 155] . El tratamiento en la sección 21.4 está adaptado de un análisis posterior de Tarjan [294] , que a su vez se basa en un análisis de Kozen [193] . Harfst y Reingold [139] dan una versión basada en el potencial de la versión anterior de Tarjan. Unido.

Tarjan y van Leeuwen [295] discuten variantes de la heurística de compresión de ruta, incluyendo "métodos de una pasada", que a veces ofrecen mejores factores constantes en su rendimiento que hacer métodos de dos pasadas. Al igual que con los análisis anteriores de Tarjan sobre la compresión de ruta básica heurístico, los análisis de Tarjan y van Leeuwen son agregados. Harfst y Reingold [139] Más tarde mostró cómo hacer un pequeño cambio en la función potencial para adaptar su trayectoria. análisis de compresión a estas variantes de un paso. Gabow y Tarjan [103] muestran que en ciertos aplicaciones, las operaciones de conjuntos disjuntos se pueden hacer para que se ejecuten en tiempo $O(m)$.

Tarjan [291] mostró que un límite inferior de se requiere tiempo para las operaciones en cualquier estructura de datos de conjuntos disjuntos que satisfacen determinadas condiciones técnicas. Este límite inferior fue posterior generalizado por Fredman y Saks [97] , quienes demostraron que en el peor de los casos, $(\lg n)$ -bit se debe acceder a las palabras de la memoria.

Parte VI: Algoritmos de gráficos

Lista de capítulos

Capítulo 22: Algoritmos de gráficos elementales
Capítulo 23: Árboles de expansión mínimos
Capítulo 24: Rutas más cortas de una sola fuente
Capítulo 25: Caminos más cortos para todos los pares
Capítulo 26: Flujo máximo

Introducción

Los gráficos son una estructura de datos omnipresente en informática y algoritmos para trabajar con son fundamentales para el campo. Hay cientos de problemas computacionales interesantes. definido en términos de gráficos. En esta parte, tocamos algunos de los más importantes.

El capítulo 22 muestra cómo podemos representar una gráfica en una computadora y luego analiza los algoritmos basado en la búsqueda de un gráfico utilizando la búsqueda primero en amplitud o la búsqueda en profundidad. Dos Se dan aplicaciones de búsqueda en profundidad: clasificación topológica de un gráfico acíclico dirigido y descomponer un gráfico dirigido en sus componentes fuertemente conectados.

El Capítulo 23 describe cómo calcular un árbol de expansión de peso mínimo de un gráfico. Tal árbol se define como la forma de menor peso de conectar todos los vértices juntos cuando cada el borde tiene un peso asociado. Los algoritmos para calcular árboles de expansión mínimos son buenos ejemplos de algoritmos codiciosos (consulte el Capítulo 16).

Los capítulos 24 y 25 consideran el problema de calcular las rutas más cortas entre vértices cuando cada borde tiene una longitud o "peso" asociado. El capítulo 24 considera el cálculo de

Página 445

caminos más cortos desde un vértice fuente dado a todos los demás vértices, y el capítulo 25 considera el cálculo de las rutas más cortas entre cada par de vértices.

Finalmente, el Capítulo 26 muestra cómo calcular un flujo máximo de material en una red. (gráfico dirigido) que tiene una fuente de material especificada, un sumidero especificado y capacidades para la cantidad de material que puede atravesar cada borde dirigido. Este general El problema surge de muchas formas, y un buen algoritmo para calcular los flujos máximos puede ser utilizado para resolver una variedad de problemas relacionados de manera eficiente.

Al describir el tiempo de ejecución de un algoritmo de gráfico en un gráfico dado $G = (V, E)$, generalmente medir el tamaño de la entrada en términos del número de vértices $|V|$ y la cantidad de aristas $|E|$ del gráfico. Es decir, hay dos parámetros relevantes que describen el tamaño de la entrada, no solo uno. Adoptamos una convención de notación común para estos parámetros. Dentro asintótico notación (como la notación O o la notación Θ), y *solo* dentro de dicha notación, el símbolo V

denota | V | y el símbolo E denota | E |. Por ejemplo, podríamos decir "el algoritmo se ejecuta en tiempo O (VE) ", lo que significa que el algoritmo se ejecuta en el tiempo O (| V || E |). Esta convención hace que fórmulas de tiempo de ejecución más fáciles de leer, sin riesgo de ambigüedad.

Otra convención que adoptamos aparece en pseudocódigo. Denotamos el conjunto de vértices de una gráfica G por V [G] y su borde establecido por E [G]. Es decir, el pseudocódigo ve los conjuntos de vértices y aristas como atributos de un gráfico.

Capítulo 22: Algoritmos de gráficos elementales

Este capítulo presenta métodos para representar un gráfico y para buscar un gráfico. Buscando un gráfico significa seguir sistemáticamente los bordes del gráfico para visitar los vértices del grafico. Un algoritmo de búsqueda de gráficos puede descubrir mucho sobre la estructura de un gráfico. Muchos Los algoritmos comienzan buscando en su gráfico de entrada para obtener esta información estructural. Otro Los algoritmos de gráficos se organizan como simples elaboraciones de algoritmos básicos de búsqueda de gráficos. Las técnicas para buscar un gráfico están en el corazón del campo de los algoritmos de gráficos.

La sección 22.1 analiza las dos representaciones computacionales de gráficos más comunes: como listas de adyacencia y como matrices de adyacencia. La sección 22.2 presenta una búsqueda de gráficos simple algoritmo llamado búsqueda de amplitud primero y muestra cómo crear un árbol de amplitud primero. Sección 22.3 presenta una búsqueda en profundidad y demuestra algunos resultados estándar sobre el orden en que la primera búsqueda visita los vértices. La sección 22.4 proporciona nuestra primera aplicación real de búsqueda en profundidad: ordenar topológicamente un gráfico acíclico dirigido. Una segunda aplicación de búsqueda en profundidad, encontrar los componentes fuertemente conectados de una gráfica dirigida, se da en la sección 22.5 .

22.1 Representaciones de gráficos

Hay dos formas estándar de representar un gráfico G = (V , E): como una colección de listas de adyacencia o como una matriz de adyacencia. Cualquiera de las dos formas es aplicable a gráficos dirigidos y no dirigidos. Por lo general, se prefiere la representación de lista de adyacencia, porque proporciona una forma compacta de representan gráficos **dispersos** , aquellos para los que | E | es mucho menor que | V | ^{2} . La mayor parte del gráfico Los algoritmos presentados en este libro asumen que un gráfico de entrada se representa en una lista de adyacencia. formar. Sin embargo, se puede preferir una representación de matriz de adyacencia cuando el gráfico es **denso** - | E | está cerca de | V | ^{2} -o cuando necesitamos poder saber rápidamente si hay una ventaja

Página 446

conectando dos vértices dados. Por ejemplo, dos de los algoritmos de rutas más cortas de todos los pares presentados en el Capítulo 25 suponen que sus gráficos de entrada están representados por matrices de adyacencia.

La **representación** de la **lista de adyacencia** de un gráfico G = (V , E) consiste en una matriz Adj de | V | liza, uno para cada vértice en V . Para cada u V , la lista de adyacencia Adj [u] contiene todos los vértices v de tal manera que hay una arista (u , v) E . Es decir, Adj [u] consta de todos los vértices adyacentes a u en G . (Alternativamente, puede contener punteros a estos vértices). Los vértices en cada lista de adyacencia normalmente se almacenan en un orden arbitrario. La figura 22.1 (b) es una representación de lista de adyacencia de el gráfico no dirigido en la Figura 22.1 (a) . De manera similar, la figura 22.2 (b) es una lista de adyacencia representación del gráfico dirigido en la figura 22.2 (a) .

Figura 22.1: Dos representaciones de un gráfico no dirigido. *(a)* Un gráfico G no dirigido que tiene cinco vértices y siete aristas. *(b)* Una lista de adyacencia representación de G . *(c)* La adyacencia-representación matricial de G .

Figura 22.2: Dos representaciones de un gráfico dirigido. *(a)* Una gráfica dirigida G *que* tiene seis vértices y ocho aristas. *(b)* Una lista de adyacencia representación de G . *(c)* La matriz de adyacencia representación de G .

Si G es un gráfico dirigido, la suma de las longitudes de todas las listas de adyacencia es $| E |$, ya que un borde de la forma (u , v) se representa haciendo que v aparezca en $Adj [u]$. Si G es un grafo no dirigido, el suma de las longitudes de todas las listas de adyacencia es 2 $| E |$, ya que si (u , v) es una arista no dirigida, entonces u aparece en la lista de adyacencia de v y viceversa. Tanto para gráficos dirigidos como no dirigidos, el La representación de lista de adyacencia tiene la propiedad deseable de que la cantidad de memoria que requiere es $\Theta (V + E)$.

Las listas de adyacencia se pueden adaptar fácilmente para representar **gráficos ponderados**, es decir, gráficos para los cuales cada borde tiene un asociado **de peso**, típicamente dada por una **función de ponderación** $w : E \rightarrow \mathbf{R}$. por Por ejemplo, sea $G = (V , E)$ una gráfica ponderada con función de ponderación w. El peso $w (u , v)$ del edge (u , v) E simplemente se almacena con el vértice v en la lista de adyacencia de u. La lista de adyacencia La representación es bastante sólida, ya que se puede modificar para admitir muchas otras variantes de gráficos.

Una posible desventaja de la representación de lista de adyacencia es que no existe una forma más rápida de determinar si una arista dada (u , v) está presente en el gráfico que buscar v en la lista de adyacencia $Adj [u]$. Esta desventaja puede remediarse mediante una representación de matriz de adyacencia del gráfico, a costa de utilizar asintóticamente más memoria. (Consulte el ejercicio 22.1-8 para obtener sugerencias de variaciones en las listas de adyacencia que permiten una búsqueda de borde más rápida).

Para la **representación de matriz de adyacencia** de un gráfico $G = (V , E)$, asumimos que los vértices están numerados 1, 2, ..., $| V |$ de alguna manera arbitraria. Entonces la representación de la matriz de adyacencia de un gráfico G consta de un $| V | \times | V |$ matriz $A = (a_{ij})$ tal que

Página 447

Las figuras 22.1 (c) y 22.2 (c) son las matrices de adyacencia de los gráficos dirigidos y no dirigidos. en las Figuras 22.1 (a) y 22.2 (a) , respectivamente. La matriz de adyacencia de un gráfico requiere $\Theta (V^2)$ memoria, independiente del número de aristas en el gráfico.

Observe la simetría a lo largo de la diagonal principal de la matriz de adyacencia en la figura 22.1 (c) . Nosotros definir la **transposición** de una matriz $A = (a_{ij})$ como la matriz dada por . Ya que en un gráfico no dirigido, (u , v) y (v , u) representan el mismo borde, la matriz de adyacencia A de un grafo no dirigido es su propia transposición: $A = A^{\mathrm{T}}$. En algunas aplicaciones, conviene almacenar solo el Entradas en y por encima de la diagonal de la matriz de adyacencia, cortando así la memoria necesaria para almacenar el gráfico casi a la mitad.

Como la representación de lista de adyacencia de un gráfico, la representación de matriz de adyacencia puede ser utilizado para gráficos ponderados. Por ejemplo, si $G = (V , E)$ es un gráfico ponderado con borde-peso función w , el peso $w (u , v)$ del borde (u , v) E simplemente se almacena como la entrada en la fila u y columna v de la matriz de adyacencia. Si no existe una arista, se puede almacenar un valor NIL como su correspondiente entrada de la matriz, aunque para muchos problemas es conveniente utilizar un valor como 0 o ∞.

Aunque la representación de la lista de adyacencia es asintóticamente al menos tan eficiente como la representación de matriz de adyacencia, la simplicidad de una matriz de adyacencia puede hacer que sea preferible cuando los gráficos son razonablemente pequeños. Además, si el gráfico no está ponderado, hay un ventaja en almacenamiento para la representación de matriz de adyacencia. En lugar de usar una palabra de memoria de la computadora para cada entrada de la matriz, la matriz de adyacencia utiliza sólo un bit por entrada.

Ejercicios 22.1-1

Dada una representación de lista de adyacencia de un grafo dirigido, ¿cuánto tiempo lleva calcular el grado de salida de cada vértice? ¿Cuánto tiempo se tarda en calcular los grados en pulgadas?

Ejercicios 22.1-2

Proporcione una representación de lista de adyacencia para un árbol binario completo en 7 vértices. Dar un representación equivalente de matriz de adyacencia. Suponga que los vértices se numeran del 1 al 7 como en un montón binario.

Ejercicios 22.1-3

La **transposición** de un gráfico dirigido $G = (V , E)$ es el gráfico $G^T = (V , E^T)$, donde $E^T = \{(v , u) : (u , v) \in E \}$. Por tanto, G^T es G con todas sus aristas invertidas. Describir algoritmos eficientes para

el cálculo de G^T de G, tanto para la adyacencia-lista y representaciones por matriz de adyacencia de G. Analiza los tiempos de ejecución de tus algoritmos.

Ejercicios 22.1-4

Dada una representación de lista de adyacencia de un multigraph $G = (V , E)$, describa un tiempo $O (V + E)$ algoritmo para calcular la representación de lista de adyacencia del grafo no dirigido "equivalente" $G' = (V , E')$, donde E' consiste en las aristas en E con todas las aristas múltiples entre dos vértices reemplazado por un solo borde y con todos los bucles automáticos eliminados.

Ejercicios 22.1-5

El **cuadrado** de una gráfica dirigida $G = (V , E)$ es la gráfica $G^2 = (V , E^2)$ tal que $(u , w) \in E^2$ si y sólo si por alguna $v \in V$, ambos $(u , v) \in E$ y $(v , w) \in E$. Es decir, G^2 contiene un borde entre u y w siempre que G contenga una ruta con exactamente dos aristas entre u y w. Describir algoritmos eficientes para calcular G^2 a partir de G tanto para la lista de adyacencia como para representaciones por matriz de adyacencia de G. Analiza los tiempos de ejecución de tus algoritmos.

Ejercicios 22.1-6

Cuando se utiliza una representación de matriz de adyacencia, la mayoría de los algoritmos de gráficos requieren tiempo $\Omega (V^2)$, pero hay algunas excepciones. Demuestre que determinar si una gráfica dirigida G contiene un **fregadero universal** -un vértice con in-grado $| V | - 1$ y fuera de grado 0: se puede determinar a tiempo $O (V)$, dada una matriz de adyacencia para G.

Ejercicios 22.1-7

La **matriz de incidencia** de un gráfico dirigido $G = (V , E)$ es a $| V | \times | E |$ matriz $B = (b_{ij})$ tal que

Describe lo que las entradas de la matriz producto BB^T representar, donde B^T es la transpuesta de B.

Ejercicios 22.1-8

Suponga que en lugar de una lista vinculada, cada entrada de la matriz $Adj[u]$ es una tabla hash que contiene el vértices v para el que $(u, v) \in E$. Si todas las búsquedas de borde son igualmente probables, ¿cuál es el esperado? ¿Es hora de determinar si hay una arista en el gráfico? ¿Qué desventajas tiene este esquema ¿tener? Sugiera una estructura de datos alternativa para cada lista de bordes que resuelva estos problemas. Hace ¿Tu alternativa tiene desventajas en comparación con la tabla hash?

22.2 Búsqueda en amplitud primero

La **búsqueda en amplitud** es uno de los algoritmos más simples para buscar un gráfico y el arquetipo para muchos algoritmos de gráficos importantes. El algoritmo de árbol de expansión mínimo de Prim (Sección 23.2) y el algoritmo de rutas más cortas de fuente única de Dijkstra (Sección 24.3) usan ideas similares a aquellos en búsqueda de amplitud primero.

Dado un gráfico $G = (V, E)$ y un vértice de **origen** distinguido s, búsqueda de amplitud primero explora sistemáticamente las aristas de G para "descubrir" todos los vértices que se pueden alcanzar desde s. Eso calcula la distancia (el número más pequeño de aristas) desde s hasta cada vértice alcanzable. También produce un "árbol de amplitud primero" con root s que contiene todos los vértices alcanzables. Para cualquier vértice v accesible desde s, el camino en el árbol en amplitud de s a v corresponde a un "camino más corto" de s a v en G, es decir, una ruta que contiene el menor número de bordes. El algoritmo funciona tanto en gráficos dirigidos como no dirigidos.

La búsqueda en amplitud primero se llama así porque expande la frontera entre lo descubierto y lo vértices no descubiertos uniformemente a lo largo de la frontera. Es decir, el algoritmo descubre todos los vértices a una distancia k de s antes de descubrir cualquier vértice a una distancia $k + 1$.

Para realizar un seguimiento del progreso, la búsqueda en amplitud colorea cada vértice en blanco, gris o negro. Todas los vértices comienzan en blanco y luego pueden volverse grises y luego negros. Se **descubre** un vértice el la primera vez que se encuentra durante la búsqueda, momento en el que se vuelve no blanco. Gris y vértices negros, por lo tanto, se han descubierto, pero la búsqueda primero en amplitud distingue entre ellos para asegurarse de que la búsqueda proceda de una manera amplia. Si $(u, v) \in E$ y el vértice u es negro, entonces el vértice v es gris o negro; es decir, todos los vértices adyacentes a los vértices negros tienen sido descubierto. Los vértices grises pueden tener algunos vértices blancos adyacentes; ellos representan el frontera entre vértices descubiertos y no descubiertos.

La búsqueda en amplitud primero construye un árbol en amplitud que inicialmente contiene solo su raíz, que es el vértice fuente s. Siempre que se descubre un vértice blanco v en el curso de la exploración lista de adyacencia de un vértice u ya descubierto , el vértice v y el borde (u, v) se agregan a el árbol. Decimos que u es el **predecesor** o **padre** de v en el árbol del primer ancho. Desde un vértice se descubre como máximo una vez, tiene como máximo un padre. Relaciones de antepasados y descendientes en el árbol primero en anchura se define en relación con la raíz s como de costumbre: si u está en una ruta en el árbol desde la raíz s al vértice v , entonces u es un antepasado de v y v es un descendiente de u .

El procedimiento BFS de búsqueda primero en amplitud asume que el gráfico de entrada $G = (V, E)$ es representados mediante listas de adyacencia. Mantiene varias estructuras de datos adicionales con cada vértice en el gráfico. El color de cada vértice $u \in V$ se almacena en la variable $color[u]$, y el predecesor de u se almacena en la variable $\pi[u]$. Si u no tiene antecesor (por ejemplo, si $u = s$ o u no se ha descubierto), entonces $\pi[u] = \text{NIL}$. La distancia desde la fuente s al vértice u

Página 450

calculado por el algoritmo se almacena en $d[u]$. El algoritmo también utiliza una cola de primero en entrar, primero en salir Q (consulte la Sección 10.1) para gestionar el conjunto de vértices grises.

```
BFS ( G , s )
 1 para cada vértice u                    V [ G ] - { s }
 2 hacer color [ u ] ← BLANCO
 3                d [ u ] ← ∞
 4                π [ u ] ← NULO
 5 colores [ s ] ← GRIS
 6 d [ s ] ← 0
 7 π [ s ] ← NULO
 8 Q ← Ø
 9 ENQUEUE ( Q , s )
10 mientras Q ≠ Ø
```

```
11 ¿ u ← DEQUEUE ( Q )
12              para cada v          Adj [ u ]
13                   hacer si color [ v ] = BLANCO
14                             luego color [ v ] ← GRIS
15                                  d [ v ] ← d [ u ] + 1
dieciséis                           π [ v ] ← u
17                                  ENQUEUE ( Q , v )
18              color [ u ] ← NEGRO
```

La figura 22.3 ilustra el progreso de BFS en un gráfico de muestra.

Figura 22.3: El funcionamiento de BFS en un gráfico no dirigido. Los bordes de los árboles se muestran sombreados como son producidos por BFS. Dentro de cada vértice u se muestra d [u]. La cola Q se muestra en el comienzo de cada iteración del *tiempo* de bucle de las líneas 10-18. Las distancias de los vértices se muestran a continuación a los vértices de la cola.

El procedimiento BFS funciona de la siguiente manera. Las líneas 1-4 pintan todos los vértices de blanco, establezca d [u] como infinito para cada vértice u , y establece el padre de cada vértice en NIL. La línea 5 pinta la fuente vértice s gris, ya que se considera descubierto cuando comienza el procedimiento. Línea 6 inicializa d [s] a 0, y la línea 7 establece que el predecesor de la fuente sea NIL. Las líneas 8-9 se inicializan Q a la cola que contiene solo los vértices s .

Página 451

El **tiempo** de bucle de las líneas 10-18 itera el tiempo que quedan vértices grises, que son vértices descubiertos que aún no han examinado completamente sus listas de adyacencia. Este **mientras** bucle mantiene el siguiente invariante:

- En la prueba de la línea 10, la cola Q consta del conjunto de vértices grises.

Aunque no usaremos este ciclo invariante para probar la exactitud, es fácil ver que se cumple antes de la primera iteración y que cada iteración del ciclo mantiene la invariante. Antes de la primera iteración, el único vértice gris y el único vértice en Q , es el vértice fuente s . Línea 11 determina el vértice gris T a la cabeza de la cola Q y lo elimina de Q . El **para** El bucle de las líneas 12-17 considera cada vértice v en la lista de adyacencia de u . Si v es blanco, entonces tiene aún no se ha descubierto, y el algoritmo lo descubre ejecutando las líneas 14-17. Es primero en gris, y su distancia d [v] se establece en d [u] +1. Entonces, u se registra como su padre. Finalmente, es colocado en la cola de la cola Q . Cuando todos los vértices de la lista de adyacencia de u han sido examinado, u está ennegrecido en las líneas 11-18. El ciclo invariante se mantiene porque siempre que un El vértice está pintado de gris (en la línea 14) también está en cola (en la línea 17), y siempre que un vértice es quitado de la cola (en la línea 11) también está pintado de negro (en la línea 18).

Los resultados de la búsqueda en amplitud pueden depender del orden en que los vecinos de un vértice dado se visitan en la línea 12: el árbol primero en anchura puede variar, pero las distancias d calculado por el algoritmo no lo hará. (Vea el ejercicio 22.2-4 .)

Análisis

Antes de probar las diversas propiedades de la búsqueda de amplitud primero, abordamos la algo más fácil trabajo de analizar su tiempo de ejecución en un gráfico de entrada $G = (V , E)$. Usamos análisis agregado, como

vimos en la Sección 17.1. Después de la inicialización, ningún vértice se blanquea nunca, y por lo tanto la prueba en la línea 13 asegura que cada vértice se pone en cola como máximo una vez y, por lo tanto, se retira de la cola como máximo una vez. Las operaciones de poner en cola y sacar de cola toman $O(1)$ tiempo, por lo que el tiempo total dedicado a la cola operaciones es $O(V)$. Debido a que la lista de adyacencia de cada vértice se escanea solo cuando el vértice se quita de la cola, cada lista de adyacencia se escanea como máximo una vez. Dado que la suma de las longitudes de todos las listas de adyacencia es $\Theta(E)$, el tiempo total empleado en la exploración de listas de adyacencia es $O(E)$. los la sobrecarga para la inicialización es $O(V)$ y, por lo tanto, el tiempo de ejecución total de BFS es $O(V + E)$. Así, primero en amplitud carreras de búsqueda en tiempo lineal en el tamaño de la representación de adyacencia lista de G.

Caminos más cortos

Al principio de esta sección, afirmamos que la búsqueda primero en amplitud encuentra la distancia a cada vértice alcanzable en un gráfico $G = (V, E)$ desde una fuente vértice dado $s \in V$. Defina el *más corto camino distancia* $\delta(s, v)$ a partir de s a v como el número mínimo de bordes en cualquier trayectoria desde el vértice s a vértice v; Si no hay camino de s a v, entonces $\delta(s, v) = \infty$. Un camino de longitud $\delta(s, v)$ de s a v es dice que es un *camino más corto*[1] a partir de s a v. Antes de mostrar la búsqueda en amplitud calcula las distancias de la ruta más corta, investigamos una propiedad importante de la ruta más corta distancias.

Lema 22.1

Sea $G = (V, E)$ una gráfica dirigida o no dirigida, y sea $s \in V$ un vértice arbitrario. Luego, para cualquier borde $(u, v) \in E$,

Página 452

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Prueba Sise puede acceder a u desde s, también lo es v. En este caso, la ruta más corta desde s a v no puede ser más largo que el camino más corto desde s a u seguido por el borde (u, v), y por lo tanto la desigualdad sostiene. Si no se puede acceder a u desde s, entonces $\delta(s, u) = \infty$, y la desigualdad se mantiene.

Queremos mostrar que BFS calcula correctamente $d[v] = \text{delta}(s, v)$ para cada vértice $v \in V$. Nosotros primero demuestre que $d[v]$ limita $\delta(s, v)$ desde arriba.

Lema 22.2

Sea $G = (V, E)$ un grafo dirigido o no dirigido, y suponga que BFS se ejecuta en G desde un dado fuente vértice $s \in V$. Luego, al terminar, para cada vértice $v \in V$, el valor $d[v]$ calculado por BFS satisface $d[v] \geq \delta(s, v)$.

Prueba Utilizamos la inducción en el número de operaciones ENQUEUE. Nuestra hipótesis inductiva es que $d[v] \geq \delta(s, v)$ para todo $v \in V$.

La base de la inducción es la situación inmediatamente después de que s se ponga en cola en la línea 9 de BFS. La hipótesis inductiva se cumple aquí, porque $d[s] = 0 = \delta(s, s)$ y $d[v] = \infty \geq \delta(s, v)$ para todo $v \in V - \{s\}$.

Para el paso inductivo, considere un vértice blanco v que se descubre durante la búsqueda de un vértice u. La hipótesis inductiva implica que $d[u] \geq \delta(s, u)$. De la asignación realizada por la línea 15 y del Lema 22.1, obtenemos

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v). \end{aligned}$$

A continuación, el vértice v se pone en cola y nunca se vuelve a poner en cola porque también está atenuado y **entonces la** cláusula de las líneas 14-17 se ejecuta solo para los vértices blancos. Por tanto, el valor de $d[v]$ nunca cambia de nuevo, y se mantiene la hipótesis inductiva.

Para probar que $d[v] = \delta(s, v)$, tenemos que demostrar primero con mayor precisión cómo la cola Q opera durante el curso de BFS. El siguiente lema muestra que en todo momento, hay como máximo dos valores d distintos en la cola.

Lema 22.3

Página 453

Suponga que durante la ejecución de BFS en un gráfico $G = (V, E)$, la cola Q contiene el vértices $v_1, v_2, ..., v_r$, donde v_1 es la cabeza de Q y v_r es la cola. Entonces, $d[v_r] \le d[v_1] + 1$ y $d[v_i] \le d[v_{i+1}]$ para $i = 1, 2, ..., r - 1$.

Prueba La prueba es por inducción sobre el número de operaciones en cola. Inicialmente, cuando la cola contiene sólo s, el lema ciertamente es válido.

Para el paso inductivo, debemos probar que el lema se cumple después de quitar la cola y poner en cola un vértice. Si se quita la cola de la cabecera v_1 de la cola, v_2 se convierte en la nueva cabecera. (Si el la cola se vacía, entonces el lema se mantiene vacío.) Según la hipótesis inductiva, $d[v_1] \le d[v_2]$. Pero entonces tenemos $d[v_r] \le d[v_1] + 1 \le d[v_2] + 1$, y las desigualdades restantes son inafectado. Por tanto, el lema sigue con v_2 como cabecera.

Poner en cola un vértice requiere un examen más detenido del código. Cuando ponemos en cola un vértice v en línea 17 de BFS, se convierte en v_{r+1}. En ese momento, ya hemos eliminado el vértice u, cuyo La lista de adyacencia se está escaneando actualmente, de la cola Q, y por la hipótesis inductiva, la nueva cabeza v_1 tiene $d[v_1] \ge d[u]$. Por lo tanto, $d[v_{r+1}] = d[v] = d[u] + 1 \le d[v_1] + 1$. De la hipótesis inductiva, también tenemos $d[v_r] \le d[u] + 1$, y entonces $d[v_r] \le d[u] + 1 = d[v] = d[v_{r+1}]$, y las desigualdades restantes no se ven afectadas. Por tanto, el lema sigue cuando v se pone en cola.

El siguiente corolario muestra que los valores d en el momento en que los vértices se ponen en cola son aumentando monótonamente con el tiempo.

Corolario 22.4

Suponga que los vértices v_i y v_j se ponen en cola durante la ejecución de BFS, y que v_i es en cola antes de v_j. Entonces $d[v_i] \le d[v_j]$ en el momento en que v_j se pone en cola.

Prueba inmediata del Lema 22.3 y la propiedad de que cada vértice recibe unvalor d finito como máximo una vez durante el curso de BFS.

Ahora podemos probar que la búsqueda primero en amplitud encuentra correctamente las distancias del camino más corto.

Teorema 22.5: (Corrección de la búsqueda primero en amplitud)

Sea $G = (V, E)$ un grafo dirigido o no dirigido, y suponga que BFS se ejecuta en G desde un dado fuente vértice $s \in V$. Luego, durante su ejecución, BFS descubre cada vértice $v \in V$ que es accesible desde la fuente s, y a la terminación, $d[v] = \delta(s, v)$ para todos los $v \in V$. Además, para cualquier vértice $v \ne s$ que es accesible desde s, una de las rutas más cortas desde s a v es un más corto trayectoria desde s a $pi[v]$ seguido por el borde $(\pi[v], v)$.

Página 454

Prueba Suponga, con el propósito de contradicción, que algún vértice recibe unvalor d no igual a su distancia de camino más corta. Sea v el vértice con mínimo $\delta (s , v)$ que recibe tal valor d incorrecto ; claramente $v \neq s$. Por el Lema 22.2 , $d [v] \geq \delta (s , v)$, y así tenemos que $d [v]>$ $\delta (s , v)$. El vértice v debe ser accesible desde s , porque si no lo es, entonces $\delta (s , v) = \infty \geq d [v]$. Vamos u ser el vértice inmediatamente anterior v en una ruta más corta desde s a v , de modo que $\delta (s , v) = \delta (s , u) + 1$. Debido a que $\delta (s , u) <\delta (s , v)$, y debido a cómo elegimos v , tenemos $d [u] = \delta (s , u)$. Poniendo estos propiedades juntas, tenemos

(22,1)

Ahora considere el momento en que BFS elige quitar el vértice u de Q en la línea 11. En este momento, el vértice v es blanco, gris o negro. Demostraremos que en cada uno de estos casos, derivamos una contradicción con la desigualdad (22.1) . Si v es blanco, entonces la línea 15 establece $d [v] = d [u] + 1$, lo que contradice desigualdad (22,1) . Si v es negro, entonces ya se eliminó de la cola y, por corolario 22.4 , tenemos $d [v] \leq d [u]$, contradiciendo nuevamente la desigualdad (22.1) . Si v es gris, entonces fue pintado de gris al quitar la cola de algún vértice w , que se eliminó de Q antes que u y para lo cual $d [v] = d [w] + 1$. Sin embargo, según el Corolario 22.4 , $d [w] \leq d [u]$, por lo que tenemos $d [v] \leq d [u] + 1$, contradiciendo una vez más la desigualdad (22.1) .

Por lo tanto concluimos que $d [v] =$ delta (s , v) para todo $v V$. Todos los vértices accesibles desde s deben ser descubierto, porque si no lo fueran, tendrían valores d infinitos . Para concluir la prueba de el teorema, observe que si $\pi [v] = u$, entonces $d [v] = d [u] + 1$. Así, podemos obtener un camino más corto de s a v mediante la adopción de una ruta más corta desde s a $\pi [v]$ y después de atravesar el borde $(\pi [v], v)$.

Árboles de ancho primero

El procedimiento BFS construye un árbol de amplitud primero a medida que busca en el gráfico, como se ilustra en la Figura 22.3 . El árbol está representado por el campo π en cada vértice. Más formalmente, para un gráfico $G = (V , E)$ con la fuente s , definimos el subgrafo predecesor de G como $G_\pi = (V_\pi , E_\pi)$, donde

$V_\pi = \{ v V : \pi [v] \neq \text{NIL} \} \{ s \}$

y

$E_\pi = \{ (\pi [v], v) : v V_\pi - \{ s \} \}$.

El subgrafo predecesor G_π es un árbol de amplitud primero si V_π consiste en los vértices alcanzables de s y, para todos $v V_\pi$, hay un camino simple único de s a v en G_π que es también un más corto trayectoria desde s a v en G . Un árbol en anchura es de hecho un árbol, ya que está conectado y $| E_\pi | = | V_\pi | -$ 1 (ver teorema B.2). Las aristas de E_π se denominan aristas de árbol .

Una vez que se ha ejecutado BFS desde una fuente s en un gráfico G , el siguiente lema muestra que el El subgrafo predecesor es un árbol de amplitud primero.

Lema 22.6

Página 455

Cuando se aplica a un gráfico dirigido o no dirigido $G = (V , E)$, el procedimiento BFS construye π *de* modo que el subgrafo predecesor $G_\pi = (V_\pi , E_\pi)$ es un árbol de ancho primero.

La línea de prueba 16 de BFS establece $\pi [v] = u$ si y solo si $(uv) E$ y $\delta (s , v) <\infty$- es decir, si v es alcanzable desde s -y así V_π consiste en los vértices en V alcanzables desde s . Dado que G_π forma una árbol, según el Teorema B.2 , contiene una ruta única desde sa cada vértice en V_π . Aplicando Teorema 22.5 De manera inductiva, llegamos a la conclusión de que cada uno de esos caminos es el más corto.

Los siguientes impresiones procedimiento fuera los vértices en una ruta más corta desde s a v , suponiendo que Ya se ha ejecutado BFS para calcular el árbol de ruta más corta.

```
CAMINO DE IMPRESIÓN ( G , s , v )
1 si v = s
2 luego imprima s
3 más si π [ v ] = NIL
4                   luego imprima "no existe ruta de" s "a" v "
5                   más PRINT-PATH ( G , s , π [ v] )
6                         imprimir v
```

Este procedimiento se ejecuta en el tiempo de forma lineal en el número de vértices en la ruta impresa, ya que cada La llamada recursiva es para una ruta un vértice más corta.

Ejercicios 22.2-1

Mostrar los d e pi valores que dan lugar a la ejecución de la búsqueda en amplitud en el grafo dirigido de Figura 22.2 (a) , utilizando el vértice 3 como fuente.

Ejercicios 22.2-2

Mostrar los d e pi valores que dan lugar a la ejecución de la búsqueda en amplitud en el grafo no dirigido de la figura 22.3 , utilizando el vértice u como fuente.

Ejercicios 22.2-3

¿Cuál es el tiempo de ejecución de BFS si su gráfico de entrada está representado por una matriz de adyacencia y ¿Se modifica el algoritmo para manejar esta forma de entrada?

Ejercicios 22.2-4

Página 456

Argumente que en una búsqueda en amplitud, el valor $d [u]$ asignado a un vértice u es independiente del orden en el que se dan los vértices en cada lista de adyacencia. Usando la Figura 22.3 como ejemplo, muestran que el árbol de amplitud primero calculado por BFS puede depender del orden dentro listas de adyacencia.

Ejercicios 22.2-5

Da un ejemplo de una gráfica dirigida $G = (V , E)$, un vértice fuente $s \ V$ y un conjunto de aristas de árbol $E_\pi \quad E$ tal que para cada vértice $v \ V$, el camino único en la gráfica (V , E_π) de s a v es una ruta más corta en G , sin embargo, el conjunto de aristas E_π no se puede producir ejecutando BFS en G , sin importar cómo se ordenan los vértices en cada lista de adyacencia.

Ejercicios 22.2-6

Hay dos tipos de luchadores profesionales: "buenos" y "malos". Entre cualquier par de los luchadores profesionales, puede haber o no una rivalidad. Supongamos que tenemos n profesionales luchadores y tenemos una lista de r parejas de luchadores por los que existen rivalidades. Dar una $O (n + r)$ algoritmo de tiempo que determina si es posible designar a algunos de los luchadores como

buenos y el resto como malos, de modo que cada rivalidad es entre un buen y un chico malo. Si es posible realizar tal designación, su algoritmo debería producirla.

Ejercicios 22.2-7: ⋆

El diámetro de un árbol $T = (V , E)$ viene dado por

es decir, el diámetro es la mayor de todas las distancias de camino más corto en el árbol. Dar un eficiente algoritmo para calcular el diámetro de un árbol y analizar el tiempo de ejecución de su algoritmo.

Ejercicios 22.2-8

Sea $G = (V , E)$ un gráfico no dirigido y conectado. Dar un algoritmo de tiempo $O (V + E)$ para calcular un camino en G que atraviesa cada borde en E exactamente una vez en cada dirección. Describe cómo puedes encuentra la salida de un laberinto si te dan una gran cantidad de centavos.

Página 457

[1] En los capítulos 24 y 25 , generalizaremos nuestro estudio de los caminos más cortos a los gráficos ponderados, en donde cada borde tiene un peso con valor real y el peso de una ruta es la suma de los pesos de sus bordes constituyentes. Los gráficos considerados en el presente capítulo no están ponderados o, de manera equivalente, todos los bordes tienen peso unitario.

22.3 Búsqueda en profundidad primero

La estrategia seguida por la búsqueda en profundidad es, como su nombre lo indica, buscar "más profundo" en el gráfico siempre que sea posible. En la búsqueda en profundidad, se exploran los bordes de los más recientes Descubrí el vértice v que aún tiene bordes inexplorados dejándolo. Cuando todos los bordes de v se hayan explorado, la búsqueda "preclasificación" para explorar los bordes dejando el vértice de la cual v se descubierto. Este proceso continúa hasta que hayamos descubierto todos los vértices que son alcanzables. desde el vértice de la fuente original. Si quedan vértices sin descubrir, entonces uno de ellos es seleccionado como una nueva fuente y la búsqueda se repite desde esa fuente. Todo este proceso es repetido hasta que se descubran todos los vértices.

Al igual que en la búsqueda en amplitud, siempre que se descubre un vértice v durante una exploración de la adyacencia lista de un vértice u ya descubierto , la búsqueda en profundidad registra este evento estableciendo v 's campo predecesor $\pi [v]$ a u . A diferencia de la búsqueda en amplitud, cuyo subgrafo predecesor forma un árbol, el subgrafo predecesor producido por una búsqueda en profundidad puede estar compuesto de varios árboles, porque la búsqueda puede repetirse desde múltiples fuentes. [2] El subgrafo predecesor de una búsqueda en profundidad se define, por lo tanto, de forma ligeramente diferente a la de una búsqueda en búsqueda: dejamos $G_\pi = (V , E_\pi)$, donde

$E_\pi = \{(\pi [v], v): v \ V \text{ y } \pi [v] \neq \text{NIL}\}$.

El subgrafo predecesor de una búsqueda en profundidad forma un bosque en profundidad compuesto de varios árboles de profundidad . Las aristas de E_π se denominan aristas de árbol .

Como en la búsqueda en amplitud, los vértices se colorean durante la búsqueda para indicar su estado. Cada El vértice es inicialmente blanco, aparece en gris cuando se descubre en la búsqueda y ennegrecido cuando está terminado , es decir, cuando su lista de adyacencia ha sido examinado por completo. Esta tecnica garantiza que cada vértice termina en exactamente un árbol de profundidad primero, de modo que estos árboles son desarticular.

Además de crear un bosque en profundidad, la búsqueda en profundidad también marca el tiempo de cada vértice. Cada

El vértice v tiene dos marcas de tiempo: la primera marca de tiempo $d[v]$ registra cuándo se descubre v *por* primera vez (y gris), y la segunda marca de tiempo $f[v]$ registra cuándo finaliza la búsqueda de examinar v 's lista de adyacencia (y ennegrece v). Estas marcas de tiempo se utilizan en muchos algoritmos de gráficos y son generalmente útil para razonar sobre el comportamiento de la búsqueda en profundidad.

El procedimiento DFS a continuación registra cuándo descubre el vértice u en la variable $d[u]$ y cuándo termina el vértice u en la variable $f[u]$. Estas marcas de tiempo son números enteros entre 1 y 2 $|V|$, desde hay un evento de descubrimiento y un evento final para cada uno de los $|V|$ vértices. Para cada vértice u ,

(22,2)

Página 458

El vértice u es BLANCO antes del tiempo $d[u]$, GRIS entre el tiempo $d[u]$ y el tiempo $f[u]$, y NEGRO después de eso.

El siguiente pseudocódigo es el algoritmo básico de búsqueda en profundidad primero. El gráfico de entrada G puede ser no dirigido o dirigido. La variable *tiempo* es una variable global que usamos para la marca de tiempo.

```
DFS ( G )
1 para cada vértice u                    V [ G ]
2 hacer color [ u ] ← BLANCO
3                  π [ u ] ← NULO
4 veces ← 0
5 para cada vértice u                    V [ G ]
6 hacer si color [ u ] = BLANCO
7                        luego DFS-VISIT ( u )
DFS-VISIT ( u )
1 color [ u ] ← GRIS  ▸ El vértice u blanco acaba de ser descubierto.
2 tiempo ← tiempo +1
3 d [ u ] tiempo
4 por cada v                Adj [ u ]  ▸ Explorar borde ( u , v ).
5 hacer si color [ v ] = BLANCO
6                        entonces π [ v ] ← u
7                                    DFS-VISIT ( v )
8 colores [ u ] NEGRO  ▸  Ennegrezca u ; esta terminado.
9 f [ u ]  ▸  tiempo ← tiempo +1
```

La figura 22.4 ilustra el progreso de DFS en el gráfico que se muestra en la figura 22.2 .

Figura 22.4: El progreso del algoritmo de búsqueda en profundidad DFS en un gráfico dirigido. Como los bordes son explorados por el algoritmo, se muestran sombreados (si son bordes de árboles) o discontinua (de lo contrario). Los bordes que no son árboles se etiquetan como B, C o F según sean bordes traseros, cruzados o delanteros. Los vértices tienen una marca de tiempo por tiempo de descubrimiento / tiempo de finalización.

El procedimiento DFS funciona de la siguiente manera. Las líneas 1-3 pintan todos los vértices de blanco e inicializan sus campos π a NIL. La línea 4 restablece el contador de tiempo global. Las líneas 5-7 verifican cada vértice en V por turno y, cuando se encuentra un vértice blanco, visítelo usando DFS-VISIT. Cada vez que se llama a DFS-VISIT (u) línea 7, el vértice u *se* convierte en la raíz de un nuevo árbol en el bosque de profundidad. Cuando regrese DFS, a cada vértice u se le ha asignado un tiempo de descubrimiento $d[u]$ y un tiempo de finalización $f[u]$.

Página 459

En cada llamada DFS-VISIT (*u*), el vértice *u* es inicialmente blanco. La línea 1 pinta *u* gris, la línea 2 se incrementa la variable global *tiempo* , y la línea 3 registra el nuevo valor de *tiempo* como tiempo de descubrimiento *d* [*u*]. Las líneas 4-7 examinan cada vértice *v* adyacente a *u* y visitan de forma recursiva *v* si es blanco. Como cada vértice *v Adj* [*u*] se considera en la línea 4, decimos que el borde *(u, v)* es explorado por la profundidad primero buscar. Finalmente, después de explorar cada borde que sale de *u* , las líneas 8-9 pintan *u de* negro y registre el tiempo de finalización en *f* [*u*].

Tenga en cuenta que los resultados de la búsqueda en profundidad pueden depender del orden en que se encuentran los vértices. examinados en la línea 5 de DFS, y en el orden en que se visitan los vecinos de un vértice en la línea 4 de DFS-VISIT. Estas diferentes órdenes de visita tienden a no causar problemas en práctica, ya que *cualquier* resultado de búsqueda en profundidad se puede utilizar de forma eficaz, con resultados equivalentes.

¿Cuál es el tiempo de ejecución de DFS? Los bucles en las líneas 1-3 y 5-7 de DFS toman tiempo Θ (*V*), excluyendo el tiempo para ejecutar las llamadas a DFS-VISIT. Como hicimos para la búsqueda en amplitud, utilizamos el análisis agregado. El procedimiento DFS-VISIT se llama exactamente una vez para cada vértice *v* *V* , ya que DFS-VISIT se invoca solo en vértices blancos y lo primero que hace es pintar el vértice gris. Durante una ejecución de DFS-VISIT *(v)* , se ejecuta el bucle en las líneas 4-7 | *Adj* [*v*] | veces. Ya que

el costo total de ejecutar las líneas 4-7 de DFS-VISIT es Θ (*E*). El tiempo de ejecución de DFS es por lo tanto Θ (*V* + *E*).

Propiedades de la búsqueda en profundidad primero

La búsqueda en profundidad proporciona información valiosa sobre la estructura de un gráfico. Quizás el más La propiedad básica de la búsqueda en profundidad es que el subgrafo predecesor $G_{\pi de}$ hecho forma un bosque de árboles, ya que la estructura de los árboles de profundidad refleja exactamente la estructura de llamadas recursivas de DFS-VISIT. Es decir, $u = \pi$ [*v*] si y solo si se llamó a DFS-VISIT (*v*) durante una búsqueda de la lista de adyacencia de *u* . Además, el vértice *v* es un descendiente del vértice *u* en la profundidad primer bosque si y solo si *v* se descubre durante el tiempo en que *u* es gris.

Otra propiedad importante de la búsqueda en profundidad es que los tiempos de descubrimiento y finalización han estructura de paréntesis . Si representamos el descubrimiento del vértice *u* con un paréntesis izquierdo "(*u* " y representar su finalización con un paréntesis derecho " *u*)", luego la historia de descubrimientos y Los acabados tienen una expresión bien formada en el sentido de que los paréntesis están correctamente anidado. Por ejemplo, la búsqueda en profundidad de la Figura 22.5 (a) corresponde a la el paréntesis que se muestra en la Figura 22.5 (b) . Otra forma de enunciar la condición del paréntesis La estructura se da en el siguiente teorema.

Página 460

Figura 22.5: Propiedades de la búsqueda en profundidad. *(a)* El resultado de una búsqueda en profundidad de un grafico. Los vértices tienen una marca de tiempo y los tipos de aristas se indican como en la Figura 22.4. *(b)* Intervalos para el tiempo de descubrimiento y el tiempo de finalización de cada vértice corresponden al paréntesis mostrado. Cada rectángulo abarca el intervalo dado por los tiempos de descubrimiento y finalización del vértice correspondiente. Se muestran los bordes de los árboles. Si dos intervalos se superponen, entonces uno está anidado dentro del otro, y el vértice correspondiente al intervalo más pequeño es un descendiente del vértice correspondiente al mayor. *(c)* La gráfica del inciso (a) se redibujó con todos los árboles y hacia adelante los bordes van hacia abajo dentro de un árbol de profundidad primero y todos los bordes posteriores suben de un descendiente a un ancestro.

Teorema 22.7: (teorema del paréntesis)

En cualquier búsqueda en profundidad de un gráfico (dirigido o no dirigido) $G = (V , E)$, para dos vértices u y v , exactamente una de las tres condiciones siguientes sostiene:

- los intervalos $[d [u], f [u]]$ y $[d [v], f [v]]$ son completamente disjuntos, y ni u ni v son descendiente del otro en el bosque de profundidad,
- el intervalo $[d [u], f [u]]$ está contenido completamente dentro del intervalo $[d [v], f [v]]$, yu es un descendiente de v en un árbol de profundidad, o
- el intervalo $[d [v], f [v]]$ está contenido completamente dentro del intervalo $[d [u], f [u]]$, y v es un descendiente de u en un árbol de profundidad primero.

Prueba Comenzamos con el caso en el que $d [u] < d [v]$. Hay dos subcampos a considerar, según $d [v] < f [u]$ o no. El primer subcaso ocurre cuando $d [v] < f [u]$, entonces v fue mientras descubierto u todavía era gris. Esto implica que v es descendiente de u . Además, dado que v

Página 461

se descubrió más recientemente que u , se exploran todos sus bordes salientes y v está terminado, antes de que la búsqueda vuelva y termine u . En este caso, por lo tanto, el intervalo $[d [v], f [v]]$ es contenido enteramente dentro del intervalo $[d [u], f [u]]$. En el otro subcaso, $f [u] < d [v]$, y La desigualdad (22.2) implica que los intervalos $[d [u], f [u]]$ y $[d [v], f [v]]$ son disjuntos. Porque el Los intervalos son inconexos, ninguno de los vértices se descubrió mientras que el otro era gris, por lo que vértice es descendiente del otro.

El caso en el que $d [v] < d [u]$ es similar, con los roles de u y v invertidos en el anterior argumento.

Corolario 22.8: (Anidación de intervalos de descendientes)

El vértice v es un descendiente propio del vértice u en el bosque de profundidad primero para a (dirigido o no dirigido) grafica G si y solo si $d [u] < d [v] < f [v] < f [u]$.

Prueba inmediata del teorema 22.7 .

El siguiente teorema da otra caracterización importante de cuándo un vértice es descendiente de otro en el bosque de profundidad.

Teorema 22.9: (teorema del camino blanco)

En un bosque de profundidad primero de un gráfico (dirigido o no dirigido) $G = (V , E)$, el vértice v es un descendiente del vértice u si y solo si en el momento $d [u]$ en que la búsqueda descubre u , se puede alcanzar el vértice v de u *a lo* largo de un camino que consta completamente de vértices blancos.

Prueba : suponga que v es descendiente de u . Sea w cualquier vértice en el camino entre u y v en el árbol de profundidad primero, por lo que w es un descendiente de u . Según el Corolario 22.8 , $d [u] < d [w]$, y entonces w es blanco en el momento $d [u]$.

⇐: Suponga que el vértice v es accesible desde u *a lo* largo de una trayectoria de vértices blancos en el tiempo $d [u]$, pero v no se convierte en descendiente de u en el árbol de profundidad. Sin pérdida de generalidad, asuma que todos los demás vértices a lo largo del camino se vuelven descendientes de u . (De lo contrario, sea v el vértice más cercano a u *a lo* largo de la ruta que no se convierte en un descendiente de u .) Sea w el predecesor de v en la ruta, de modo que w es un descendiente de u (w y u pueden de hecho ser lo mismo vértice) y, según el Corolario 22.8 , $f [w] \le f [u]$. Tenga en cuenta que v debe descubrirse después de que u sea descubierto, pero antes de que w finalice. Por lo tanto, $d [u] < d [v] < f [w] \le f [u]$. Teorema 22.7 entonces implica que el intervalo $[d [v], f [v]]$ está contenido completamente dentro del intervalo $[d [u], f [u]]$. Por Corolario 22.8 , después de todo , v debe ser descendiente de u .

Página 462

Clasificación de aristas

Otra propiedad interesante de la búsqueda en profundidad es que la búsqueda se puede utilizar para clasificar bordes del gráfico de entrada $G = (V , E)$. Esta clasificación de bordes se puede utilizar para obtener importantes información sobre un gráfico. Por ejemplo, en la siguiente sección , veremos que un gráfico dirigido es acíclico si y solo si una búsqueda en profundidad no produce bordes "posteriores" (Lema 22.11).

Podemos definir cuatro tipos de bordes en términos del bosque de profundidad primero G_π producido por un bosque de profundidad primero buscar en G .

1. Los bordes de los árboles son los bordes del bosque de profundidad primero G_π . Edge (u , v) es un borde de árbol si v fue primero descubierto explorando edge (u , v).
2. Los bordes posteriores son los bordes (u , v) que conectan un vértice u con un antepasado v en una profundidad. primer árbol. Los bucles automáticos, que pueden ocurrir en gráficos dirigidos, se consideran regresivos. bordes.
3. Las aristas hacia adelante son aquellas aristas que no son árboles (u , v) que conectan un vértice u con un descendiente v en un árbol de profundidad primero.
4. Los bordes transversales son todos los demás bordes. Pueden ir entre vértices en la misma profundidad primero árbol, siempre que un vértice no sea un antepasado del otro, o pueden ir entre vértices en diferentes árboles de profundidad primero.

En las Figuras 22.4 y 22.5 , los bordes están etiquetados para indicar su tipo. La figura 22.5 (c) también muestra cómo se puede volver a dibujar la gráfica de la Figura 22.5 (a) para que todos los árboles y los bordes delanteros hacia abajo en un árbol de profundidad primero y todos los bordes posteriores suben. Cualquier gráfico se puede volver a dibujar en este Moda.

El algoritmo DFS se puede modificar para clasificar los bordes a medida que los encuentra. La idea clave es que cada arista (u , v) se puede clasificar por el color del vértice v que se alcanza cuando el primero se explora el borde (excepto que los bordes delantero y transversal no se distinguen):

1. BLANCO indica el borde de un árbol,
2. GRIS indica un borde posterior y
3. NEGRO indica un borde delantero o transversal.

El primer caso es inmediato a partir de la especificación del algoritmo. Para el segundo caso, Observe que los vértices grises siempre forman una cadena lineal de descendientes correspondientes a la pila de invocaciones DFS-VISIT activas; el número de vértices grises es uno más que el

profundidad en el bosque de profundidad primero del vértice descubierto más recientemente. Exploración siempre procede del vértice gris más profundo, por lo que un borde que alcanza otro vértice gris alcanza un antepasado. El tercer caso maneja la posibilidad restante; se puede demostrar que tal ventaja (u , v) es un borde delantero si d [u] < d [v] y un borde transversal si d [u]> d [v]. (Vea el ejercicio 22.3-4 .)

En un gráfico no dirigido, puede haber alguna ambigüedad en la clasificación de tipos, ya que (u , v) y (v , u) son realmente el mismo borde. En tal caso, el borde se clasifica como el *primer* tipo en el lista de clasificación que se aplica. De manera equivalente (vea el ejercicio 22.3-5), el borde se clasifica de acuerdo a cualquiera de (u , v) o (v , u) que se encuentre primero durante la ejecución del algoritmo.

Ahora mostramos que los bordes delantero y transversal nunca ocurren en una búsqueda en profundidad de un gráfico no dirigido.

Página 463

Teorema 22.10

En una búsqueda en profundidad de un gráfico G no dirigido , cada borde de G es un borde de árbol o un borde trasero.

Demostración Sea (u , v) una arista arbitraria de G , y suponga sin pérdida de generalidad que d [u] < d [v]. Entonces, v debe ser descubierto y terminado antes de que terminemos u (mientras que u es gris), ya que v es en la lista de adyacencia de u . Si la arista (u , v) se explora primero en la dirección de u a v , entonces v es sin descubrir (blanco) hasta ese momento, porque de lo contrario ya habríamos explorado este borde en la dirección de v a u . Por tanto, (u , v) se convierte en el borde de un árbol. Si (u , v) se explora primero en el dirección de v a u , entonces (u , v) es un borde posterior, ya que u todavía es gris en el momento en que el borde es el primero explorado.

Veremos varias aplicaciones de estos teoremas en las siguientes secciones.

Ejercicios 22.3-1

Haga un gráfico de 3 por 3 con etiquetas de fila y columna BLANCO, GRIS y NEGRO. En cada celda (i , j), indique si, en cualquier punto durante una búsqueda en profundidad de un gráfico dirigido, puede ser una arista desde un vértice de color i hasta un vértice de color j . Para cada posible borde, indique qué tipos de bordes que puede ser. Haga un segundo gráfico de este tipo para la búsqueda en profundidad de un gráfico no dirigido.

Ejercicios 22.3-2

Muestre cómo funciona la búsqueda en profundidad en el gráfico de la Figura 22.6 . Suponga que el bucle **for** de Las líneas 5-7 del procedimiento DFS consideran los vértices en orden alfabético y suponen que cada lista de adyacencia está ordenada alfabéticamente. Muestre los tiempos de descubrimiento y finalización de cada vértice y mostrar la clasificación de cada borde.

Figura 22.6: Una gráfica dirigida para usar en los Ejercicios 22.3-2 y 22.5-2.

Ejercicios 22.3-3

Muestre la estructura de paréntesis de la búsqueda en profundidad que se muestra en la Figura 22.4 .

Página 464

Ejercicios 22.3-4

Muestre que la arista (u , v) es

1. el borde de un árbol o el borde delantero si y solo si $d[u] < d[v] < f[v] < f[u]$,
2. un borde posterior si y solo si $d[v] < d[u] < f[u] < f[v]$, y
3. un borde transversal si y solo si $d[v] < f[v] < d[u] < f[u]$.

Ejercicios 22.3-5

Demuestre que en un gráfico no dirigido, clasificar un borde (u , v) como un borde de árbol o un borde posterior según si (u , v) o (v , u) se encuentra primero durante la búsqueda en profundidad es equivalente a clasificarlo según la prioridad de tipos en el esquema de clasificación.

Ejercicios 22.3-6

Vuelva a escribir el procedimiento DFS, usando una pila para eliminar la recursividad.

Ejercicios 22.3-7

Da un contraejemplo a la conjetura de que si hay una ruta de u a v en una gráfica dirigida G , y si $d[u] < d[v]$ en una búsqueda en profundidad de G , entonces v es un descendiente de u en la búsqueda en profundidad primero bosque producido.

Ejercicios 22.3-8

Da un contraejemplo a la conjetura de que si hay una ruta de u a v en una gráfica dirigida G , entonces cualquier búsqueda en profundidad debe resultar en $d[v] \leq f[u]$.

Ejercicios 22.3-9

Página 465

Modifique el pseudocódigo para la búsqueda en profundidad para que imprima todos los bordes en la dirección gráfico G , junto con su tipo. Muestre qué modificaciones, si las hay, deben hacerse si G es no dirigido.

Ejercicios 22.3-10

Explica cómo un vértice u de una gráfica dirigida puede terminar en un árbol de profundidad que contiene solo u , a pesar de que u tiene ambos bordes entrantes y salientes en G .

Ejercicios 22.3-11

Demuestre que se puede utilizar una búsqueda en profundidad de un gráfico no dirigido G para identificar el componentes de G , y que el bosque de profundidad contiene tantos árboles como G ha conectado componentes. Más precisamente, muestre cómo modificar la búsqueda en profundidad para que cada vértice v sea asignó una etiqueta entera cc [v] entre 1 y k , donde k es el número de componentes de G , tales que cc [u] = cc [v] si y solo si u y v están en el mismo componente.

Ejercicios 22.3-12: ⋆

Un gráfico dirigido G = (V , E) está conectado individualmente si implica que hay como máximo uno camino simple de u a v para todos los vértices u , v V . Dar un algoritmo eficiente para determinar si un gráfico dirigido está conectado individualmente o no.

[2] Puede parecer arbitrario que la búsqueda en amplitud se limite a una sola fuente mientras que en profundidad La primera búsqueda puede buscar desde múltiples fuentes. Aunque conceptualmente, la búsqueda en amplitud podría proceder de múltiples fuentes y la búsqueda en profundidad podría limitarse a una fuente, nuestro El enfoque refleja cómo se utilizan normalmente los resultados de estas búsquedas. La búsqueda en amplitud es generalmente empleado para encontrar distancias de camino más cortas (y el subgrafo predecesor asociado) de una fuente determinada. La búsqueda en profundidad es a menudo una subrutina en otro algoritmo, como veremos ver más adelante en este capítulo.

22.4 Orden topológico

Esta sección muestra cómo se puede utilizar la búsqueda en profundidad para realizar un tipo topológico de grafo acíclico dirigido, o un "dag" como a veces se le llama. Una especie topológica de dag G = (V , E) es un ordenamiento lineal de todos sus vértices tal que si G contiene una arista (u , v), entonces u

Página 466

aparece antes de v en el pedido. (Si el gráfico no es acíclico, entonces no hay ordenamiento lineal posible.) Un tipo topológico de un gráfico puede verse como un orden de sus vértices a lo largo de un línea horizontal para que todos los bordes dirigidos vayan de izquierda a derecha. Por tanto, la clasificación topológica diferente del tipo habitual de "clasificación" estudiado en la Parte II .

Los gráficos acíclicos dirigidos se utilizan en muchas aplicaciones para indicar las precedencias entre eventos. La figura 22.7 da un ejemplo que surge cuando el profesor Bumstead se viste con el Mañana. El profesor debe ponerse ciertas prendas antes que otras (por ejemplo, calcetines antes que zapatos). Otros artículos se pueden poner en cualquier orden (por ejemplo, calcetines y pantalones). Un borde dirigido (u , v) en el dag de la Figura 22.7 (a) indica que la prenda u debe ponerse antes que la v . UNA Por lo tanto, este dag topológico da una orden para vestirse. La figura 22.7 (b) muestra el dag ordenado topológicamente como un orden de vértices a lo largo de una línea horizontal tal que todos los bordes dirigidos van de izquierda a derecha.

Figura 22.7: *(a) El* profesor Bumstead clasifica topológicamente su ropa cuando se viste.
Cada borde dirigido (*u, v*) significa que la prenda *u* debe ponerse antes que la *v* . los
Los tiempos de descubrimiento y finalización de una búsqueda en profundidad se muestran junto a cada vértice. *(b)* El mismo gráfico mostrado topológicamente ordenado. Sus vértices están dispuestos de izquierda a derecha en orden de disminución del tiempo de finalización. Tenga en cuenta que todos los bordes dirigidos van de izquierda a derecha.

El siguiente algoritmo simple ordena topológicamente un dag.

CLASE TOPOLÓGICO (*G*)
1 llame a DFS (*G*) para calcular los tiempos de finalización *f* [*v*] para cada vértice *v*
2 cuando termine cada vértice, insértelo al frente de una lista vinculada
3 **devuelve** la lista enlazada de vértices

La figura 22.7 (b) muestra cómo aparecen los vértices ordenados topológicamente en orden inverso a su tiempos de finalización.

Podemos realizar una ordenación topológica en el tiempo $\Theta (V + E)$, ya que la búsqueda en profundidad toma $\Theta (V + E)$ tiempo y se necesita $O (1)$ tiempo para insertar cada uno de los $| V |$ vértices en el frente de la lista vinculada.

Demostramos la exactitud de este algoritmo utilizando el siguiente lema clave que caracteriza Gráficos acíclicos dirigidos.

Lema 22.11

Un gráfico dirigido *G* es acíclico si y solo si una búsqueda en profundidad de *G* no produce bordes posteriores.

Página 467

Prueba : Suponga que hay un borde posterior (*u* , *v*). Entonces, el vértice *v* es un antepasado del vértice *u* en el bosque de profundidad primero. Por lo tanto, hay un camino de *v* a *u* en *G* , y el borde posterior (*u* , *v*) completa un ciclo.

⇐: Suponga que *G* contiene un ciclo *c* . Mostramos que una búsqueda en profundidad de *G* produce una borde. Sea *v* el primer vértice que se descubrirá en *c* , y sea (*u* , *v*) la arista precedente en *c* .
En el tiempo *d* [*v*], los vértices de *c* forman un camino de vértices blancos de *v* a *u* . Por el camino blanco teorema, el vértice *u se* convierte en un descendiente de *v* en el bosque de profundidad primero. Por lo tanto, (*u* , *v*) es un borde trasero.

Teorema 22.12

TOPOLÓGICA-ORDENAR (*G*) produce una especie topológico de un grafo acíclico dirigido *G* .

Prueba Suponga que DFS se ejecuta en un dag dado $G = (V , E)$ para determinar los tiempos de finalización de su vértices. Basta mostrar que para cualquier par de vértices distintos $u , v \in V$, si hay una arista en *G* de *u* a *v* , luego $f [v] < f [u]$. Considere cualquier borde (*u* , *v*) explorado por DFS (*G*). Cuando este borde explorado, *v* no puede ser gris, ya que entonces *v* sería un ancestro de *u* y (*u* , *v*) sería un borde posterior, contradiciendo el Lema 22.11 . Por lo tanto, *v* debe ser blanco o negro. Si *v* es blanco, se convierte en descendiente de *u* , por lo que $f [v] < f [u]$. Si *v* es negro, ya ha sido terminado, por lo que $f [v]$ ya se ha configurado. Debido a que todavía estamos explorando desde *ti* , todavía tenemos para asignar una marca de tiempo a $f [u]$, y una vez que lo hagamos, también tendremos $f [v] < f [u]$. Por lo tanto, para cualquier borde (*u* , *v*) en el dag, tenemos $f [v] < f [u]$, lo que demuestra el teorema.

Ejercicios 22.4-1

Muestre el orden de los vértices producido por TOPOLOGICAL-SORT cuando se ejecuta en el dag de la figura 22.8 , bajo el supuesto del ejercicio 22.3-2 .

Figura 22.8: Un dag para clasificación topológica.

Ejercicios 22.4-2

Proporcione un algoritmo de tiempo lineal que tome como entrada un gráfico acíclico dirigido $G = (V , E)$ y dos vértices s y t , y devuelve el número de caminos de s a t en G . Por ejemplo, en el dirigido

Página 468

gráfico acíclico de la figura 22.8 , hay exactamente cuatro caminos desde el vértice p al vértice v : pov , por yv , $posr$ yv y psr yv . (Su algoritmo solo necesita contar las rutas, no enumerarlas).

Ejercicios 22.4-3

Dar un algoritmo que determine si un gráfico no dirigido dado $G = (V , E)$ contiene un ciclo. Su algoritmo debe ejecutarse en tiempo $O (V)$, independientemente de $| E |$.

Ejercicios 22.4-4

Demuestre o refute: si un gráfico dirigido G contiene ciclos, entonces TOPOLOGICAL-SORT (G) produce un ordenamiento de vértices que minimiza el número de bordes "malos" que son inconsistentes con el pedido producido.

Ejercicios 22.4-5

Otra forma de realizar la clasificación topológica en un gráfico acíclico dirigido $G = (V , E)$ es encontrar repetidamente un vértice de grado 0, generarlo y eliminarlo y todos sus bordes salientes del gráfico. Explique cómo implementar esta idea para que se ejecute en el tiempo $O (V + E)$. Qué ¿Qué pasa con este algoritmo si G tiene ciclos?

22.5 Componentes fuertemente conectados

Ahora consideramos una aplicación clásica de búsqueda en profundidad: descomponer un gráfico dirigido en sus componentes fuertemente conectados. Esta sección muestra cómo hacer esta descomposición utilizando dos búsquedas en profundidad. Muchos algoritmos que funcionan con gráficos dirigidos comienzan con tal descomposición. Después de la descomposición, el algoritmo se ejecuta por separado en cada componente conectado. Luego, las soluciones se combinan de acuerdo con la estructura de conexiones entre componentes.

Recuerde del Apéndice B que un componente fuertemente conectado de una gráfica dirigida $G = (V , E)$ es un conjunto máximo de vértices CV tal que para cada par de vértices u y v en C , tenemos ambos y ; que es, vértices u y v son accesibles unos de otros. Figura 22.9 muestra un ejemplo.

Página 469

Figura 22.9: *(a)* un grafo dirigido G . Los componentes fuertemente conectados de G se muestran como regiones sombreadas. Cada vértice está etiquetado con sus tiempos de descubrimiento y finalización. Los bordes de los árboles son sombreado. *(b)* El gráfico G^T , la transpuesta de G . El bosque de profundidad primero calculado en la línea 3 de Se muestra COMPONENTES FUERTEMENTE CONECTADOS, con los bordes de los árboles sombreados. Cada componente fuertemente conectado corresponde a un árbol de profundidad primero. Vértices *b, c, g* y *h* , que están muy sombreados, son las raíces de los árboles de profundidad primero producidos por la profundidad de primero buscar de G^T . *(c)* El gráfico de componentes acíclicos G^{SCC} obtenido al contraer todos los bordes dentro cada componente fuertemente conectado de G *de* modo que solo un vértice permanece en cada componente.

Nuestro algoritmo para encontrar componentes fuertemente conectados de un gráfico $G = (V, E)$ usa el transpuesta de G , que se define en el ejercicio 22.1-3 como la gráfica $G^T = (V, E^T)$, donde $E^T = \{(u, v) : (v, u) \in E\}$. Es decir, E^T consta de los bordes de G con sus direcciones invertidas. Dado una representación de lista de adyacencia de G , el tiempo para crear G^T es $O(V + E)$. Es interesante observar que G y G^T tiene exactamente los mismos componentes fuertemente conectados: u y v son accesible desde uno al otro en G si y sólo si son accesible desde uno al otro en G^T . Figura 22.9 (b) muestra la transposición de la gráfica en la figura 22.9 (a) , con la fuertemente conectada componentes sombreados.

El siguiente algoritmo de tiempo lineal (es decir, $\Theta(V + E)$ -tiempo) calcula los valores fuertemente conectados componentes de un gráfico dirigido $G = (V, E)$ utilizando dos búsquedas en profundidad, una en G y otra en G^T .

```
COMPONENTES FUERTEMENTE CONECTADOS ( G )
1 llame a DFS ( G ) para calcular los tiempos de finalización f [ u ] para cada vértice u
2 calcular G^T
3 llame a DFS ( G^T ), pero en el bucle principal de DFS, considere los vértices
                en orden decreciente f [ u ] (calculado en la línea 1)
4 generar los vértices de cada árbol en el bosque de profundidad primero formado en
línea 3 como
                componente separado fuertemente conectado
```

La idea detrás de este algoritmo proviene de una propiedad clave del gráfico de componentes $G^{SCC} = (V^{SCC}, E^{SCC})$, que definimos de la siguiente manera. Suponga que G tiene componentes fuertemente conectados $C_1, C_2, ..., C_k$. El conjunto de vértices V^{SCC} es $\{v_1, v_2, ..., v_k\}$, y contiene un vértice v_i para cada fuerte

Página 470

conectado componente C_i de G . Hay una arista $(v_i, v_j) \in E^{SCC}$ si G contiene una arista dirigida (x, y) para algunos $x \in C_i$ y algunos $y \in C_j$. Mirado de otra manera, contrayendo todos los bordes cuyos vértices incidentes están dentro del mismo componente fuertemente conectado de G , el resultado

gráfico es G^{SCC} . La figura 22.9 (c) muestra el gráfico de componentes del gráfico de la figura 22.9 (a) .

La propiedad clave es que el gráfico de componentes es un dag, lo que implica el siguiente lema.

Lema 22.13

Sean C y C' componentes fuertemente conectados distintos en el gráfico dirigido $G = (V, E)$, sean u , v C , sea u', $v' C'$, y suponga que hay un camino en G . Entonces tampoco puede haber un camino en G .

Prueba si hay un camino en G , entonces hay caminos y en G . Por lo tanto, u y v 'son accesibles desde uno al otro, contradiciendo de este modo la suposición de que C y C 'son componentes distintos fuertemente conectados.

Veremos que al considerar los vértices en la segunda búsqueda en profundidad primero en orden decreciente de los tiempos de finalización que se calcularon en la primera búsqueda en profundidad, estamos, en esencia, visitando los vértices del grafo de componentes (cada uno de los cuales corresponde a un componente conectado de G) en orden topológico.

Debido a que COMPONENTES FUERTEMENTE CONECTADOS realiza dos búsquedas en profundidad, existe el potencial de ambigüedad cuando discutimos $d[u]$ o $f[u]$. En esta sección, estos valores Siempre consulte los tiempos de descubrimiento y finalización calculados por la primera llamada de DFS, en línea 1.

Extendemos la notación para tiempos de descubrimiento y finalización a conjuntos de vértices. Si es $U V$, entonces definimos $d(U) = \min_{u U} \{d[u]\}$ y $f(U) = \max_{u U} \{f[u]\}$. Es decir, $d(U)$ y $f(U)$ son los tiempo más temprano descubrimiento y la última hora de finalización, respectivamente, de cualquier vértice en U .

El siguiente lema y su corolario dan una propiedad clave que relaciona fuertemente componentes y tiempos de finalización en la primera búsqueda en profundidad.

Lema 22.14

Sean C y C' componentes fuertemente conectados distintos en el gráfico dirigido $G = (V, E)$. Suponer que hay una arista $(u, v) E$, donde $u C$ y $v C'$. Entonces $f(C) > f(C')$.

Prueba Hay dos casos, dependiendo de qué componente fuertemente conectado, C o C', haya el primer vértice descubierto durante la búsqueda en profundidad.

Si $d(C) < d(C')$, deja x ser el primer vértice descubierto en C . En el tiempo $d[x]$, todos los vértices en C y C' son blancas. Hay un camino en G desde x hasta cada vértice en C *que* consta solo de vértices blancos. Como $(u, v) E$, para cualquier vértice $w C'$, también hay una ruta en el tiempo $d[x]$ desde x hasta w en G

Página 471

que consta solo de vértices blancos: . Según el teorema del camino blanco, todos los vértices en C y C' se convierten en descendientes de x en el primer árbol de profundidad. Según el corolario 22.8 , $f[x] = f(C) > f(C')$.

Si en cambio tenemos $d(C) > d(C')$, sea y el primer vértice descubierto en C'. En el momento $d[y]$, todos Los vértices en C' son blancos y hay un camino en G desde y hasta cada vértice en C' que consta solo de vértices blancos. Según el teorema del camino blanco, todos los vértices de C' se convierten en descendientes de y en el árbol de profundidad primero, y por el Corolario 22.8 , $f[y] = f(C')$. En el momento $d[y]$, todos los vértices de C son blancos. Dado que hay una arista (u, v) de C a C', el lema 22.13 implica que no puede haber una ruta de C 'a C . Por tanto, no se puede llegar a ningún vértice en C desde y . En el tiempo $f[y]$, por lo tanto, todos los vértices en C todavía son blancos. Entonces, para cualquier vértice $w C$, tenemos $f[w] > f[y]$, lo que implica que $f(C) > f(C')$.

El siguiente corolario nos dice que cada borde en G^{T} que va entre diferentes fuertemente componentes conectados va de un componente con un tiempo de acabado anterior (en el primer búsqueda en profundidad) a un componente con un tiempo de finalización posterior.

Corolario 22.15

Sean C y C' componentes fuertemente conectados distintos en el gráfico dirigido $G = (V, E)$. Suponer que hay una arista $(u, v) \in E^T$, donde $u \in C$ y $v \in C'$. Entonces $f(C) < f(C')$.

Prueba Puesto que $(u, v) \in E^T$, tenemos $(v, u) \in E$. Dado que los componentes fuertemente conectados de G y G^T son iguales, el lema 22.14 implica que $f(C) < f(C')$.

El corolario 22.15 proporciona la clave para comprender por qué los CONECTADOS FUERTEMENTE El procedimiento COMPONENTS funciona. Examinemos lo que sucede cuando realizamos la segunda búsqueda en profundidad, que está en G^T. Comenzamos con el componente C fuertemente conectado cuyo tiempo de finalización $f(C)$ es máximo. La búsqueda comienza desde algún vértice $x \in C$, y visita todos los vértices en C. Según el Corolario 22.15, no hay aristas en G^T de C a ninguna otra fuertemente componente conectado, por lo que la búsqueda de x no visitará vértices en ningún otro componente. Por lo tanto, el árbol con raíz en x contiene exactamente los vértices de C. Habiendo completado la visita a todos vértices en C, la búsqueda en la línea 3 selecciona como raíz un vértice de algún otro fuertemente conectado componente C' cuya acabado tiempo $f(C')$ es máxima sobre todos los componentes distintos de C. Nuevamente, la búsqueda visitará todos los vértices en C', pero según el Corolario 22.15, las únicas aristas en G^T de C' a cualquier otro componente debe ser a C, que ya hemos visitado. En general, cuando la búsqueda en profundidad de G^T en la línea 3 visita cualquier componente fuertemente conectado, cualquier los bordes de ese componente deben ser para componentes que ya fueron visitados. Cada profundidad primero árbol, por lo tanto, será exactamente un componente fuertemente conectado. El siguiente teorema formaliza este argumento.

Teorema 22.16

Página 472

COMPONENTES FUERTEMENTE CONECTADOS (G) calcula correctamente los componentes de un grafo dirigido G.

Prueba Argumentamos por inducción sobre el número de árboles de profundidad primero encontrados en la búsqueda de profundidad primero de G^T en la línea 3 que los vértices de cada árbol forman un componente fuertemente conectado. los La hipótesis inductiva es que los primeros k árboles producidos en la línea 3 están fuertemente conectados componentes. La base de la inducción, cuando $k = 0$, es trivial.

En el paso inductivo, asumimos que cada uno de los primeros k árboles de profundidad primero producidos en la línea 3 es un componente fuertemente conectado, y consideramos el $(k + 1)$ st árbol producido. Deja que la raíz de este árbol sea el vértice u, y sea u *la* componente C fuertemente conectada. Por como nosotros elija raíces en la búsqueda en profundidad en la línea 3, $f[u] = f(C) > f(C')$ para cualquier fuertemente conectado componente C' distinto de C que aún no se ha visitado. Por la hipótesis inductiva, en el momento que la búsqueda visita u, todos los demás vértices de C son blancos. Por lo tanto, según el teorema del camino blanco, todos los demás vértices de C son descendientes de u en su primer árbol en profundidad. Además, por el inductivo hipótesis y por el Corolario 22.15, cualquier arista en G^T que deje C debe ser fuertemente componentes conectados que ya han sido visitados. Por lo tanto, ningún vértice en ningún El componente conectado distinto de C será un descendiente de u durante la búsqueda en profundidad de G^T. Por lo tanto, los vértices del primer árbol en profundidad en G^T que tiene sus raíces en u forman exactamente uno fuertemente componente conectado, que completa el paso inductivo y la prueba.

Aquí hay otra forma de ver cómo funciona la segunda búsqueda en profundidad. Considera el gráfico de componente $(G^T)^{SCC}$ de G^T. Si mapeamos cada componente fuertemente conectado visitado en el segunda búsqueda en profundidad primero a un vértice de $(G^T)^{SCC}$, los vértices de $(G^T)^{SCC}$ se visitan en el inversa de un orden ordenado topológicamente. Si invertimos los bordes de $(G^T)^{SCC}$, obtenemos el gráfico $((G^T)^{SCC})^T$. Como $((G^T)^{SCC})^T = G^{SCC}$ (vea el ejercicio 22.5-4), la segunda búsqueda en profundidad visita los vértices de G^{SCC} en orden topológico.

Ejercicios 22.5-1

¿Cómo puede cambiar el número de componentes fuertemente conectados de un gráfico si se crea un nuevo borde? ¿adicional?

Ejercicios 22.5-2

Muestre cómo funciona el procedimiento COMPONENTES FUERTEMENTE CONECTADOS en el gráfico de la Figura 22.6 . Específicamente, muestre los tiempos de finalización calculados en la línea 1 y el bosque producido en la línea 3. Suponga que el ciclo de las líneas 5-7 de DFS considera vértices en orden alfabético orden y que las listas de adyacencia están en orden alfabético.

Ejercicios 22.5-3

Página 473

El profesor Deaver afirma que el algoritmo para componentes fuertemente conectados puede ser simplificado mediante el uso del gráfico original (en lugar de la transposición) en la segunda profundidad primero buscar y escanear los vértices en orden de tiempos de finalización *crecientes* . Es el profesor ¿correcto?

Ejercicios 22.5-4

Demuestre que para cualquier grafo dirigido G , tenemos $((G^T)^{SCC})^T = G^{SCC}$. Es decir, la transposición del gráfico componente de G^T es el mismo que el gráfico de componente de G .

Ejercicios 22.5-5

Dar un algoritmo de tiempo $O(V + E)$ para calcular el gráfico de componentes de un gráfico dirigido $G = (V, E)$. Asegúrese de que haya como máximo un borde entre dos vértices en el gráfico de componentes su algoritmo produce.

Ejercicios 22.5-6

Dado un gráfico dirigido $G = (V, E)$, explique cómo crear otro gráfico $G' = (V, E')$ tal que (a) G' tiene los mismos componentes fuertemente conectados que G , (b) G' tiene el mismo componente grafica como G , y (c) E' es lo más pequeño posible. Describe un algoritmo rápido para calcular G' .

Ejercicios 22.5-7

Se dice que un grafo dirigido $G = (V, E)$ está **semiconectado** si, para todos los pares de vértices $u, v \in V$, tenemos o . Proporcione un algoritmo eficiente para determinar si G es o no semiconectado. Demuestre que su algoritmo es correcto y analice su tiempo de ejecución.

Problemas 22-1: clasificación de aristas por búsqueda de amplitud primero

Un bosque que prioriza la profundidad clasifica los bordes de un gráfico en árboles, bordes posteriores, posteriores y transversales. UNA
El árbol de amplitud primero también se puede utilizar para clasificar los bordes accesibles desde la fuente de la búsqueda.
en las mismas cuatro categorías.

Página 474

a. Demuestre que en una búsqueda en amplitud de un gráfico no dirigido, las siguientes propiedades
sostener:
1. No hay bordes traseros ni bordes delanteros.
2. Para cada borde de árbol (u , v), tenemos $d [v] = d [u] + 1$.
3. Para cada borde transversal (u , v), tenemos $d [v] = d [u]$ o $d [v] = d [u] + 1$.

si. Demuestre que en una búsqueda en amplitud de un gráfico dirigido, se cumplen las siguientes propiedades:
1. No hay bordes delanteros.
2. Para cada borde de árbol (u , v), tenemos $d [v] = d [u] + 1$.
3. Para cada borde transversal (u , v), tenemos $d [v] \le d [u] + 1$.
4. Para cada borde posterior (u , v), tenemos $0 \le d [v] \le d [u]$.

Problemas 22-2: puntos de articulación, puentes y componentes biconectados

Sea $G = (V , E)$ un gráfico no dirigido y conectado. Un **punto de articulación** de G es un vértice cuyo
eliminación desconecta G . Un **puente** de G es un borde cuya eliminación desconecta G . UNA
El componente biconectado de G es un conjunto máximo de aristas de modo que dos aristas cualesquiera del conjunto se encuentran
en un ciclo simple común. La figura 22.10 ilustra estas definiciones. Podemos determinar
puntos de articulación, puentes y componentes biconectados mediante búsqueda en profundidad. Sea $G_\pi =$
(V , E_π) sea un árbol de G *en* profundidad primero .

Figura 22.10: Los puntos de articulación, puentes y componentes biconectados de un
gráfica no dirigida para usar en el problema 22-2. Los puntos de articulación están muy sombreados.
vértices, los puentes son los bordes fuertemente sombreados, y los componentes biconectados son los
bordes en las regiones sombreadas, con una numeración *CCO* mostrada.

a. Demuestre que la raíz de G_π es un punto de articulación de G si y solo si tiene al menos dos
niños en G_π .

si. Sea v un vértice no raíz de G_π . Demuestre que v es un punto de articulación de G si y solo si
v tiene un hijo s tal que no hay borde posterior de s o cualquier descendiente de s a un
antepasado del v .

C. Dejar

Muestre cómo calcular *bajo* [v] para todos los vértices v V en el tiempo $O (E)$.

re. Muestre cómo calcular todos los puntos de articulación en tiempo $O (E)$.

mi. Demuestre que una arista de G es un puente si y solo si no se encuentra en un ciclo simple de
G .

F. Muestre cómo calcular todos los puentes de G en tiempo $O (E)$.

gramo. Demostrar que el vértice de corte de G partición de los bordes nonbridge de G .

Página 475

h. Dar un algoritmo de tiempo $O (E)$ para etiquetar cada borde e de G con un entero positivo *bcc* [e]

tal que *bcc* [*e*] = *bcc* [*e* ′] si y solo si *e* y *e* ′ están en el mismo biconectado componente.

Problemas 22-3: gira de Euler

Un recorrido de Euler de un gráfico dirigido y conectado $G = (V, E)$ es un ciclo que atraviesa cada borde de G exactamente una vez, aunque puede visitar un vértice más de una vez.

a. Demuestre que G tiene un recorrido de Euler si y solo si grado de entrada (v) = grado de salida (v) para cada vértice $v \in V$.
si. Describe un algoritmo de tiempo $O(E)$ para encontrar un recorrido de Euler de G, si existe. (*Pista:* Fusionar ciclos de borde-disjuntos.)

Problemas 22-4: Accesibilidad

Sea $G = (V, E)$ una gráfica dirigida en la que cada vértice $u \in V$ está etiquetado con un número entero único $L(u)$ del conjunto $\{1, 2, ..., |V|\}$. Para cada vértice $u \in V$, sea ser el conjunto de vértices que son accesibles desde u . Defina min (u) como el vértice en $R(u)$ cuya etiqueta es mínima, es decir, min (u) es el vértice v tal que $L(v) = \min \{L(w) : w \in R(u)\}$. Dar un tiempo $O(V + E)$ algoritmo que calcula min (u) para todos los vértices $u \in V$.

Notas del capítulo

Incluso [87] y Tarjan [292] son excelentes referencias para algoritmos de gráficos.

La búsqueda en amplitud primero fue descubierta por Moore [226] en el contexto de encontrar caminos a través de laberintos. Lee [198] descubrió de forma independiente el mismo algoritmo en el contexto del enrutamiento de cables en placas de circuito.

Hopcroft y Tarjan [154] abogaron por el uso de la representación de lista de adyacencia sobre la representación de matriz de adyacencia para gráficos dispersos y fueron los primeros en reconocer la importancia algorítmica de la búsqueda en profundidad. La búsqueda en profundidad se ha utilizado ampliamente desde finales de la década de 1950, especialmente en programas de inteligencia artificial.

Tarjan [289] proporcionó un algoritmo de tiempo lineal para encontrar componentes fuertemente conectados. los El algoritmo para componentes fuertemente conectados en la Sección 22.5 está adaptado de Aho, Hopcroft, y Ullman [6] , que se lo atribuyen a SR Kosaraju (inédito) y M. Sharir [276] . Gabow [101] también desarrolló un algoritmo para componentes fuertemente conectados que se basa en ciclos de contratación y utiliza dos pilas para que se ejecute en tiempo lineal. Knuth [182] fue el primero para proporcionar un algoritmo de tiempo lineal para la clasificación topológica.

Capítulo 23: Árboles de expansión mínimos

Página 476

Visión general

En el diseño de circuitos electrónicos, a menudo es necesario hacer los pines de varios componentes eléctricamente equivalentes conectándolos juntos. Para interconectar un conjunto de n pines, podemos usar una disposición de n - 1 cables, cada uno conectando dos pines. De todos esos arreglos, el que utiliza la menor cantidad de cable suele ser el más deseable.

Podemos modelar este problema de cableado con un gráfico conectado y no dirigido $G = (V, E)$, donde V es el conjunto de pines, E es el conjunto de posibles interconexiones entre pares de pines, y para cada edge (u, v) $\in E$, tenemos un peso $w(u, v)$ que especifica el costo (cantidad de cable necesario) para conecta u y v . Luego deseamos encontrar un subconjunto acíclico TE que conecte todos los vértices y cuyo peso total

se minimiza. Como T es acíclico y conecta todos los vértices, debe formar un árbol, que llamar a un **árbol de expansión** , ya que "vanos" el grafo G . Llamamos al problema de determinar el árbol. T el **problema del árbol de expansión mínimo** . [1] La figura 23.1 muestra un ejemplo de un gráfico y su árbol de expansión mínimo.

Figura 23.1: Un árbol de expansión mínimo para un gráfico conectado. Los pesos en los bordes son se muestra, y los bordes en un árbol de expansión mínima están sombreados. El peso total del árbol que se muestra es 37. Este árbol de expansión mínimo no es único: eliminar el borde (b, c) y reemplazarlo con el borde (a, h) produce otro árbol de expansión con peso 37.

En este capítulo, examinaremos dos algoritmos para resolver el árbol de expansión mínimo Problema: algoritmo de Kruskal y algoritmo de Prim. Cada uno puede ejecutarse fácilmente a tiempo $O (E \lg V)$ usando montones binarios ordinarios. Al usar montones de Fibonacci, el algoritmo de Prim se puede aceleró para correr en el tiempo $O (E + V \lg V)$, lo cual es una mejora si $| V |$ es mucho más pequeño que $| E |$.

Los dos algoritmos son algoritmos codiciosos, como se describe en el Capítulo 16 . En cada paso de un algoritmo, se debe hacer una de varias opciones posibles. Los defensores de la estrategia codiciosa tomando la mejor decisión en este momento. Esta estrategia generalmente no está garantizada. para encontrar soluciones óptimas a los problemas a nivel mundial. Para el problema del árbol de expansión mínimo, Sin embargo, podemos demostrar que ciertas estrategias codiciosas producen un árbol de expansión con un mínimo peso. Aunque el presente capítulo puede leerse independientemente del capítulo 16 , el codicioso Los métodos presentados aquí son una aplicación clásica de las nociones teóricas introducidas allí.

La sección 23.1 presenta un algoritmo "genérico" de árbol de expansión mínimo que aumenta un árbol agregando un borde a la vez. La sección 23.2 da dos formas de implementar el genérico algoritmo. El primer algoritmo, debido a Kruskal, es similar a los componentes conectados algoritmo de la Sección 21.1 . El segundo, debido a Prim, es similar a los caminos más cortos de Dijkstra. algoritmo (Sección 24.3).

Página 477

[1] La frase "árbol de expansión mínimo" es una forma abreviada de la frase "peso mínimo árbol de expansión. "No estamos, por ejemplo, minimizando el número de aristas en T , ya que todos árboles que se extienden tienen exactamente $| V |$ - 1 aristas por el teorema B.2 .

23.1 Cultivo de un árbol de expansión mínimo

Suponga que tenemos una gráfica conectada y no dirigida $G = (V, E)$ con una función de peso $w : E \rightarrow \mathbf{R}$, y deseamos encontrar un árbol de expansión mínimo para el G . Los dos algoritmos que consideramos en Este capítulo utiliza un enfoque codicioso del problema, aunque difieren en cómo aplican este Acercarse.

Esta estrategia codiciosa es capturada por el siguiente algoritmo "genérico", que aumenta la árbol de expansión mínimo un borde a la vez. El algoritmo gestiona un conjunto de aristas A , manteniendo el siguiente ciclo invariante:

- Antes de cada iteración, A es un subconjunto de algún árbol de expansión mínimo.

En cada paso, determinamos una arista (u, v) que se puede agregar a A sin violar esta invariante, en el sentido de que $A \cup \{(u, v)\}$ es también un subconjunto de un árbol de expansión mínimo. Nosotros llamamos tal borde es un borde **seguro** para A , ya que se puede agregar con seguridad a A mientras se mantiene el invariante.

```
GENÉRICO-MST ( G , w )
1 A ← Ø
2 mientras que A no forma un árbol de expansión
3 do encontrar una arista ( u , v ) es seguro para una
4                   A ← A          {( u , v )}
5 devuelve A
```

Usamos el invariante de bucle de la siguiente manera:

- **Inicialización:** Después de la línea 1, el conjunto A satisface trivialmente el invariante del ciclo.
- **Mantenimiento:** El bucle en las líneas 2-4 mantiene el invariante agregando solo bordes seguros.
- **Terminación:** Todos los bordes agregados a A están en un árbol de expansión mínimo, por lo que el conjunto A se devuelve en la línea 5 debe ser un árbol de expansión mínimo.

La parte complicada es, por supuesto, encontrar una ventaja segura en la línea 3. Hay que existir, ya que cuando la línea 3 es ejecutado, el invariante dicta que hay un árbol de expansión T tal que AT . Dentro de **mientras** cuerpo del bucle, A debe ser un subconjunto propio de T , y por lo tanto no debe ser un borde (u, v) T tal que (u, v) $\notin A$ y (u, v) es seguro para A .

En el resto de esta sección, proporcionamos una regla (Teorema 23.1) para reconocer los bordes seguros. La siguiente sección describe dos algoritmos que usan esta regla para encontrar bordes seguros de manera eficiente.

Primero necesitamos algunas definiciones. Un **corte** ($S, V - S$) de un gráfico no dirigido $G = (V , E)$ es un partición de V . La figura 23.2 ilustra esta noción. Decimos que un borde (u , v) E **cruza** el corte ($S , V - S$) si uno de sus extremos está en S y el otro es en $V - S$. Decimos que un corte **respeta** un conjunto A de aristas si ninguna arista en A cruza el corte. Un borde es un **borde ligero que** cruza un corte si su el peso es el mínimo de cualquier borde que cruce el corte. Tenga en cuenta que puede haber más de una

Página 478

borde ligero que cruza un corte en el caso de corbatas. De manera más general, decimos que un borde es un **borde ligero** satisfaciendo una propiedad dada si su peso es el mínimo de cualquier borde que satisfaga la propiedad.

Figura 23.2: Dos formas de ver un corte ($S, V - S$) del gráfico de la Figura 23.1. *(a)* El los vértices del conjunto S se muestran en negro y los de $V - S$ se muestran en blanco. Los bordes que cruzan el corte están los que conectan vértices blancos con vértices negros. El borde (d, c) es el borde ligero único que cruza el corte. Un subconjunto A de los bordes está sombreado; tenga en cuenta que el corte ($S, V - S$) respeta A , ya que ningún borde de A cruza el corte. *(b)* La misma gráfica con los vértices en el el conjunto S a la izquierda y los vértices del conjunto $V - S$ a la derecha. Un borde cruza el corte si conecta un vértice de la izquierda con un vértice de la derecha.

Nuestra regla para reconocer bordes seguros viene dada por el siguiente teorema.

Teorema 23.1

Sea $G = (V , E)$ un gráfico no dirigido y conectado con una función de peso de valor real w definida en E . Sea A un subconjunto de E que está incluido en algún árbol de expansión mínimo para G , sea ($S , V - S$) sea cualquier corte de G que respete A , y sea (u , v) un cruce de borde ligero ($S , V - S$). Entonces, borde (U , v) es seguro para *una* .

Prueba Sea T un árbol de expansión mínimo que incluye A , y suponga que T no contiene el borde claro (u , v), ya que si lo hace, hemos terminado. Construiremos otro mínimo árbol de expansión T' que incluye $A \cup \{(u , v)\}$ mediante el uso de una técnica de cortar y pegar, por lo tanto mostrando que (u , v) es un borde seguro para A .

La arista (u , v) forma un ciclo con las aristas en la trayectoria p desde u hasta v en T , como se ilustra en Figura 23.3 . Dado que u y v están en lados opuestos del corte ($S , V - S$), hay al menos un borde en T en el camino p que también cruza el corte. Sea (x , y) cualquier borde. El borde (x , y) no es en A , porque el corte respeta A . Dado que (x , y) está en la ruta única de u a v en T , eliminando (x , y) divide T en dos componentes. Agregar (u , v) los vuelve a conectar para formar una nueva expansión árbol $T' = T - \{(x , y)\} \cup \{(u , v)\}$.

Página 479

Figura 23.3: Demostración del teorema 23.1. Los vértices en S son negros y los vértices en $V - S$ son blancas. Se muestran las aristas del árbol de expansión mínimo T, pero las aristas del gráfico G no son. Los bordes en A están sombreados y (u, v) es un borde claro que cruza el corte $(S, V - S)$. los borde (x, y) es una ventaja en el camino único p de u a v en T. Un árbol de expansión mínimo T' que contiene (u, v) se forma quitando el borde (x, y) de T y agregando el borde (u, v).

A continuación, mostramos que T' es un árbol de expansión mínimo. Dado que (u, v) es un cruce de bordes ligero $(S, V - S)$ y (x, y) también cruzan este corte, $w(u, v) \le w(x, y)$. Por lo tanto,

$$\begin{aligned} w(T') &= w(T - w(x, y) + w(u, v) \\ &\le w(T). \end{aligned}$$

Pero T es un árbol de expansión mínimo, de modo que $w(T) \le w(T')$; por tanto, T' debe ser una extensión mínima árbol también.

Queda por demostrar que (u, v) es en realidad una ventaja segura para A. Tenemos AT', ya que AT y $(x, y) \notin A$; por tanto, $A \{(u, v)\} T'$. En consecuencia, dado que T' es un árbol de expansión mínimo, (u, v) es seguro para *una*.

El teorema 23.1 nos da una mejor comprensión del funcionamiento del GENERIC-MST algoritmo en un gráfico conectado $G = (V, E)$. A medida que avanza el algoritmo, el conjunto A es siempre acíclico; de lo contrario, un árbol de expansión mínimo que incluya A contendría un ciclo, que es un contradicción. En cualquier punto de la ejecución del algoritmo, el gráfico $G_A = (V, A)$ es un bosque, y cada uno de los componentes conectados de G_A es un árbol. (Algunos de los árboles pueden contener solo uno vértice, como es el caso, por ejemplo, cuando comienza el algoritmo: A está vacío y el bosque contiene $|V|$ árboles, uno para cada vértice.) Además, cualquier borde seguro (u, v) para A conecta distintas componentes de G_A, ya que $A \{(u, v)\}$ debe ser acíclico.

Se ejecuta el bucle en las líneas 2-4 de GENERIC-MST $|V| - 1$ vez cada uno de los $|V| - 1$ aristas de un árbol de expansión mínimo se determina sucesivamente. Inicialmente, cuando $A = \emptyset$, hay $|V|$

Página 480

árboles en G_A , y cada iteración reduce ese número en 1. Cuando el bosque contiene solo un árbol único, el algoritmo termina.

Los dos algoritmos de la sección 23.2 utilizan el siguiente corolario del teorema 23.1 .

Corolario 23.2

Sea $G = (V , E)$ un gráfico no dirigido y conectado con una función de peso de valor real w definida en E . Sea A un subconjunto de E que está incluido en algún árbol de expansión mínimo para G , y sea $C = (V_C , E_C)$ ser un componente conectado (árbol) en el bosque $G_A = (V , A)$. Si (u , v) es un borde ligero conexión C a algún otro componente en G_A , entonces (u , v) es seguro para A .

Prueba El corte $(V_C , V - V_C)$ respeta A , y (u , v) es un borde ligero para este corte. Por tanto, (u , v) es seguro para *una* .

Ejercicios 23.1-1

Vamos a (u , v) un borde mínimo peso en un gráfico de G . Demuestre que (u , v) pertenece a algunos árbol de expansión mínimo de G .

Ejercicios 23.1-2

El profesor Sabatier conjetura lo contrario del teorema 23.1 . Sea $G = (V , E)$ un conectado, grafo no dirigido con una función de ponderación de valor real- w define en E . Sea A un subconjunto de E que se incluye en algún árbol de expansión mínimo para G , sea $(S , V - S)$ cualquier corte de G que respeta A , y sea (u , v) un borde seguro para el cruce A $(S , V - S)$. Entonces, (u , v) es una luz borde para el corte. Demuestre que la conjetura del profesor es incorrecta dando un contraejemplo.

Ejercicios 23.1-3

Demuestre que si un borde (u , v) está contenido en algún árbol de expansión mínimo, entonces es un borde claro cruzando algún corte del gráfico.

Ejercicios 23.1-4

Dé un ejemplo simple de una gráfica tal que el conjunto de aristas $\{(u , v)$: existe un corte $(S , V - S)$ tal que (u , v) es un cruce de borde ligero $(S , V - S)\}$ no forma un árbol de expansión mínimo.

Página 481

Ejercicios 23.1-5

Sea e una arista de peso máximo en algún ciclo de $G = (V , E)$. Demuestra que hay un mínimo árbol de expansión de $G' = (V , E - \{ e \})$ que es también un árbol de expansión mínima de G . Es decir, hay un árbol de expansión mínimo de G que no incluye e .

Ejercicios 23.1-6

Demuestre que un gráfico tiene un árbol de expansión mínimo único si, para cada corte del gráfico, hay un borde de luz único que cruza el corte. Demuestre que lo contrario no es cierto dando un contraejemplo.

Ejercicios 23.1-7

Argumente que si todos los pesos de los bordes de un gráfico son positivos, entonces cualquier subconjunto de bordes que conecte todos los vértices y tiene un peso total mínimo debe ser un árbol. Dé un ejemplo para demostrar que No se sigue la misma conclusión si permitimos que algunas ponderaciones no sean positivas.

Ejercicios 23.1-8

Sea T un árbol de expansión mínimo de un gráfico G , y sea L la lista ordenada del borde pesos de T . Demuestre que para cualquier otro árbol de expansión mínimo T' de G , la lista L es también la lista ordenada de los pesos de los bordes de T'.

Ejercicios 23.1-9

Deje que T sea un árbol de expansión mínimo de un grafo $G = (V , E)$, y dejar que V'sea un subconjunto de V . Sea T' el subgrafo de T inducido por V', y sea G' el subgrafo de G inducido por V'. Demuestra que si T' está conectado, entonces T' es un árbol de expansión mínimo de G'.

Ejercicios 23.1-10

Página 482

Dado un gráfico G y un árbol de expansión mínimo T , suponga que disminuimos el peso de uno de los bordes en T . Demostrar que T es todavía un árbol de expansión mínima de G . Más formalmente, deja T sea un árbol de expansión mínimo para G con pesos de los bordes dados por la función de peso w . Elige uno edge $(x , y) T$ y un número positivo k , y defina la función de peso w' por

Demuestre que T es un árbol de expansión mínimo para G con pesos de borde dados por w'.

Ejercicios 23.1-11:

Dado un gráfico G y un árbol de expansión mínimo T , suponga que disminuimos el peso de uno de los bordes no en T . Dar un algoritmo para encontrar el árbol de expansión mínimo en el gráfico modificado.

23.2 Los algoritmos de Kruskal y Prim

Los dos algoritmos de árbol de expansión mínimo descritos en esta sección son elaboraciones del algoritmo genérico. Cada uno usa una regla específica para determinar una ventaja segura en la línea 3 de GENÉRICO-MST. En el algoritmo de Kruskal, el conjunto A es un bosque. La ventaja segura agregada a A es siempre un borde de menor peso en el gráfico que conecta dos componentes distintos. En Prim's algoritmo, el conjunto A forma un solo árbol. La ventaja segura agregada a A es siempre la de menor peso borde que conecta el árbol a un vértice que no está en el árbol.

Algoritmo de Kruskal

El algoritmo de Kruskal se basa directamente en el algoritmo genérico de árbol de expansión mínimo dado en la Sección 23.1 . Encuentra un borde seguro para agregar al bosque en crecimiento al encontrar, de todos los bordes que conectan dos árboles cualesquiera en el bosque, un borde (u , v) de menor peso. Deje C_1 y C_2 denotan los dos árboles que están conectados por (u , v). Dado que (u , v) debe ser un borde ligero que conecta C_1 con algún otro árbol, el Corolario 23.2 implica que (u , v) es una ventaja segura para C_1 . El algoritmo de Kruskal es un algoritmo codicioso, porque en cada paso agrega al bosque un borde del menor peso posible.

Nuestra implementación del algoritmo de Kruskal es como el algoritmo para calcular conexiones componentes de la Sección 21.1 . Utiliza una estructura de datos de conjuntos disjuntos para mantener varios conjuntos de elementos. Cada conjunto contiene los vértices de un árbol del bosque actual. La operacion FIND-SET (u) devuelve un elemento representativo del conjunto que contiene u . Por lo tanto, podemos determinar si dos vértices u y v pertenecen al mismo árbol probando si encontrar-SET (u) es igual a FIND-SET (v). La combinación de árboles es realizada por la UNION procedimiento.

```
MST-KRUSKAL ( G , w )
1 A ← Ø
2 para cada vértice v                    V [ G ]
3 hacer MAKE-SET ( v )
4 ordena los bordes de E en orden no decreciente por peso w
5 para cada borde ( u , v ) E , tomado en orden no decreciente por peso
```

Página 483

```
6 hacer si FIND-SET ( u ) ≠ FIND-SET ( v )
7                    luego A ← A          {( u , v )}
8                          UNIÓN ( u , v )
9 devuelve A
```

El algoritmo de Kruskal funciona como se muestra en la Figura 23.4 . Las líneas 1-3 inicializan el conjunto A al vacío configurar y crear $| V |$ árboles, uno que contiene cada vértice. Los bordes en E se ordenan en orden no decreciente por peso en la línea 4. El bucle **for** en las líneas 5-8 verifica, para cada borde (u , v), si los puntos finales u y v pertenecen al mismo árbol. Si lo hacen, entonces el borde (u , v) no puede añadirse al bosque sin crear un ciclo, y el borde se descarta. De lo contrario, los dos los vértices pertenecen a diferentes árboles. En este caso, la arista (u , v) se suma a A en la línea 7, y la los vértices de los dos árboles se fusionan en la línea 8.

Figura 23.4: La ejecución del algoritmo de Kruskal en el gráfico de la Figura 23.1. Sombreado los bordes pertenecen al bosque A *que se* está cultivando. Los bordes son considerados por el algoritmo en ordenados ordenar por peso. Una flecha apunta al borde que se está considerando en cada paso del

algoritmo. Si el borde une dos árboles distintos en el bosque, se agrega al bosque, por lo tanto fusionando los dos árboles.

El tiempo de ejecución del algoritmo de Kruskal para un gráfico $G = (V , E)$ depende de la implementación de la estructura de datos de conjuntos disjuntos. Asumiremos el bosque disjoint-set-forest implementación de la Sección 21.3 con las heurísticas de unión por rango y de compresión de ruta, ya que es la implementación asintóticamente más rápida conocida. La inicialización del conjunto A en la línea 1 requiere

Página 484

O (1) tiempo, y el tiempo para ordenar los bordes en la línea 4 es $O (E \lg E)$. (Contabilizaremos el costo de la $| V |$ Operaciones MAKE-SET en el **de** bucle de las líneas 2-3 en un momento.) La **para** bucle de las líneas 5-8 realizan operaciones $O (E)$ FIND-SET y UNION en el bosque de conjuntos disjuntos. A lo largo con el $| V |$ Operaciones MAKE-SET, toman un total de $O ((V + E) \alpha (V))$ tiempo, donde α es el función de crecimiento muy lento definida en la Sección 21.4 . Como se supone que G está conectado, tenemos $| E | \geq | V | - 1$, por lo que las operaciones de conjuntos disjuntos toman $O (E \alpha (V))$ tiempo. Además, dado que $\alpha (| V |) = O (\lg V) = O (\lg E)$, el tiempo de ejecución total del algoritmo de Kruskal es $O (E \lg E)$. Observando que $| E | < | V |^2$, tenemos $\lg | E | = O (\lg V)$, por lo que podemos reformular el tiempo de ejecución de Algoritmo de Kruskal como $O (E \lg V)$.

El algoritmo de Prim

Como el algoritmo de Kruskal, el algoritmo de Prim es un caso especial del mínimo genérico: algoritmo de árbol de expansión de la Sección 23.1 . El algoritmo de Prim funciona de manera muy similar al de Dijkstra. algoritmo para encontrar las rutas más cortas en un gráfico, que veremos en la Sección 24.3 . Prim's El algoritmo tiene la propiedad de que las aristas del conjunto A siempre forman un solo árbol. Como es ilustrado en la figura 23.5 , el árbol comienza desde un vértice raíz arbitrario r y crece hasta que el árbol abarca todos los vértices en V . En cada paso, se agrega un borde claro al árbol A que conecta A con un vértice aislado de $G_A = (V , A)$. Según el corolario 23.2 , esta regla agrega solo los bordes que son seguros para A ; por lo tanto, cuando el algoritmo termina, los bordes de A forman un árbol de expansión mínimo. Esta estrategia es codiciosa ya que el árbol se aumenta en cada paso con una ventaja que contribuye la mínima cantidad posible al peso del árbol.

Figura 23.5: La ejecución del algoritmo de Prim en el gráfico de la Figura 23.1. El vértice de la raíz es *un* . Los bordes sombreados están en el árbol que se está cultivando y los vértices del árbol se muestran en negro. En cada paso del algoritmo, los vértices del árbol determinan un corte del gráfico y una luz el borde que cruza el corte se agrega al árbol. En el segundo paso, por ejemplo, el algoritmo tiene un opción de agregar borde (*b, c*) o borde (*a, h*) al árbol ya que ambos son bordes claros que se cruzan el corte.

Página 485

La clave para implementar el algoritmo de Prim de manera eficiente es facilitar la selección de una nueva ventaja.
que se añaden al árbol formado por los bordes en A . En el pseudocódigo siguiente, el
El gráfico G y la raíz r del árbol de expansión mínimo que se va a cultivar son entradas del algoritmo.
Durante la ejecución del algoritmo, todos los vértices que *no* están en el árbol residen en una prioridad mínima.
cola Q basada en un campo *clave* . Para cada vértice v , la *clave* [v] es el peso mínimo de cualquier borde
conectando v a un vértice en el árbol; por convención, *clave* [v] = ∞ si no existe tal borde. los
el campo π [v] nombra al padre de v en el árbol. Durante el algoritmo, el conjunto A de GENERIC-
MST se mantiene implícitamente como

$A = \{(v , \pi [v]): v \ V - \{ r \} - Q \}$.

Cuando el algoritmo termina, la cola de prioridad mínima Q está vacía; la extensión mínima
árbol A para G es así

$A = \{(v , \pi [v]): v \ V - \{ r \}\}$.

```
MST-PRIM ( G , w , r )
 1 para cada u                V [ G ]
 2 hacer tecla [ u ] ← ∞
 3                 π [ u ] ← NULO
 4 tecla [ r ] ← 0
 5 Q ← V [ G ]
 6 mientras Q ≠ Ø
 7 ¿ u ← EXTRACTO-MIN ( Q )
 8                 para cada v            Adj [ u ]
 9                     hacer si v         Q y w ( u , v ) < tecla [ v ]
10                                entonces π [ v ] ← u
11                                     tecla [ v ] ← w ( u , v )
```

El algoritmo de Prim funciona como se muestra en la Figura 23.5 . Las líneas 1-5 establecen la clave de cada vértice en ∞
(a excepción de la raíz r , cuya clave se establece en 0 para que sea el primer vértice procesado), establezca el
padre de cada vértice a NIL, e inicializar la cola de prioridad mínima Q para contener todos los
vértices. El algoritmo mantiene el siguiente ciclo invariante de tres partes:

Antes de cada iteración del **tiempo** de bucle de líneas 6-11,

1. $A = \{(v , \pi [v]): v \ V - \{ r \} - Q \}$.
2. Los vértices ya colocadas en el árbol de expansión mínima son aquellos en los $V - Q$.
3. Para todos los vértices $v \ Q$, si π [v] ≠ NIL, entonces la *tecla* [v] <∞ y la *tecla* [v] es el peso de una luz
   edge (v , π [v]) conectando v a algún vértice ya colocado en la extensión mínima
   árbol.

La línea 7 identifica un vértice $u \ Q$ incidente en un borde claro que cruza el corte ($V - Q$, Q) (con el
excepción de la primera iteración, en la que $u = r$ debido a la línea 4). Quitar u del conjunto Q agrega
al conjunto $V - Q$ de vértices en el árbol, lo que añade (u , π [u]) para A . El bucle **for** de las líneas 8-11
actualice los campos *key* y π de cada vértice v adyacente a u pero no en el árbol. La actualización
mantiene invariante la tercera parte del ciclo.

El rendimiento del algoritmo de Prim depende de cómo implementemos la cola de prioridad mínima
Q . Si Q se implementa como un min-heap binario (consulte el Capítulo 6), podemos usar BUILD-MIN-
Procedimiento HEAP para realizar la inicialización en las líneas 1-5 en tiempo $O (V)$. El cuerpo del **tiempo**
se ejecuta el bucle | V | veces, y dado que cada operación EXTRACT-MIN toma O (lg V) tiempo, la

Página 486

el tiempo total para todas las llamadas a EXTRACT-MIN es $O (V \lg V)$. Se ejecuta el ciclo **for** en las líneas 8-11
$O (E)$ veces en total, ya que la suma de las longitudes de todas las listas de adyacencia es 2 | E |. Dentro de
for loop, la prueba de pertenencia a Q en la línea 9 se puede implementar en tiempo constante por
manteniendo un bit para cada vértice que dice si está o no en Q , y actualizando el bit cuando el
vértice se elimina de Q . La asignación en la línea 11 implica una TECLA DE DISMINUCIÓN implícita
operación en el min-heap, que se puede implementar en un min-heap binario en tiempo O (lg V).
Por lo tanto, el tiempo total para el algoritmo de Prim es $O (V \lg V + E \lg V) = O (E \lg V)$, que es
asintóticamente lo mismo que para nuestra implementación del algoritmo de Kruskal.

El tiempo de ejecución asintótico del algoritmo de Prim se puede mejorar, sin embargo, usando
Montones de Fibonacci. El capítulo 20 muestra que si | V | los elementos están organizados en un montón de Fibonacci,
podemos realizar una operación EXTRACTO-MIN en O (lg V) tiempo amortizado y una DISMINUCIÓN-

Operación LLAVE (para implementar la línea 11) en $O(1)$ tiempo amortizado. Por lo tanto, si usamos un Montón de Fibonacci para implementar la cola de prioridad mínima Q, el tiempo de ejecución del algoritmo de Prim mejora a $O(E + V \lg V)$.

Ejercicios 23.2-1

El algoritmo de Kruskal puede devolver diferentes árboles de expansión para el mismo gráfico de entrada G, dependiendo sobre cómo se rompen los lazos cuando los bordes se clasifican en orden. Demuestre eso para cada mínimo árbol de expansión T de G, hay una manera de ordenar los bordes de G en el algoritmo de Kruskal para que el algoritmo regresa T.

Ejercicios 23.2-2

Suponga que la gráfica $G = (V, E)$ se representa como una matriz de adyacencia. Dar un simple implementación del algoritmo de Prim para este caso que se ejecuta en tiempo $O(V^2)$.

Ejercicios 23.2-3

¿Es la implementación del montón de Fibonacci del algoritmo de Prim asintóticamente más rápida que la Implementación del montón binario para un gráfico disperso $G = (V, E)$, donde $|E| = \Theta(V)$? Que tal para un gráfico denso, donde $|E| = \Theta(V^2)$? ¿Cómo debe $|E|$ y $|V|$ estar relacionado para el montón de Fibonacci la implementación sea asintóticamente más rápida que la implementación del montón binario?

Ejercicios 23.2-4

Suponga que todos los pesos de los bordes en un gráfico son números enteros en el rango de 1 a $|V|$. ¿Qué tan rápido puede ¿Haces que funcione el algoritmo de Kruskal? ¿Qué pasa si los pesos de los bordes son números enteros en el rango de 1 a W para alguna W constante ?

Página 487

Ejercicios 23.2-5

Suponga que todos los pesos de los bordes en un gráfico son números enteros en el rango de 1 a $|V|$. ¿Qué tan rápido puede haces que el algoritmo de Prim funcione? ¿Qué pasa si los pesos de los bordes son números enteros en el rango de 1 a ¿W para alguna W constante ?

Ejercicios 23.2-6: ⋆

Suponga que los pesos de los bordes en un gráfico se distribuyen uniformemente en el intervalo semiabierto [0, 1). ¿Qué algoritmo, Kruskal o Prim, puedes hacer que funcione más rápido?

Ejercicios 23.2-7: ⋆

Suponga que una gráfica G tiene un árbol de expansión mínimo ya calculado. ¿Qué tan rápido puede el ¿Se actualizará el árbol de expansión mínimo si se agregan un nuevo vértice y bordes incidentes a G ?

Ejercicios 23.2-8

El profesor Toole propone un nuevo algoritmo de divide y vencerás para calcular el mínimo abarcando árboles, que es el siguiente. Dado un gráfico $G = (V , E)$, particione el conjunto V de vértices en dos conjuntos V_1 y V_2 tales que $| V_1 |$ y $| V_2 |$ difieren como máximo en 1. Sea E_1 el conjunto de aristas que inciden solo en los vértices en V_1, y sea E_2 el conjunto de aristas que inciden solo en vértices en V_2. Resuelva de forma recursiva un problema de árbol de expansión mínimo en cada uno de los dos subgrafos $G_1 = (V_1 , E_1)$ y $G_2 = (V_2 , E_2)$. Finalmente, seleccione la arista de peso mínimo en E que cruza el corte (V_1 , V_2), y use este borde para unir los dos árboles de expansión mínimos resultantes en un solo árbol de expansión.

O argumente que el algoritmo calcula correctamente un árbol de expansión mínimo de G, o proporcione un ejemplo en el que falla el algoritmo.

Problemas 23-1: segundo mejor árbol de expansión mínimo

Sea $G = (V , E)$ una gráfica conectada no dirigida con función de peso $w : E \rightarrow \mathbf{R}$, y supongamos que $| E | \geq | V |$ y todos los pesos de los bordes son distintos.

Página 488

Un segundo mejor árbol de expansión mínimo se define de la siguiente manera. Sea el conjunto de todos los que abarcan árboles de G, y dejar que T 'ser un árbol de expansión mínimo de G. Entonces un segundo mejor mínimo árbol de expansión es un árbol de expansión T tal que .

a. Demuestre que el árbol de expansión mínimo es único, pero que el segundo mejor mínimo el árbol de expansión no necesita ser único.
si. Deje que T sea un árbol de expansión mínimo de G. Demuestre que existen aristas $(u , v) T$ y $(x , y) \notin T$ tal que $T - \{(u , v)\} \{(x , y)\}$ es una segunda mejor árbol de expansión mínima de G.
C. Sea T un árbol de expansión de G y, para dos vértices cualesquiera $u , v V$, sea $max [u , v]$ un borde de peso máximo en la trayectoria única entre u y v en T. Describe una $O (V_2)$ - algoritmo de tiempo que, dada T, calcula $max [u , v]$ para todos $u , v V$.
re. Dar un algoritmo eficiente para calcular la segunda mejor árbol de expansión mínimo de G.

Problemas 23-2: árbol de expansión mínimo en gráficos dispersos

Para un gráfico conectado muy escaso $G = (V , E)$, podemos mejorar aún más el $O (E + V \lg V)$ tiempo de ejecución del algoritmo de Prim con montones de Fibonacci "preprocesando" G para disminuir el número de vértices antes de ejecutar el algoritmo de Prim. En particular, elegimos, para cada vértice u, la arista de peso mínimo (u , v) incidente en u, y ponemos (u , v) en el mínimo árbol de expansión en construcción. Luego contraemos todos los bordes elegidos (ver Sección B.4). Más bien que contraer estos bordes uno a la vez, primero identificamos conjuntos de vértices que se unen en el mismo vértice nuevo. Luego, creamos el gráfico que habría resultado de contraer estos bordes uno a la vez, pero lo hacemos "cambiando el nombre" de los bordes de acuerdo con los conjuntos en los que se colocaron sus puntos finales. Varios bordes del gráfico original se pueden renombrar del mismo modo. como el uno al otro. En tal caso, solo resulta un borde, y su peso es el mínimo de pesos de los bordes originales correspondientes.

Inicialmente, establecemos que el árbol de expansión T mínimo que se está construyendo esté vacío, y para cada edge $(u , v) E$, establecemos $orig [u , v] = (u , v)$ y $c [u , v] = w (u , v)$. Usamos el atributo $orig$ para referencia el borde del gráfico inicial que está asociado con un borde en el contraído grafico. El atributo c tiene el peso de un borde y, a medida que los bordes se contraen, se actualiza de acuerdo con el esquema anterior para elegir los pesos de los bordes. El procedimiento MST-REDUCE toma las entradas G, $orig$, c y T, y devuelve un gráfico contraído G' y los atributos actualizados $orig'$ y c' para el gráfico G'. El procedimiento también acumula los bordes de G en la extensión mínima árbol T.

```
MST-REDUCIR ( G , orig , c , T )
  1 por cada v                V [ G ]
```

```
3 hacer marca [ v ] ← FALSO
4 para cada u                V [ G ]
5 hacer si marca [ u ] = FALSE
6                    luego elige v              Ajusta [ u ] de manera que c [ u , v ] se minimice
7                          UNIÓN ( u , v )
8                          T ← T          { orig [ u , v ]}
9                          marca [ u ] ← marca [ v ] ← VERDADERO
10 V [ G ′] ← {FIND-SET ( v ): v                    V [ G ]}
11 E [ G ′] Ø
12 para cada ( x , y ) E [ G ]
```



```
13 ¿ u ← FIND-SET ( x )
14              v ← FIND-SET ( y )
15              si ( u , v ) E [ G ′]
dieciséis                luego E [ G ′] ← E [ G ′] {( u , v )}
17                             orig ′ [ u , v ] ← orig [ x , y ]
18                             c ′ [ u , v ] ← c [ x , y ]
19                       de lo contrario, si c [ x , y ] < c ′ [ u , v ]
20                                luego orig ′ [ u , v ] ← orig [ x , y ]
21                                      c ′ [ u , v ] ← c [ x , y ]
22 listas de adyacencia de constructo Adj para G ′
23 devuelve G ′, orig ′, c ′ y T
```

a. Sea T el conjunto de aristas devueltas por MST-REDUCE, y sea A el mínimo árbol de expansión del gráfico G' formado por la llamada MST-PRIM (G', c', r), donde r es cualquier vértice en $V [G']$. Demuestre que $T \{ orig' [x , y]: (x , y) A \}$ es un árbol de expansión mínimo de G .

si. Argumenta que $| V [G'] | \leq | V | / 2$.

C. Muestre cómo implementar MST-REDUCE para que se ejecute en tiempo $O (E)$. (*Sugerencia:* use estructuras de datos simples.)

re. Supongamos que ejecutamos k fases de MST-REDUCE, utilizando las salidas G', $orig' yc'$ producido por una fase tal como las entradas G , $orig$, y c a la siguiente fase y acumulando bordes de T . Argumenta que el tiempo de ejecución total de las k fases es $O (k E)$.

mi. Suponga que después de ejecutar k fases de MST-REDUCE, como en el inciso d), ejecutamos Prim algoritmo llamando a MST-PRIM (G', c', r), donde G' y c' son devueltos por el último fase yr es cualquier vértice en $V [G']$. Muestre cómo elegir k para que la ejecución general el tiempo es $O (E \lg \lg V)$. Argumenta que tu elección de k minimiza la asintótica general tiempo de ejecución.

F. Para qué valores de $| E |$ (en términos de $| V |$) ¿el algoritmo de Prim con preprocesamiento ¿Batir asintóticamente el algoritmo de Prim sin preprocesamiento?

Problemas 23-3: árbol de expansión de cuellos de botella

Un árbol de expansión de cuello de botella T de un gráfico no dirigido G es un árbol de expansión de G cuyo mayor peso mínimo de borde es de todos los árboles que abarcan de G . Decimos que el valor del cuello de botella árbol de expansión es el peso de la arista máxima de peso en T .

a. Argumente que un árbol de expansión mínimo es un árbol de expansión de cuello de botella.

La parte (a) muestra que encontrar un árbol de expansión de cuello de botella no es más difícil que encontrar un mínimo árbol de expansión. En las partes restantes, mostraremos que se puede encontrar uno en tiempo lineal.

si. Dar un algoritmo de tiempo lineal que, dado un gráfico G y un entero b , determina si el valor del árbol de expansión del cuello de botella es como máximo b .

C. Utilice su algoritmo para el inciso b) como una subrutina en un algoritmo de tiempo lineal para el problema del árbol de expansión del cuello de botella. (*Sugerencia:* es posible que desee utilizar una subrutina que contrae conjuntos de aristas, como en el procedimiento MST-REDUCE descrito en el problema 23-2.)

Página 490

Problemas 23-4: algoritmos alternativos de árbol de expansión mínimo

En este problema, damos pseudocódigo para tres algoritmos diferentes. Cada uno toma un gráfico como de entrada y devuelve un conjunto de bordes T . Para cada algoritmo, debe probar que T es un árbol de expansión mínimo o probar que T no es un árbol de expansión mínimo. También describa el implementación más eficiente de cada algoritmo, ya sea que calcule o no un mínimo árbol de expansión.

a. QUIZÁS-MST-A (G , w)
si. 1 clasifique los bordes en orden no creciente de pesos de borde w
C. 2 $T \leftarrow E$
re. 3 **para** cada borde e , tomado en orden no creciente por peso
mi. 4 **hacer si** $T - \{ e \}$ es un gráfico conectado
F. 5 **luego** $T \leftarrow T - e$
gramo. 6 **retorno** T

h.
yo. QUIZÁS-MST-B (G , w)
j. 1 $T \leftarrow \emptyset$
k. 2 **para** cada borde e , tomado en orden arbitrario
l. 3 **hacer si** T $\{ e \}$ no tiene ciclos
metro. 4 **luego** $T \leftarrow T$ *mi*
norte. 5 **volver** T

o.
pags. QUIZÁS-MST-C (G , w)
q. 1 $T \leftarrow \emptyset$
r. 2 **para** cada borde e , tomado en orden arbitrario
s. 3 **hacer** $T \leftarrow T$ $\{ e \}$
t. 4 **si** T tiene un ciclo c
u. 5 **entonces** sea e' la arista de peso máximo en c
v. 6 $T \leftarrow T - \{ e' \}$
w. 7 **vuelve** T

X.

Notas del capítulo

Tarjan [292] examina el problema del árbol de expansión mínimo y proporciona excelentes material. Una historia del problema del árbol de expansión mínimo ha sido escrita por Graham y Infierno [131] .

Tarjan atribuye el primer algoritmo de árbol de expansión mínimo a un artículo de 1926 de O. Boruvka. El algoritmo de Boruvka consiste en ejecutar $O(\lg V)$ iteraciones del procedimiento MST-REDUCE descrito en el problema 23-2 . El algoritmo de Kruskal fue informado por Kruskal [195] en 1956. El El algoritmo comúnmente conocido como algoritmo de Prim fue inventado por Prim [250] , pero también fue inventado anteriormente por V. Jarník en 1930.

La razón por la que los algoritmos codiciosos son efectivos para encontrar árboles de expansión mínimos es que conjunto de bosques de un gráfico forma una matriz gráfica. (Consulte la Sección 16.4 .)

Cuando $|E| = \Omega(V \lg V)$, el algoritmo de Prim, implementado con montones de Fibonacci se ejecuta en tiempo $O(E)$. Para gráficos más dispersos, utilizando una combinación de las ideas del algoritmo de Prim, Kruskal algoritmo, y el algoritmo de Boruvka, junto con estructuras de datos avanzadas, Fredman y

Página 491

Tarjan [98] proporciona un algoritmo que se ejecuta en tiempo $O(E \lg^* V)$. Gabow, Galil, Spencer y Tarjan [102] mejoró este algoritmo para que se ejecute en tiempo $O(E \lg \lg^* V)$. Chazelle [53] da un algoritmo que corre en tiempo, donde es el inverso funcional de la función de Ackermann. (Véanse las notas del capítulo 21 para una breve discusión de la función de Ackermann y su inversa.) A diferencia de los algoritmos anteriores de árbol de expansión mínimo, el algoritmo de Chazelle no sigue el método codicioso.

Un problema relacionado es la verificación del árbol de expansión , en la que se nos da un gráfico $G = (V, E)$ y un árbol TE , y deseamos determinar si T es un árbol de expansión mínimo de G . Rey [177] proporciona un algoritmo de tiempo lineal para la verificación del árbol de expansión, basado en el trabajo anterior de

Komlós [188] y Dixon, Rauch y Tarjan [77] .

Los algoritmos anteriores son todos deterministas y caen en el modelo basado en la comparación descrito en el Capítulo 8 . Karger, Klein y Tarjan [169] dan un mínimo aleatorio: algoritmo de árbol de expansión que se ejecuta en el tiempo esperado $O(V + E)$. Este algoritmo utiliza la recursividad en de una manera similar al algoritmo de selección de tiempo lineal en la Sección 9.3 : una llamada recursiva en un El problema auxiliar identifica un subconjunto de los bordes E' que no pueden estar en ninguna extensión mínima. árbol. Otra llamada recursiva en $E - E'$ encuentra el árbol de expansión mínimo. El algoritmo también utiliza ideas del algoritmo de Boruvka y el algoritmo de King para la verificación del árbol de expansión.

Fredman y Willard [100] mostraron cómo encontrar un árbol de expansión mínimo en el tiempo $O(V + E)$ utilizando un algoritmo determinista que no se basa en la comparación. Su algoritmo asume que el Los datos son números enteros de b bits y que la memoria de la computadora consta de palabras direccionables de b bits.

Capítulo 24: Rutas más cortas de una sola fuente

Visión general

Un automovilista desea encontrar la ruta más corta posible de Chicago a Boston. Dado un camino mapa de los Estados Unidos en el que la distancia entre cada par de intersecciones adyacentes es marcado, ¿cómo podemos determinar esta ruta más corta?

Una forma posible es enumerar todas las rutas de Chicago a Boston, sumar las distancias en cada ruta y seleccione la más corta. Sin embargo, es fácil ver que incluso si rechazamos rutas que contienen ciclos, hay millones de posibilidades, la mayoría de las cuales simplemente no son vale la pena considerarlo. Por ejemplo, una ruta de Chicago a Houston a Boston es obviamente una mala elección, porque Houston está a mil millas de distancia.

En este capítulo y en el capítulo 25 , mostramos cómo resolver estos problemas de manera eficiente. en un problema de caminos más cortos , se nos da un gráfico dirigido y ponderado $G = (V, E)$, con peso función $w : E \to \mathbf{R}$ mapeo de bordes a pesos de valor real. El peso de la trayectoria $p = v_0, v_1, \ldots, v_k$ es la suma de los pesos de sus aristas constituyentes:

Definimos el peso de la ruta más corta de u a v por

Página 492

Un camino más corto desde el vértice u al vértice v se define entonces como cualquier camino p con peso $w(p) = \delta(u, v)$.

En el ejemplo de Chicago a Boston, podemos modelar la hoja de ruta como un gráfico: los vértices representan intersecciones, los bordes representan los segmentos de la carretera entre las intersecciones y los pesos de los bordes representar distancias de carreteras. Nuestro objetivo es encontrar el camino más corto desde una intersección determinada en Chicago (digamos, Clark St. y Addison Ave.) a una intersección determinada en Boston (digamos, Brookline Ave. y Yawkey Way).

Los pesos de los bordes se pueden interpretar como métricas distintas de las distancias. A menudo están acostumbrados a representan tiempo, costo, multas, pérdidas o cualquier otra cantidad que se acumule linealmente a lo largo de un camino y que se desea minimizar.

El algoritmo de búsqueda primero en amplitud de la Sección 22.2 es un algoritmo de rutas más cortas que funciona en gráficos no ponderados, es decir, gráficos en los que se puede considerar que cada borde tiene una unidad peso. Debido a que muchos de los conceptos de la búsqueda primero en amplitud surgen en el estudio de rutas en gráficos ponderados, se recomienda al lector que revise la Sección 22.2 antes de continuar.

Variantes

En este capítulo, nos centraremos en el problema de los caminos más cortos de una sola fuente : dado un gráfico G

$G = (V, E)$, queremos encontrar un camino más corto desde un vértice fuente dado $s \in V$ a cada vértice $v \in V$. Muchos otros problemas pueden resolverse mediante el algoritmo para el problema de fuente única, incluyendo las siguientes variantes.

- **Problema de rutas más cortas de destino único:** encuentre una ruta más corta a un vértice de destino t de cada vértice v. Al invertir la dirección de cada borde en el gráfico, podemos reducir este problema a un problema de fuente única.
- **Problema de ruta más corta de un solo par:** encuentre una ruta más corta de u a v para vértices u y v. Si resolvemos el problema de fuente única con el vértice de fuente u, resolvemos este problema también. Además, no se conocen algoritmos para este problema que se ejecuten asintóticamente más rápido que los mejores algoritmos de fuente única en el peor de los casos.
- **Problema de rutas más cortas de todos los pares:** encuentre una ruta más corta de u a v para cada par de vértices u y v. Aunque este problema se puede resolver ejecutando una fuente única algoritmo una vez desde cada vértice, por lo general se puede resolver más rápido. Además, su la estructura es de interés por derecho propio. El capítulo 25 aborda el problema de todos los pares en detalle.

Subestructura óptima de un camino más corto

Los algoritmos de rutas más cortas generalmente se basan en la propiedad de que una ruta más corta entre dos vértices contiene otros caminos más cortos dentro de él. (El algoritmo de flujo máximo de Edmonds-Karp en el Capítulo 26 también se basa en esta propiedad.) Esta propiedad de subestructura óptima es un sello de la aplicabilidad tanto de la programación dinámica (Capítulo 15) como del método codicioso (Capítulo 16). El algoritmo de Dijkstra, que veremos en la sección 24.3 , es un algoritmo codicioso, y el Algoritmo de Floyd-Warshall, que encuentra las rutas más cortas entre todos los pares de vértices (ver

Página 493

Capítulo 25), es un algoritmo de programación dinámica. El siguiente lema establece el óptimo-propiedad de la subestructura de los caminos más cortos con mayor precisión.

Lema 24.1: (Los subtrayectos de los caminos más cortos son los caminos más cortos)

Dado un gráfico dirigido y ponderado $G = (V, E)$ con función de peso $w : E \to \mathbf{R}$, sea $p = v_1, v_2, ..., v_k$ ser un camino más corto desde el vértice v_1 al vértice v_k y, para cualquier i y j tal que $1 \le i \le j \le k$, sea $p_{ij} = v_i, v_{i+1}, ..., v_j$ ser el subtrayecto de p desde el vértice v_i al vértice v_j. Entonces, p_{ij} es el más corto camino de v_i a v_j.

Prueba Si descomponemos la ruta p en , entonces tenemos que $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Ahora, suponga que hay un camino de v_i a v_j con peso . Luego, es un camino de v_1 a v_k cuyo peso es menor que $w(p)$, que contradice la suposición de que p es el camino más corto de v_1 a v_k.

Bordes de peso negativo

En algunos casos del problema de las rutas más cortas de una sola fuente, puede haber bordes cuyas los pesos son negativos. Si el gráfico $G = (V, E)$ no contiene ciclos de peso negativo alcanzables desde la fuente s, entonces para todo $v \in V$, el peso de la ruta más corta $\delta(s, v)$ permanece bien definido, incluso si tiene un valor negativo. Sin embargo, si hay un ciclo de peso negativo accesible desde s, Los pesos del camino más corto no están bien definidos. Ningún camino de s a un vértice en el ciclo puede ser un el camino más corto: siempre se puede encontrar un camino de menor peso que siga el "más corto" propuesto ruta y luego atraviesa el ciclo de peso negativo. Si hay un ciclo de peso negativo en algunos trayectoria desde s a v, definimos $\delta(s, v) = -\infty$.

La figura 24.1 ilustra el efecto de los pesos negativos y los ciclos de peso negativo en los pesos de ruta. Debido a que sólo hay un camino de s a *una* (la ruta s, a), *delta* $(s, un) = w(s, un) = 3$. De manera similar, sólo hay una trayectoria desde s a b, y así $\delta(s, b) = w(s, un) + w(un, b) = 3 + (-4) = -1$. hay infinitamente muchos caminos de s a c : $s, c, s, c, d, c, s, c, d, c, d, c$, y así en. Debido a que el ciclo c, d, c tiene un peso $6 + (-3) = 3 > 0$, la ruta más corta desde s a c es s, c, con el peso $\delta(s, c) = 5$. De manera similar, la ruta más corta desde s a d es decir s, c, d, con peso $\delta(s, d) = w(s, c) + w(c, d) = 11$. Análogamente, hay infinitamente muchos caminos de s a e : $s, e, s, e, f, e, s, e, f, e, f, e$, etc. Dado que el ciclo e, f, e tiene un peso $3 + (-6) = -3 < 0$, sin embargo, no hay camino más corto desde s a e. Al atravesar el peso negativo ciclo e, f, e arbitrariamente muchas veces, podemos encontrar caminos de s de *correo* con arbitrariamente grande pesos negativos, por lo que $\delta(s, e) = -\infty$. De manera similar, $\delta(s, f) = -\infty$. Dado que g es accesible desde f, también podemos encontrar caminos con arbitrariamente grandes pesos negativos de s a g, y $\delta(s, g) = -\infty$.

Los vértices h , i y j también forman un ciclo de peso negativo. No se puede acceder a ellos desde s , sin embargo, y entonces $\delta (s , h) = \delta (s , i) = \delta (s , j) = \infty$.

Página 494

Figura 24.1: Pesos de aristas negativos en un gráfico dirigido. Dentro de cada vértice se muestra su peso del camino más corto desde la fuente s . Debido vértices e y f forman un ciclo de peso negativo alcanzables desde s , tienen pesos de ruta más corta de -∞. Debido a que el vértice g es accesible desde un vértice cuyo peso de la ruta más corta es -∞, también tiene un peso de la ruta más corta de -∞. Vértices como h, i y j no son accesibles desde s , por lo que sus pesos de ruta más corta son ∞, incluso aunque se encuentran en un ciclo de peso negativo.

Algunos algoritmos de rutas más cortas, como el algoritmo de Dijkstra, asumen que todos los pesos de los bordes en los gráficos de entrada no son negativos, como en el ejemplo de la hoja de ruta. Otros, como Bellman-Algoritmo de Ford, permite bordes de peso negativo en el gráfico de entrada y produce una respuesta correcta siempre que no se pueda acceder a ciclos de peso negativo desde la fuente. Normalmente, si existe tal un ciclo de peso negativo, el algoritmo puede detectar e informar de su existencia.

Ciclos

¿Puede un camino más corto contener un ciclo? Como acabamos de ver, no puede contener un peso negativo ciclo. Tampoco puede contener un ciclo de ponderación positiva, ya que eliminar el ciclo del camino produce una ruta con los mismos vértices de origen y destino y un peso de ruta menor. Ese es, si $p = v_0 , v_1 , ..., v_k$ es un camino y $c = v_i , v_{i+1} , ..., v_j$ es un ciclo de peso positivo en este ruta (de modo que $v_i = v_j$ y $w (c) > 0$), entonces la ruta $p ' = v_0 , v_1 , ..., v_i , v_{j+1} , v_{j+2} , ... , v_k$ tiene peso $w (p ') = w (p) - w (c) < w (p)$, por lo que p no puede ser el camino más corto de v_0 a v_k .

Eso deja solo ciclos de peso 0. Podemos eliminar un ciclo de peso 0 de cualquier camino para producir otro camino cuyo peso es el mismo. Por lo tanto, si hay un camino más corto desde un vértice fuente s a un vértice de destino v que contiene un ciclo de peso 0, entonces hay otro camino más corto desde s a v sin este ciclo. Siempre que una ruta más corta tenga ciclos de 0 ponderaciones, podemos Eliminamos repetidamente estos ciclos del camino hasta que tengamos un camino más corto que esté libre de ciclos. Por tanto, sin perder la generalidad podemos asumir que cuando estamos encontrando caminos más cortos, no tienen ciclos. Dado que cualquier camino acíclico en un gráfico $G = (V , E)$ contiene como máximo $| V |$ distinto vértices, también contiene como máximo $| V | - 1$ aristas. Por lo tanto, podemos restringir nuestra atención a los caminos de como máximo $| V | - 1$ aristas.

Representando caminos más cortos

A menudo deseamos calcular no solo los pesos de las rutas más cortas, sino también los vértices de las rutas más cortas como bien. La representación que usamos para los caminos más cortos es similar a la que usamos para la amplitud primeros árboles en la Sección 22.2. Dado un gráfico $G = (V , E)$, mantenemos para cada vértice $v \in V$ a predecesor $\pi [v]$ que es otro vértice o NIL. Los algoritmos de rutas más cortas en este capítulo establece los atributos de π modo que la cadena de los predecesores que se origina en un vértice v carreras hacia atrás a lo largo de una ruta más corta desde s a v . Así, dado un vértice v para el cual $\pi [v] \neq$ NIL, el El procedimiento PRINT-PATH (G , s , v) de la Sección 22.2 se puede utilizar para imprimir una ruta más corta desde s a v .

Página 495

Durante la ejecución de un algoritmo de rutas más cortas, sin embargo, los valores de π no necesitan indicar caminos más cortos. Al igual que en la búsqueda en amplitud, nos interesará el subgrafo predecesor $G_\pi = (V_\pi , E_\pi)$ inducido por los valores de π. Aquí de nuevo, definimos el conjunto de vértices V_π como el conjunto de vértices de G con predecesores no NIL, además de la fuente s :

$V_\pi = \{ v \in V : \pi [v] \neq \text{NIL} \} \cup \{ s \}$.

El conjunto de aristas dirigidas E_π es el conjunto de aristas inducidas por los valores de π para los vértices en V_π :

$E_\pi = \{ (\pi [v], v) \in E : v \in V_\pi - \{ s \} \}$.

Demostraremos que los valores de π producidos por los algoritmos de este capítulo tienen la propiedad que en la terminación G_π es un "árbol de caminos más cortos", informalmente, un árbol enraizado que contiene un árbol más corto ruta desde la fuente s a cada vértice accesible desde s . Un árbol de caminos más cortos es como el primer árbol en anchura de la Sección 22.2 , pero contiene las rutas más cortas desde la fuente definida en términos de pesos de aristas en lugar de números de aristas. Para ser precisos, sea $G = (V , E)$ un gráfica dirigida ponderada con función de ponderación $w : E \rightarrow \mathbf{R}$, y suponga que G no contiene ciclos de peso negativo alcanzables desde el vértice de origen $s \in V$, de modo que los caminos más cortos estén bien definido. Un árbol de caminos más cortos con raíces en s es un subgrafo dirigido $G' = (V' , E')$, donde $V' \subseteq V$ y $E' \subseteq E$, tal que

1. V' es el conjunto de vértices alcanzables desde s en G ,
2. G' forma un árbol enraizado con raíz s , y
3. para todos $v \in V'$ la trayectoria simple única de s a v en G' es un camino más corto desde s a v en G .

Los caminos más cortos no son necesariamente únicos, ni tampoco los árboles de los caminos más cortos. Por ejemplo, La figura 24.2 muestra un gráfico dirigido y ponderado y dos árboles de caminos más cortos con la misma raíz.

Figura 24.2: (a) A ponderado, gráfico dirigido con pesos ruta más corta desde la fuente s . (b) El bordes sombreados forman un árbol de más corta caminos arraigada en la fuente s . (c) Otro árbol de caminos más cortos con la misma raíz.

Relajación

Los algoritmos de este capítulo utilizan la técnica de relajación . Para cada vértice $v \in V$, nosotros mantener un atributo $d [v]$, que es un límite superior en el peso de un camino más corto desde fuente s a v . Llamamos $d [v]$ una estimación de la ruta más corta . Inicializamos las estimaciones de ruta más corta y predecesores mediante el siguiente procedimiento de tiempo $\Theta (V)$.

```
INICIALIZAR FUENTE ÚNICA ( G , s )
1 para cada vértice v                    V [ G ]
2 hacer d [ v ] ← ∞
3                  π [ v ] ← NULO
4 d [ s ] 0
```

Después de la inicialización, $\pi [v] = \text{NIL}$ para todo $v \in V$, $d [s] = 0$ y $d [v] = \infty$ para $v \in V - \{ s \}$.

Página 496

El proceso de relajar [1] una ventaja (u , v) consiste en probar si podemos mejorar la ruta más corta a v encontrada hasta ahora pasando por u y, si es así, actualizando $d [v]$ y $\pi [v]$. UNA El paso de relajación puede disminuir el valor de la estimación del camino más corto $d [v]$ y actualizar v 's campo predecesor $\pi [v]$. El siguiente código realiza un paso de relajación en el borde (u , v).

```
RELAJARSE ( u , v , w )
1 si d [ v ]> d [ u ] + w ( u , v )
2 luego d [ v ] ← d [ u ] + w ( u , v )
```

3 $\pi[v] \leftarrow u$

La figura 24.3 muestra dos ejemplos de relajación de un borde, uno en el que una estimación de la ruta más corta disminuye y uno en el que no cambia ninguna estimación.

Figura 24.3: Relajación de un borde (u, v) con peso $w(u, v) = 2$. La estimación de la ruta más corta de cada vértice se muestra dentro del vértice. (a) Porque $d[v] > d[u] + w(u, v)$ antes de relajación, el valor de $d[v]$ disminuye. (b) Aquí, $d[v] \le d[u] + w(u, v)$ antes de la relajación paso, por lo que $d[v]$ no cambia con la relajación.

Cada algoritmo en este capítulo llama INITIALIZE-SINGLE-SOURCE y luego repetidamente relaja los bordes. Además, la relajación es el único medio por el cual el camino más corto estima y los predecesores cambian. Los algoritmos de este capítulo difieren en la cantidad de veces que se relajan cada edge y el orden en que se relajan los bordes. En el algoritmo de Dijkstra y los caminos más cortos algoritmo para gráficos acíclicos dirigidos, cada borde se relaja exactamente una vez. En el Bellman-Ford algoritmo, cada borde se relaja muchas veces.

[1] Puede parecer extraño que el término "relajación" se utilice para una operación que tensa la parte superior Unido. El uso del término es histórico. El resultado de un paso de relajación puede verse como un relajación de la restricción $d[v] \le d[u] + w(u, v)$, que, por la desigualdad del triángulo (Lema 24.10), debe cumplirse si $d[u] = \delta(s, u)$ y $d[v] = \delta(s, v)$. Es decir, si $d[v] \le d[u] + w(u, v)$, no hay "presión" para satisfacer esta restricción, por lo que la restricción es "relajada".

Propiedades de los caminos más cortos y la relajación.

Para probar que los algoritmos de este capítulo son correctos, apelaremos a varias propiedades de caminos más cortos y relajación. Declaramos estas propiedades aquí, y la Sección 24.5 las prueba formalmente. Para su referencia, cada propiedad indicada aquí incluye el lema o número de corolario de la Sección 24.5. Las últimas cinco de estas propiedades, que se refieren a estimaciones de la ruta más corta o el subgráfico predecesor, suponen implícitamente que el gráfico es inicializado con una llamada a INITIALIZE-SINGLE-SOURCE (G, s) y que la única forma en que estimaciones de la ruta más corta y el cambio de subgrafo predecesor son por alguna secuencia de Pasos de relajación.

Desigualdad de triángulos (Lema 24.10)

- Para cualquier borde $(u, v) \in E$, tenemos $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Página 497

Propiedad del límite superior (Lema 24.11)

- Siempre tenemos $d[v] \ge \delta(s, v)$ para todos los vértices $v \in V$, y una vez que $d[v]$ alcanza el valor $\delta(s, v)$, nunca cambia.

Propiedad sin ruta (Corolario 24.12)

- Si no hay un camino de s a v, entonces siempre tenemos $d[v] = delta(s, v) = \infty$.

Propiedad de convergencia (Lema 24.14)

- Si es un camino más corto en G para algunos $u, v \in V$, y si $d[u] = \delta(s, u)$ en cualquier momento antes de relajar el borde (u, v), entonces $d[v] = \delta(s, v)$ en todo momento después.

Propiedad de relajación del camino (Lema 24.15)

- Si $p = v_0, v_1, ..., v_k$ es un camino más corto de $s = v_0$ a v_k, y los bordes de p están relajados en el orden $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$, luego $d[v_k] = \delta(s, v_k)$. Esta propiedad tiene independientemente de cualquier otro paso de relajación que ocurra, incluso si están entremezclados con relajaciones de los bordes de p.

Propiedad de predecesor-subgrafo (Lema 24.17)

- Una vez que $d[v] = \delta(s, v)$ para todo $v \in V$, el subgrafo predecesor es un árbol de rutas más cortas arraigado en s.

Bosquejo del capítulo

La sección 24.1 presenta el algoritmo de Bellman-Ford, que resuelve la fuente única más corta Problema de trayectorias en el caso general en el que las aristas pueden tener un peso negativo. El Bellman El algoritmo de Ford es notable por su simplicidad y tiene el beneficio adicional de detectar si se puede acceder a un ciclo de peso negativo desde la fuente. La sección 24.2 da un tiempo lineal algoritmo para calcular las rutas más cortas de una sola fuente en un gráfico acíclico dirigido. La sección 24.3 cubre el algoritmo de Dijkstra, que tiene un tiempo de ejecución menor que el de Bellman-Algoritmo de Ford, pero requiere que los pesos de los bordes no sean negativos. La sección 24.4 muestra cómo El algoritmo de Bellman-Ford se puede utilizar para resolver un caso especial de "programación lineal". Finalmente, La sección 24.5 demuestra las propiedades de las trayectorias más cortas y la relajación mencionadas anteriormente.

Requerimos algunas convenciones para hacer aritmética con infinitos. Asumiremos que para cualquier número real $a \neq -\infty$, tenemos $a + \infty = \infty + a = \infty$. Además, para que nuestras pruebas se mantengan en el presencia de ciclos de peso negativo, asumiremos que para cualquier número real $a \neq \infty$, tenemos $a + (-\infty) = (-\infty) + a = -\infty$.

Todos los algoritmos de este capítulo asumen que el gráfico dirigido G se almacena en la lista de adyacencia representación. Además, con cada borde se almacena su peso, de modo que a medida que atravesamos cada lista de adyacencia, podemos determinar los pesos de los bordes en $O(1)$ tiempo por borde.

24.1 El algoritmo de Bellman-Ford

Página 498

El *algoritmo de Bellman-Ford* resuelve el problema de las rutas más cortas de una sola fuente en general. caso en el que los pesos de los bordes pueden ser negativos. Dado un gráfico dirigido y ponderado $G = (V, E)$ con fuente s y función de peso $w : E \rightarrow \mathbf{R}$, el algoritmo de Bellman-Ford devuelve un valor booleano valor que indica si hay o no un ciclo de peso negativo que se puede alcanzar desde el fuente. Si existe tal ciclo, el algoritmo indica que no existe una solución. Si no hay tal ciclo, el algoritmo produce los caminos más cortos y sus pesos.

El algoritmo utiliza la relajación, disminuyendo progresivamente una estimación $d[v]$ sobre el peso de un el camino más corto desde la fuente s a cada vértice $v \in V$ hasta que logre el camino más corto real peso $\delta(s, v)$. El algoritmo devuelve VERDADERO si y solo si el gráfico no contiene valores negativos ciclos de peso que son accesibles desde la fuente.

```
BELLMAN-FORD ( G , w , s )
1 INICIALIZAR FUENTE ÚNICA ( G , s )
2 para i ← 1 a | V [ G ] | - 1
3 hacer para cada borde ( u , v ) E [ G ]
4                          no RELAX ( u , v , w )
5 para cada borde ( u , v ) E [ G ]
6 hacer si d [ v ]> d [ u ] + w ( u , v )
7                          luego regresa FALSO
8 devuelve VERDADERO
```

La figura 24.4 muestra la ejecución del algoritmo de Bellman-Ford en un gráfico con 5 vértices. Después de inicializar los valores d y π de todos los vértices en la línea 1, el algoritmo hace $|V| - 1$ pases sobre los bordes del gráfico. Cada pasada es una iteración del ciclo **for** de las líneas 2-4 y consiste de relajar cada borde del gráfico una vez. Las figuras 24.4 (b) - (e) muestran el estado del algoritmo después de que cada uno de los cuatro pase por los bordes. Después de hacer $|V| - 1$ pasa, las líneas 5-8 comprueban si hay un ciclo de peso negativo y devuelve el valor booleano apropiado. (Veremos un poco más tarde por qué este cheque funciona.)

Figura 24.4: La ejecución del algoritmo Bellman-Ford. La fuente es el vértice s . El d los valores se muestran dentro de los vértices, y los bordes sombreados indican valores predecesores: si borde (u, v) está sombreado, entonces $\pi [v] = u$. En este ejemplo particular, cada pasada relaja los bordes en el orden (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y). (a) La situación justo antes la primera pasada sobre los bordes. (b) - (e) La situación después de cada pasada sucesiva sobre los bordes. Los valores $d y \pi$ del *inciso* e) son los valores finales. El algoritmo Bellman-Ford devuelve VERDADERO en este ejemplo.

El algoritmo de Bellman-Ford se ejecuta en el tiempo $O (VE)$, ya que la inicialización en la línea 1 toma $\Theta (V)$ tiempo, cada uno de los $| V | - 1$ pasa sobre los bordes en las líneas 2-4 toma $\Theta (E)$ tiempo, y el bucle **for** de las líneas 5-7 toman tiempo $O (E)$.

Página 499

Para probar la exactitud del algoritmo de Bellman-Ford, comenzamos mostrando que si hay sin ciclos de ponderación negativa, el algoritmo calcula las ponderaciones correctas de la ruta más corta para todos vértices accesibles desde la fuente.

Lema 24.2

Sea $G = (V , E)$ una gráfica dirigida y ponderada con fuente $s y$ función de ponderación $w : E \rightarrow \mathbf{R}$, y suponga que G no contiene ciclos de peso negativo que sean alcanzables desde s . A continuación, después el $| V | - 1$ iteraciones del bucle **for** de las líneas 2-4 de BELLMAN-FORD, tenemos $d [v] = \delta (s , v)$ para todos los vértices v que son accesibles desde s .

Prueba Demostramos el lema apelando a la propiedad de la ruta de relajación. Considere cualquier vértice v que es accesible desde s , y sea $p = v_0 , v_1 , ..., v_k$, Donde $v_0 = s$ y $v_k = v$, ser cualquier acíclico camino más corto desde s a v . La ruta p tiene como máximo $| V | - 1$ aristas, por lo que $k \leq | V | - 1$. Cada uno de los $| V | - 1$ iteraciones del bucle **for** de las líneas 2-4 relaja todos los bordes E. Entre los bordes relajados en la i th iteración, para $i = 1, 2, ..., k$, es (v_{i-1} , v_i). Por la propiedad de relajación de trayectoria, por lo tanto, $d [v] = d [v_k] = \delta (s , v_k) = \delta (s , v)$.

Corolario 24.3

Sea $G = (V , E)$ una gráfica dirigida y ponderada con vértice fuente $s y$ función de ponderación $w : E \rightarrow \mathbf{R}$. Luego, para cada vértice $v V$, hay un camino de s a v si y sólo si Bellman-FORD termina con $d [v] < \infty$ cuando es ejecutado en G .

Prueba La prueba se deja como ejercicio 24.1-2 .

Teorema 24.4: (Corrección del algoritmo de Bellman-Ford)

Que Bellman-FORD puede ejecutar en un ponderada, dirigida gráfico $G = (V , E)$ con fuente $s y$ función de ponderación $w : E \rightarrow \mathbf{R}$. Si G no contiene ciclos de peso negativo que sean accesibles desde s , entonces el algoritmo devuelve VERDADERO, tenemos $d [v] = \delta (s , v)$ para todos los vértices $v V$, y el El subgrafo predecesor G_π es un árbol de rutas más cortas con raíces en s . Si G contiene un negativo ciclo de peso alcanzable desde s , entonces el algoritmo devuelve FALSO.

Prueba Suponga que la gráfica G no contiene ciclos de peso negativo que sean alcanzables desde el fuente s . En primer lugar, demostramos la afirmación de que en la terminación, $d [v] = delta (s , v)$ para todos los vértices $v V$. Si el vértice v es accesible desde s , entonces el Lema 24.2 prueba esta afirmación. Si v no es accesible desde s , entonces la reclamación se deriva de la propiedad no-path. Por lo tanto, se prueba la afirmación. El predecesor La propiedad del subgrafo, junto con la afirmación, implica que G_π es un árbol de caminos más cortos. Ahora usamos el reclamo para demostrar que BELLMAN-FORD devuelve VERDADERO. Al finalizar, tenemos para todos bordes $(u , v) E$, por lo que ninguna de las pruebas en la línea 6 hace que BELLMAN-FORD regrese FALSO. Por tanto, devuelve VERDADERO.

Página 500

$d[v] = \delta(s, v)$
$\leq \delta(s, u) + w(u, v)$ (por la desigualdad del triángulo)
$= d[u] + w(u, v)$,

Por el contrario, suponga que el gráfico G contiene un ciclo de peso negativo que se puede alcanzar desde el fuente s ; sea este ciclo $c = v_0, v_1, ..., v_k$, donde $v_0 = v_k$. Luego,

(24,1)

Suponga, a efectos de contradicción, que el algoritmo de Bellman-Ford devuelve VERDADERO.
Por lo tanto, $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ para $i = 1, 2, ..., k$. Sumando las desigualdades alrededor del ciclo c Nos da

Dado que $v_0 = v_k$, cada vértice en c aparece exactamente una vez en cada una de las sumas y , y entonces

Además, según el Corolario 24.3 , $d[v_i]$ es finito para $i = 1, 2, ..., k$. Así,

lo que contradice la desigualdad (24.1). Concluimos que el algoritmo Bellman-Ford devuelve VERDADERO si el gráfico G no contiene ciclos de peso negativo alcanzables desde la fuente y FALSO de otra manera.

Ejercicios 24.1-1

Ejecute el algoritmo de Bellman-Ford en la gráfica dirigida de la figura 24.4 , utilizando el vértice z como fuente. En cada pasada, relaje los bordes en el mismo orden que en la figura, y muestre d y π valores después de cada pasada. Ahora, cambie el peso del borde (z, x) a 4 y ejecute el algoritmo nuevamente, utilizando s como fuente.

Ejercicio 24.1-2