

هياكل البيانات للمبرمجين

تأليف

Allen B. Downey

ترجمة

رضوى العربي

أكاديمية
حسوب



هياكل البيانات للمبرمجين

مرجع عملي إلى هياكل البيانات والخوارزميات يحتاج إليه كل مهندس برمجيات

Book Title: Think Data Structures

Author: Allen B. Downey

Translator: Radwa Elaraby

Editor: Jamil Bailony - Mostafa Almahmoud

Cover Design: Sirin Diraneyya

Publication Year: 2023

Edition: 1.0

اسم الكتاب: هياكل البيانات للمبرمجين

المؤلف: آلن ب. دوني

المترجم: رضوى العربي

المحرر: جميل بيلوني - مصطفى المحمود

تصميم الغلاف: سيرين ديرانية

سنة النشر:

رقم الإصدار:

بعض الحقوق محفوظة - أكاديمية حسوب.

أكاديمية حسوب أحد مشاريع شركة حسوب محدودة المسؤولية.

مسجلة في المملكة المتحدة برقم 07571594.

<https://academy.hsoub.com>

academy@hsoub.com



Copyright Notice

The author publishes this work under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

This license is acceptable for Free Cultural Works.

The licensor cannot revoke these freedoms as long as you follow the license terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Read the text of the full license on the following link:

<https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode>



The illustrations used in this book is created by the author and all are licensed with a license compatible with the previously stated license.

إشعار حقوق التأليف والنشر

ينشر المصنّف هذا العمل وفقاً لرخصة المشاع الإبداعي نَسَب المصنّف - غير تجاري - الترخيص بالمثل 4.0 دولي (CC BY-NC-SA 4.0).

لك مطلق الحرية في:

- المشاركة — نسخ وتوزيع ونقل العمل لأي وسط أو شكل.
- التعديل — المزج، التحويل، والإضافة على العمل.

هذه الرخصة متوافقة مع أعمال الثقافة الحرة. لا يمكن للمرخص إلغاء هذه الصلاحيات طالما اتبعت شروط الرخصة:

- نَسَب المصنّف — يجب عليك نَسَب العمل لصاحبه بطريقة مناسبة، وتوفير رابط للترخيص، وبيان إذا ما قد أجريت أي تعديلات على العمل. يمكنك القيام بهذا بأي طريقة مناسبة، ولكن على ألا يتم ذلك بطريقة توحي بأن المؤلف أو المرخص مؤيد لك أو لعملك.
- غير تجاري — لا يمكنك استخدام هذا العمل لأغراض تجارية.
- الترخيص بالمثل — إذا قمت بأي تعديل، تغيير، أو إضافة على هذا العمل، فيجب عليك توزيع العمل الناتج بنفس شروط ترخيص العمل الأصلي.

منع القيود الإضافية — يجب عليك ألا تطبق أي شروط قانونية أو تدابير تكنولوجية تقيد الآخرين من ممارسة الصلاحيات التي تسمح بها الرخصة. اقرأ النص الكامل للرخصة عبر الرابط التالي:

الصور المستخدمة في هذا الكتاب من إعداد المؤلف وهي كلها مرخصة برخصة متوافقة مع الرخصة السابقة.

عن الناشر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

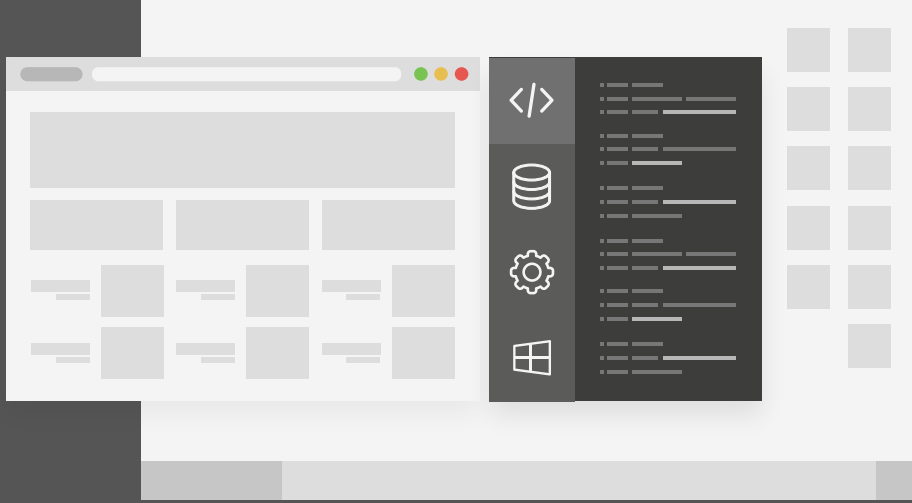


تهدف أكاديمية حسوب إلى تعليم البرمجة باللغة العربية وإثراء المحتوى البرمجي العربي عبر توفير دورات برمجة وكتب ودروس عالية الجودة من متخصصين في مجال البرمجة والمجالات التقنية الأخرى، بالإضافة إلى توفير قسم للأسئلة والأجوبة للإجابة على أي سؤال يواجه المتعلم خلال رحلته التعليمية لتكون معه وتؤهله حتى دخول سوق العمل.



حسوب شركة تقنية في مهمة لتطوير العالم العربي. تبني حسوب منتجات تركز على تحسين مستقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

دورة علوم الحاسوب



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حاسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



المحتويات باختصار

11	تمهيد
16	1. الواجهات Interfaces
22	2. تحليل الخوارزميات
30	3. قائمة المصفوفة ArrayList
42	4. القائمة المترابطة LinkedList
52	5. القائمة ازدواجية الترابط Doubly-Linked List
60	6. التنقل في الشجرة Tree Traversal
70	7. كل الطرق تؤدي إلى روما
77	8. المفهرس Indexer
86	9. الواجهة Map
91	10. التعمية Hashing
99	11. الواجهة HashMap
108	12. الواجهة TreeMap
116	13. شجرة البحث الثنائي Binary Search Tree
126	14. حفظ البيانات عبر Redis
137	15. الزحف على ويكيبيديا
146	16. البحث المنطقي Boolean Search
156	17. الترتيب Sorting

جدول المحتويات

11	تمهيد
11	عن الكتاب
11	فلسفة الكتاب
12	المتطلبات الأساسية
13	0.1: العمل مع الشيفرة
14	المساهمون
15	المساهمة
16	1. الواجهات Interfaces
17	1.1 لماذا هنالك نوعان من الصنف List؟
17	1.2 الواجهات في لغة جافا
18	1.3 الواجهة List
20	1.4 تمرين 1
22	2. تحليل الخوارزميات
23	2.1 الترتيب الانتقائي Selection sort
25	2.2 ترميز Big O
26	2.3 تمرين 2
30	3. قائمة المصفوفة ArrayList
30	3.1 تصنيف توابع الصنف MyArrayList
32	3.2 تصنيف التابع add
35	3.3 حجم المشكلة
36	3.4 هياكل البيانات المترابطة linked data structures
38	3.5 تمرين 3
41	3.6 ملحوظة متعلقة بكنس المهملات garbage collection
42	4. القائمة المترابطة LinkedList
42	4.1 تصنيف توابع الصنف MyLinkedList
45	4.2 الموازنة بين الصنفين MyArrayList و MyLinkedList
46	4.3 التشخيص Profiling

48	4.4 تفسير النتائج
50	4.5 تمرين 4
52	5. القائمة ازدواجية الترابط Doubly-Linked List
52	5.1 نتائج تشخيص الأداء
54	5.2 تشخيص توابع الصنف LinkedList
56	5.3 الإضافة إلى نهاية قائمة من الصنف LinkedList
57	5.4 القوائم ازدواجية الترابط Doubly-linked list
58	5.5 اختيار هيكل البيانات الأنسب
60	6. التنقل في الشجرة Tree Traversal
60	6.1 محركات البحث
61	6.2 تحليل مستند HTML
63	6.3 استخدام مكتبة jsoup
65	6.4 التنقل في شجرة DOM
65	6.5 البحث بالعمق أولاً Depth-first search
66	6.6 المكدسات Stacks في جافا
68	6.7 التنفيذ التكراري لتقنية البحث بالعمق أولاً
70	7. كل الطرق تؤدي إلى روما
70	7.1 البداية
71	7.2 الواجهتان Iterables و Iterators
73	7.3 الصنف WikiFetcher
75	7.4 تمرين 5
77	8. المفهرس Indexer
77	8.1 اختيار هيكل البيانات
79	8.2 الصنف TermCounter
81	8.3 تمرين 6
86	9. الواجهة Map
86	9.1 تنفيذ الصنف MyLinearMap
87	9.2 تمرين 7
88	9.3 تحليل الصنف MyLinearMap

91	10. التعمية Hashing
91	10.1 التعمية Hashing
94	10.2 كيف تعمل التعمية؟
95	10.3 التعمية والقابلية للتغيير mutation
97	10.4 تمرين 8
99	11. الواجهة HashMap
99	11.1 تمرين 9
100	11.2 تحليل الصنف MyHashMap
102	11.3 مقايضات ما بين الزمن والأداء
103	11.4 تشخيص الصنف MyHashMap
104	11.5 إصلاح الصنف MyHashMap
106	11.6 مخططات أصناف UML
108	12. الواجهة TreeMap
108	12.1 ما هي مشكلة التعمية hashing؟
109	12.2 أشجار البحث الثنائية
111	12.3 تمرين 10
112	12.4 تنفيذ الصنف TreeMap
116	13. شجرة البحث الثنائي Binary Search Tree
116	13.1 الصنف MyTreeMap
117	13.2 البحث عن القيم values
119	13.3 تنفيذ التابع put
120	13.4 التنقل بالترتيب In-order
122	13.5 التوابع اللوغاريتمية
124	13.6 الأشجار المتزنة ذاتيا Self-balancing trees
125	13.7 تمرين إضافي
126	14. حفظ البيانات عبر Redis
127	14.1 قاعدة بيانات Redis
128	14.2 خوادم وعملاء Redis
128	14.3 إنشاء مفهرس يعتمد على Redis

131	14.4	أنواع البيانات في قاعدة بيانات Redis
133	14.5	تمرين 11
134	14.6	المزيد من الاقتراحات
135	14.7	تلميحات بسيطة بشأن التصميم
137		15. الزحف على ويكيبيديا
137	15.1	المفهرس المبني على قاعدة بيانات Redis
140	15.2	تحليل أداء عملية البحث
141	15.3	تحليل أداء عملية الفهرسة
142	15.4	التنقل في مخطط graph
143	15.5	تمرين 12
146		16. البحث المنطقي Boolean Search
146	16.1	الزاحف crawler
149	16.2	استرجاع البيانات
149	16.3	البحث المنطقي/الثنائي Boolean search
150	16.4	تمرين 13
152	16.5	الواجهتان Comparable و Comparator
155	16.6	ملحقات
156		17. الترتيب Sorting
157	17.1	الترتيب بالإدراج Insertion sort
159	17.2	تمرين 14
160	17.3	تحليل أداء خوارزمية الترتيب بالدمج
162	17.4	خوارزمية الترتيب بالجذر Radix sort
164	17.5	خوارزمية الترتيب بالكومة Heap sort
165	17.6	الكومة المُقيّدة Bounded heap
166	17.7	تعقيد المساحة Space complexity

تمهيد

عن الكتاب

هذا الكتاب مترجم عن الكتاب الشهير *Think Data Structures* لمؤلفه Allen B. Downey والذي يعد مرجعًا عمليًا في شرح موضوعي هياكل البيانات والخوارزميات اللذين يحتاج إلى تعلمهما كل مبرمج ومهندس برمجيات يتطلع إلى احتراف مهنته وصقل عمله ورفع مستواه.

فلسفة الكتاب

تُعَدُّ هياكل البيانات *data structures* والخوارزميات *algorithms* واحدةً من أهم الاختراعات التي وقعت بالخمسين عامًا الأخيرة، وهي من الأدوات الأساسية التي لا بُدَّ أن يدرسها مهندسي البرمجيات. غالبًا ما تكون الكتب المتناولة لتلك الموضوعات -وفقًا للكاتب- ضخمةً للغاية، كما أنها عادةً ما تُركِّز على الجانب النظري، وتتبع نهج "من أسفل لأعلى" بشكل مفرط.

- **نظرية للغاية:** يعتمد التحليل الحسابي للخوارزميات على افتراضات تبسيطية تُجَدِّد من فائدتها من الناحية العملية. تتناول الكثير من المصادر هذا الموضوع، ولكنها عادةً ما تُركِّز على الجانب الحسابي وتُغفل تلك الافتراضات التبسيطية. في المقابل، يُركِّز هذا الكتاب على الجانب الأكثر عملية من هذا الموضوع ويتجاهل بقية الأجزاء.

- **ضخمة للغاية:** عادةً ما يصل عدد صفحات الكتب التي تتناول هذا الموضوع إلى 500 صفحة على الأقل، بل يبلُغ البعض منها أكثر من 1000 صفحة. بالتركيز على الجوانب التي يراها الكاتب أكثر أهميةً للمهندسي البرمجيات، لا يتعدى هذا الكتاب 200 صفحة.

• **نهج "من أسفل لأعلى" مفرط:** تهتم كثير من كتب هياكل البيانات بطريقة عمل هياكل البيانات (أي طريقة تنفيذها implementations)، ولا تُعطي نفس الأهمية لطريقة إستخدامها (الواجهات interfaces). يتبع هذا الكتاب أسلوبًا مختلفًا، حيث يعتمد على نهج "من أعلى لأسفل"، فيبدأ بالواجهات، وبالتالي يتمكّن القراء من تعلّم كيفية استخدام الهياكل المتاحة بإطار عمل جافا للتجميعات Java Collections Framework قبل أن يتعمقوا بفهم تفاصيل طريقة عملها.

أخيرًا، تُقدّم بعض الكتب هذه المادة العلمية بدون سياق واضح وبدون أي حافز، فتعرض الهياكل البيانية واحدةً تلو الأخرى. يحاول هذا الكتاب تنظيم الموضوعات نوعًا ما من خلال التركيز على تطبيق مُحدّد -محرك بحث-، ويستخدم هذا التطبيق هياكل البيانات بشكل مكثف، وهو في الواقع موضوع مهم وشيق بحد ذاته.

في الحقيقة، سيدفعنا هذا التطبيق إلى دراسة بعض الموضوعات التي ربما لن نتعرّض لها ببعض الفصول الدراسية التمهيدية الخاصة بمادة هياكل البيانات، حيث سنتعرّض هنا مثلاً، لحفظ هياكل البيانات persistent data structure مثل Redis.

اضطرّ الكاتب لاتخاذ بعض القرارات الصعبة المتعلقة بما ينبغي ألا يتضمنه الكتاب، وتوصّل إلى حلول وسط في العموم، إذ يتضمن الكتاب القليل من الموضوعات التي لن يحتاج معظم القراء إلى استخدامها إطلاقًا، ولكنها مع ذلك مهمة، فغالبًا ما سيتوقّع البعض منك معرفتها، خاصةً بمقابلات العمل. وبالنسبة لتلك الموضوعات، سي طرح الكاتب المادة العلمية طرْحًا تقليديًا، كما يمزجه ببعض من آرائه الخاصة ليدفعك إلى التفكير النقدي.

يُقدّم الكتاب أيضًا بعض الأساسيات التي تُمارَس عادةً بهندسة البرمجيات، بما في ذلك نظم التحكم بالإصدار version control، واختبار الوحدات unit testing. تتضمن غالبية فصول الكتاب تمرينًا يسمَح للقراء بتطبيق ما تعلموه خلال الفصل، حيث يُوفّر كل تمرين اختبارات أوتوماتيكية لفحص الحل، وبالإضافة إلى ذلك، يُوفّر الكاتب حلًا لغالبية التمارين ببداية الفصل التالي.

المتطلبات الأساسية

هذا الكتاب مُخصّص لطلبة الجامعات بمجال علوم الحاسوب والمجالات المرتبطة به، ولمهندسي البرمجيات المحترفين، وللمتدربين بمجال هندسة البرمجيات، وكذلك للأشخاص الذين يستعدون لمقابلات العمل التقنية.

ينبغي أن تكون على معرفة جيدة بلغة البرمجة جافا قبل أن تبدأ بقراءة هذا الكتاب. وبالتحديد، لا بُدّ أن تُعرّف كيف تُعرّف صنفًا class جديدًا يمتدّ extend أو يرث من صنف آخر موجود، إلى جانب إمكانية تعريف صنف يُنفذ واجهة interface. إذا لم تكن لديك تلك المعرفة، فيمكنك البدء بأي من الكتابين التاليين:

- Downey and Mayfield, Think Java (O'Reilly Media, 2016): مُخصَّص للقراء الذين لا يملكون أي خبرة بالبرمجة.

- Sierra and Bates, Head First Java (O'Reilly Media, 2005): مناسب للقراء الذين لديهم اطلاع على لغة برمجة أخرى فعليًا.

إذا لم تكن الواجهات interfaces بلغة جافا مألوفةً بالنسبة لك، فيُمكنك قراءة درس ما المقصود بالواجهة؟.

ملحوظة متعلقة بالمصطلحات

قد تكون كلمة واجهة interface مربكةً بعض الشيء. تشير تلك الكلمة ضمن عبارة واجهة تطوير التطبيقات application programming interface إلى مجموعة من الأصناف classes والتوابع methods التي تُوفّر إمكانيات محددة.

علاوةً على ذلك، تشير تلك الكلمة بلغة جافا إلى خاصية ضمن اللغة. تُشبه تلك الخاصية الأصناف وتُخصَّص مجموعة من التوابع، ولكي نتجنَّب الخلط بينهما، سنستخدم كلمة واجهة بنمط الخط العادي للإشارة إلى الفكرة العامة للواجهة، بينما سنستخدم كلمة interface بنمط خط الشيفرة للإشارة إلى تلك الخاصية.

بالإضافة إلى ما سبق، ينبغي أن تكون على علم بكلّ من معاملات الأنواع type parameters والأنواع المُعمَّمة generic types. على سبيل المثال، ينبغي أن تُعرّف كيف تُنشئ كائنًا باستخدام معامل نوع مثل `ArrayList<Integer>`، وإن لم يكن ذلك مألوفًا بالنسبة لك، فيُمكنك القراءة عن معاملات الأنواع.

ينبغي أن تكون على دراية أيضًا بإطار عمل جافا للتجميعات JCF. بالتحديد، لا بُدَّ أن تكون على معرفة بالواجهة List وبالصنفين ArrayList و LinkedList.

من الأفضل لو كنت قد سمعت عن أداة Apache Ant، وهي أداة بناء أوتوماتيكية للغة جافا، كما ينبغي أن تكون على معرفة بإطار عمل جافا لاختبار الوحدات JUnit.

0.1: العمل مع الشيفرة

تتوفّر شيفرة هذا الكتاب بـ مستودع Git. يُعدّ Git نظام تحكم بالإصدار يسمّح لك بتعقّب الملفات التي يتكوّن منها مشروع معين، ويُطلَق اسم مستودع repository على مجموعة الملفات التي يتحكّم بها ذلك النظام.

يُعدّ GitHub خدمة استضافة تُوفّر مساحة تخزين لمستودعات Git مع واجهة إنترنت مناسبة، كما تُوفّر أساليب متعددة للعمل مع الشيفرة، منها:

- يُمكنك أن تُنشئ نسخةً من مستودع مُخزَّن بـ GitHub من خلال النقر على زر "اشتق Fork". إذا لم يكن لديك حساب على الموقع فعلاً، فستحتاج أولاً إلى إنشائه، وبعد إجراء الاشتقاق ستحصل على نسختك من المستودع على GitHub، والتي تستطيع أن تُستخدمها لتعقب الشيفرة التي ستكتبها بنفسك. بهذا يكون قد أصبح بإمكانك نسخ المستودع أي تحميل نسخة من ملفاته إلى حاسوبك.
 - يُمكنك أيضاً نسخ المستودع بدون الاشتقاق. إذا اخترت تلك الطريقة، فلن تكون بحاجة لإنشاء حساب GitHub، ولكنك لن تتمكن من حفظ تعديلاتك على الموقع.
 - إذا لم تكن ترغب باستخدام Git على الإطلاق، فيمكنك أن تُحمّل الشيفرة بهيئة مجلد أرشيفي ZIP باستخدام زر "حمّل Download" بصفحة GitHub أو عبر الرابط <http://thinkdast.com/zip> وبعدما تنتهي من نسخ المستودع أو فك ضغط المجلد الأرشيفي، ستجد مجلد اسمه ThinkDataStructures، وستجد بداخله مجلداً فرعياً اسمه code.
- لقد صُمِّمت أمثلة هذا الكتاب وأُختبِرَت باستخدام الإصدار السابع من عدة تطوير جافا Java SE Development Kit. إذا كنت تستخدم إصداراً أقدم، فقد لا تعمل بعض الأمثلة؛ أما إذا كنت تستخدم إصداراً أحدث، فينبغي أن يعمل جميعها بشكل سليم.

المساهمون

هذا الكتاب عبارة عن نسخة مُعدّلة من منهج دراسي كتبه المؤلف لمدرسة فلاتريون Flatiron بمدينة نيويورك. تُقدّم تلك المدرسة العديد من الدورات المتعلقة بالبرمجة وتطوير الويب عبر الإنترنت، كما أنها تُقدّم دورةً مبنيةً على مادة هذا الكتاب. تُوفّر تلك الدورة بيئة تطوير عبر الإنترنت، ومساعدة من المدربين والطلاب الآخرين، بالإضافة إلى شهادة إكمال الدورة.

يُوجّه المؤلف شكره إلى كل من جو بيرجس Joe Burgess وآن جون Ann John وتشارلز بليتشر Charles Pletcher بمدرسة Flatiron، الذين وفروا التوجيه والاقتراحات والتعديلات بدايةً من التصورات الأولية، وانتهاءً بالتنفيذات والاختبارات.

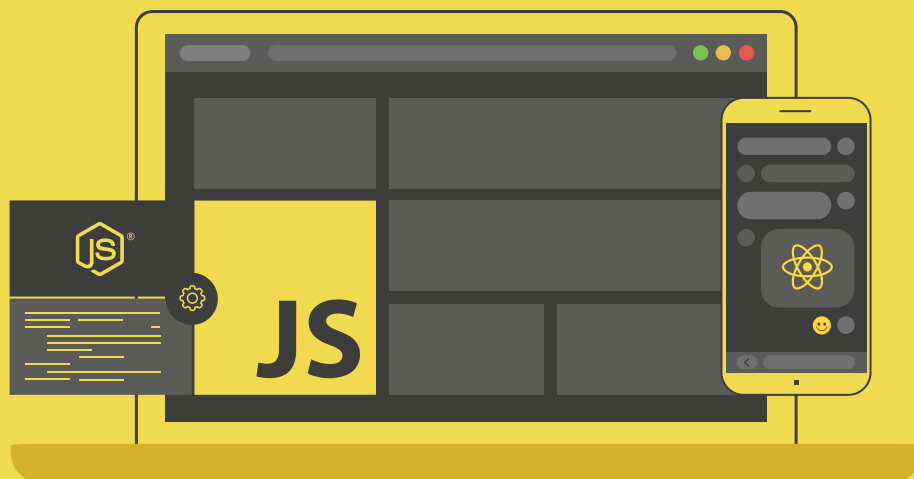
يمتن المؤلف لمراجعيه التقنيين باري ويطمان Barry Whitman وباتريك وايت Patrick White وكريس مايفيلد Chris Mayfield الذين قدموا إليه الكثير من الاقتراحات المفيدة وعثروا على الكثير من الأخطاء، وبالطبع أي أخطاء متبقية فهي من خطأ المؤلف وليس خطأهم.

يشكر المؤلف مدربين وطلاب دورة هياكل البيانات والخوارزميات بكلية Olin الذين قرؤوا الكتاب وأرسلوا إليه رأيهم المفيد.

المساهمة

يرجى إرسال بريد إلكتروني إلى academy@hsoub.com إذا كان لديك اقتراح أو تصحيح على النسخة العربية من الكتاب أو أي ملاحظة حول أي مصطلح من المصطلحات المستعملة. إذا ضمنت جزءاً من الجملة التي يظهر الخطأ فيها على الأقل، فهذا يسهل علينا البحث، ونشكرك أيضاً إن أضفت أرقام الصفحات والأقسام.

دورة تطوير التطبيقات باستخدام لغة JavaScript



احترف تطوير التطبيقات بلغة جافا سكريبت
انطلاقاً من أبسط المفاهيم وحتى بناء تطبيقات حقيقية

التحق بالدورة الآن



1. الواجهات Interfaces

يُقدِّم هذا الكتاب ثلاثة موضوعات:

- هياكل البيانات Data Structures: سناقش هياكل البيانات التي يُوفِّرها إطار التجميعات في لغة جافا Java Collections Framework والتي تُختصرُ إلى JCF، وسنتعلم كيفية استخدام بعض هياكل البيانات مثل القوائم والخرائط وسنرى طريقة عملها.
- تحليل الخوارزميات Algorithms: سنتعرض لتقنيات تساعد على تحليل الشيفرة وعلى التنبؤ بسرعة تنفيذها ومقدار الذاكرة الذي تتطلبه.
- استرجاع المعلومات Information retrieval: سنستخدم الموضوعين السابقين: هياكل البيانات والخوارزميات لإنشاء محرك بحثٍ بسيطٍ عبر الإنترنت، وذلك لنستفيد منهما عملياً ونجعل التمارين أكثر تشويقاً.

وسناقش تلك الموضوعات وفقاً للترتيب التالي:

- سنبدأ بالواجهة List، وسنكتب صنفين ينفذ كلٌ منهما تلك الواجهة بطريقة مختلفة، ثم سنوازن بين هذين الصنفين اللذين كتبناهما وبين صنفي جافا ArrayList وLinkedList.
- بعد ذلك، سنقدِّم هياكل بيانات شجرية الشكل، ونبدأ بكتابة شيفرة التطبيق الأول. حيث سيقراً هذا التطبيق صفحاتٍ من موقع Wikipedia، ثم يُحلَّل محتوياتها ويعطي النتيجة على هيئة شجرة، وفي النهاية سيمر عبر تلك الشجرة بحثاً عن روابط ومزايا أخرى. سنستخدم تلك الأدوات لاختبار الفرضية الشهيرة "الطريق إلى الفلسفة" الذي يمكنك معرفة الفكرة العامة عنه بقراءة المقال باللغة الإنجليزية Getting to Philosophy.

- سنتطرق للواجهة Map وصنف جافا HashMap المُنفَّذ لها، ثم سنكتب أصنافاً تُنفَّذ تلك الواجهة باستخدام جدول hash وشجرة بحث ثنائية.
 - أخيراً، سنستخدم تلك الأصناف وبعض الأصناف الأخرى التي سنتناولها عبر الكتاب لتنفيذ محرك بحث عبر الإنترنت. سيكون هذا المحرك بمنزلة زاحف crawler يبحث عن الصفحات ويقرأها، كما أنه سيُفهرس ويُخزّن محتويات صفحات الإنترنت بهيئةً تُمكنه من إجراء عملية البحث فيها بكفاءة، كما أنه سيعمل مثل مُسترجع للمعلومات، أي أنه سيستقبل استفسارات من المُستخدم ويعيد النتائج ذات الصلة.
- ولنبداً الآن.

1.1 لماذا هنالك نوعان من الصنف List؟

عندما يبدأ المبرمجون باستخدام إطار عمل جافا للتجميعات، فإنهم عادةً يختارون أي الصنفين يختارون ArrayList أم LinkedList. فلماذا تُوفّر جافا تنفيذين implementations للواجهة List؟ وكيف ينبغي الاختيار بينهما؟ سنجيب عن تلك الأسئلة خلال الفصول القليلة القادمة.

سنبدأ باستعراض الواجهات والأصناف المُنفَّذة لها، وسنقدّم فكرة البرمجة إلى واجهة.

سننقذ في التمارين القليلة الأولى أصنافاً مشابهة للصنفين ArrayList و LinkedList، لكي نتمكن من فهم طريقة عملهما، وسنرى أن لكل منهما عيوباً ومميزات، فبعض العمليات تكون أسرع وتحتاج إلى مساحة أقل عند استخدام الصنف ArrayList، وبعضها الآخر يكون أسرع وأصغر عند استخدام الصنف LinkedList، وبهذا يمكن القول: إن تحديد الصنف الأفضل لتطبيقٍ معيّن يعتمد على نوعية العمليات الأكثر استخداماً ضمن ذلك التطبيق.

1.2 الواجهات في لغة جافا

تُحدّد الواجهة بلغة جافا مجموعةً من التوابع methods، ولا بُدّ لأي صنفٍ يُنفَّذ تلك الواجهة أن يُوفّر تلك التوابع. على سبيل المثال، انظر إلى شيفرة الواجهة Comparable المُعرّفة ضمن الحزمة java.lang:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

يستخدم تعريف تلك الواجهة معامل نوع type parameter اسمه T، وبذلك تكون تلك الواجهة من النوع المُعمّم generic type، وينبغي لأي صنفٍ يُنفَّذ تلك الواجهة أن:

- يُحدد النوع الذي يشير إليه معامل النوع T.

- يُوفّر تابعًا اسمه `compareTo` يستقبل كائنًا كمعامل `parameter` ويعيد قيمةً من النوع `int`.

وفي مثالٍ على ما نقول، انظر إلى الشيفرة المصدرية للصف `java.lang.Integer` فيما يلي:

```
public final class Integer extends Number implements
Comparable<Integer> {

    public int compareTo(Integer anotherInteger) {
        int thisVal = this.value;
        int anotherVal = anotherInteger.value;
        return (thisVal<anotherVal ? -1 : (thisVal==anotherVal ? 0 :
1));
    }

    // حُذفت التوابيع الأخرى
}
```

يمتدُّ هذا الصف من الصف `Number`، وبالتالي فإنه يرث التوابيع ومتغيرات التّسخ `instance variables` المُعرّفة في ذلك الصف، كما أنه يُنفذ أيضًا الواجهة `Comparable<Integer>`، ولذلك فإنه يُوفّر تابعًا اسمه `compareTo` ويستقبل معاملاً من النوع `Integer` ويُعيد قيمةً من النوع `int`. عندما يُصرّح صنفٌ معيّن بأنه يُنفذ واجهةً معينة، فإن المُصرّف `compiler` يتأكّد من أن ذلك الصف يُوفّر جميع التوابيع المُعرّفة في تلك الواجهة.

لاحظ أنّ تنفيذ التابع `compareTo` الوارد في الأعلى يستخدم عاملاً ثلاثيًا `ternary operator` يُكتَب أحيانًا على النحو التالي: `?`. إذا لم يكن لديك فكرة عن العوامل الثلاثية، فيمكنك قراءة مقال ما هو العامل الثلاثي؟ (باللغة الإنجليزية).

1.3 الواجهة List

يحتوي إطار التجميعات في لغة جافا JCF على واجهة اسمها `List`، ويُوفّر تنفيذين لها هما `ArrayList` و `LinkedList`. تُعرّف تلك الواجهة ما ينبغي أن يكون عليه الكائن لكي يُمثّل قائمةً من النوع `List`، ومن ثمّ فلا بُدّ أن يُوفّر أي صنفٍ يُنفذ تلك الواجهة مجموعةً محدّدةً من التوابيع، منها `add` و `get` و `remove`، بالإضافة إلى 20 تابعًا آخر.

يُوفّر كلا الصنفين `ArrayList` و `LinkedList` تلك التوابيع، وبالتالي يُمكن التبديل بينهما. ويعني ذلك أنه في حالة وجود تابعٍ مُصمّمٍ ليعمل مع كائنٍ من النوع `List`، فإن بإمكانه العمل أيضًا مع كائنٍ من النوع `ArrayList` أو من النوع `LinkedList`، أو من أيّ نوعٍ آخر يُنفذ الواجهة `List`.

يُوضَّح المثال التالي تلك الفكرة:

```
public class ListClientExample {
    private List list;

    public ListClientExample() {
        list = new LinkedList();
    }

    private List getList() {
        return list;
    }

    public static void main(String[] args) {
        ListClientExample lce = new ListClientExample();
        List list = lce.getList();
        System.out.println(list);
    }
}
```

كما نرى، لا يقوم الصنف `ListClientExample` بعملٍ مفيد، غير أنه يحتوي على بعض العناصر الضرورية لتغليف قائمة من النوع `List`، إذ يتضمَّن متغير نسخة من النوع `List`. سنستخدم هذا الصنف لتوضيح فكرة معينة، ثم سنحتاج إلى استخدامه في التمرين الأول.

يُهيئ باني الصنف `ListClientExample` القائمة `list` باستنساخ `instantiating`-أي بإنشاء- كائنٍ جديدٍ من النوع `LinkedList`، بينما يعيد الجالب `getList` مرجعًا `reference` إلى الكائن الداخلي المُمثل للقائمة، في حين يحتوي التابع `main` على أسطرٍ قليلةٍ من الشيفرة لاختبار تلك التوابع.

النقطة الأساسية التي أردنا الإشارة إليها في هذا المثال هو أنه يحاول استخدام `List`، دون أن يلجأ لتحديد نوع القائمة هل هي `LinkedList` أم `ArrayList` ما لم تكن هناك ضرورة، فكما نرى متغير النسخة كيف أنه مُعرَّف ليكون من النوع `List`، كما أن التابع `getList` يعيد قيمةً من النوع `List`، دون التطرق لنوع القائمة في أيٍّ منهما. وبالتالي إذا غيَّرت رأيك مستقبلاً وقررت أن تستخدم كائنًا من النوع `ArrayList`، فكل ما ستحتاج إليه هو تعديل الباني دون الحاجة لإجراء أي تعديلاتٍ أخرى.

تُطلَق على هذا الأسلوب تسمية البرمجة المعتمدة على الواجهات أو البرمجة إلى واجهة. وللمزيد من المعلومات عنها، يمكنك قراءة المقال البرمجة المعتمدة على الواجهات المتاح بنسخته الإنجليزية للتعرف أكثر

على هذا الأسلوب. تجدر الإشارة هنا إلى أنَّ الكلام هنا عن الواجهات بمفهومها العام وليس مقتصرًا على الواجهات بلغة جافا.

في أسلوب البرمجة المعتمدة على الواجهات، تعتمد الشيفرة المكتوبة على الواجهات فقط مثل `List`، ولا تعتمد على تنفيذات معينة لتلك الواجهات، مثل `ArrayList`. وبهذا، ستعمل الشيفرة حتى لو تغيّر التنفيذ المعتمد عليها في المستقبل.

وفي المقابل، إذا تغيّرت الواجهة، فلا بُدَّ أيضًا من تعديل الشيفرة التي تعتمد على تلك الواجهة، ولهذا السبب يتجنّب مطورو المكتبات تعديل الواجهات إلا عند الضرورة القصوى.

1.4 تمرين 1

نظرًا لأن هذا التمرين هو الأول، فقد حرصنا على تبسيطه. انسخ الشيفرة الموجودة في القسم السابق، وأجرِ التبديل التالي: ضع الصنف `ArrayList` بدلًا من الصنف `LinkedList`. لاحظ هنا أن الشيفرة تُطبّق مبدأ البرمجة إلى واجهة، ولذا فإنك لن تحتاج لتعديل أكثر من سطرٍ واحدٍ فقط وإضافة تعليمة `import`.

لكن قبل كل شيء، يجب ضبط بيئة التطوير المستخدمة؛ كما يجب أيضًا أن تكون عارفًا بكيفية تصريف شيفرات جافا وتشغيلها لكي تتمكن من حل التمارين. وقد طُوّرت أمثلة هذا الكتاب باستخدام الإصدار السابع من عدة تطوير جافا `Java SE Development Kit`، فإذا كنت تستخدم إصدارًا أحدث، فينبغي أن يعمل كل شيء على ما يرام؛ أما إذا كنت تستخدم إصدارًا أقدم، فربما لا تكون الشيفرة متوافقة مع عدة التطوير لديك.

يُفضّل استخدام بيئة تطوير تفاعلية IDE لأنها تُوفّر مزايا إضافية مثل فحص قواعد الصياغة `syntax` والإكمال التلقائي لتعليقات الشيفرة وتحسين هيكلية الشيفرة المصدريّة `refactoring`، وهذا من شأنه أن يُساعدك على تجنّب الكثير من الأخطاء، وعلى العثور عليها بسرعة إن وُجدت، ولكن إذا كنت متقدمًا بطلب وظيفة في شركة ما مثلًا وتنتظر مقابلة عمل تقنية، فهذه الأدوات لن تكون تحت تصرفك غالبًا في أثناء المقابلة، ولهذا لعلّ من الأفضل التعوّد على كتابة الشيفرة بدونها أيضًا.

إذا لم تكن قد حملت الشيفرة المصدريّة للكتاب إلى الآن، فانظر إلى التعليمات في القسم 0.1.

ستجد الملفات والمجلدات التالية داخل مجلد اسمه `code` من مستودع شيفرات الكتاب:

- `build.xml`: هو ملف Ant يساعد على تصريف الشيفرة وتشغيلها.
- `lib`: يحتوي على المكتبات اللازمة لتشغيل الأمثلة (مكتبة `Unit` فقط في هذا التمرين).
- `src`: يحتوي على الشيفرة المصدريّة.

إذا ذهبت إلى المجلد `src/com/allendowney/thinkdast` فستجد ملفات الشيفرة التالية الخاصة

بهذا التمرين:

- `ListClientExample.java`: يحتوي على الشيفرة المصدرية الموجودة في القسم السابق.

- `ListClientExampleTest.java`: يحتوي على اختبارات `JUnit` للصف

`.ListClientExample`

راجع الصف `ListClientExample` وبعد أن تتأكد أنك فهمت كيف يعمل، صرّفه وشغله؛ وإذا كنت

تستخدم أداة `Ant`، فاهرب إلى مجلد `code` ونفذ الأمر `ant ListClientExample`.

ربما تتلقى تحذيرًا يشبه التالي:

List is a raw type. References to generic type List<E> should be parameterized.

سبب ظهور هذا التحذير هو أننا لم نُحدّد نوع عناصر القائمة، وقد فعلنا ذلك بهدف تبسيط المثال، لكن يُمكن حل إشكالية هذا التحذير بتعديل كل `List` أو `LinkedList` إلى `List<Integer>` أو `LinkedList<Integer>` على الترتيب.

يُجري الصف `ListClientExampleTest` اختبارًا واحدًا، يُنشئ من خلاله كائنًا من النوع `ListClientExample`، ويستدعي تابعه الجالب `getList`، ثم يفحص ما إذا كانت القيمة المعادة منه هي كائن من النوع `ArrayList`. سيفشل هذا الاختبار في البداية لأن التابع سيعيد قيمةً من النوع `LinkedList` لا من النوع `ArrayList`، لهذا شغل الاختبار ولاحظ كيف أنه سيفشل.

قد يكون هذا الاختبار مناسبًا لهذا التمرين على وجه الخصوص، ولكنه بشكل عام ليس مثالاً جيدًا على الاختبارات؛ فالاختبارات الجيدة ينبغي أن تتأكد من تلبية الصف الذي يجري اختبارها لمتطلبات الواجهة، لا أن تكون هذه الاختبارات مبنيةً على تفاصيل التنفيذ.

والآن لنعدّل الشيفرة كما يلي: ضع `LinkedList` بدلًا من `ArrayList` ضمن الصف `ListClientExample`، وربما تحتاج أيضًا إلى إضافة تعليمة `import`. صرّف الصف `ListClientExample` وشغله، ثم شغل الاختبار مرةً أخرى. يُفترض أن ينجح الاختبار بعد هذا التعديل.

إن سبب نجاح هذا الاختبار هو تعديلك للتسمية `LinkedList` في باني الصف، دون تعديل لاسم الواجهة `List` في أي مكانٍ آخر. لكن ماذا سيحدث لو فعلت؟ دعنا نجرب. عدّل اسم الواجهة `List` في مكان واحد أو أكثر إلى الصف `ArrayList`، عندها ستجد أن البرنامج ما يزال بإمكانه العمل بشكل صحيح، ولكنه الآن يحدد تفاصيل زائدةً عن الحاجة، وبالتالي إذا أردت أن تُبدّل الواجهة مرةً أخرى في المستقبل، فستضطر إلى إجراء تعديلاتٍ أكثر على الشيفرة.

تُرى، ماذا سيحدث لو استخدمت `List` بدلًا من `ArrayList` داخل باني الصف `ListClientExample`؟ ولماذا لا تستطيع إنشاء نسخة من `List`؟

دورة تطوير التطبيقات باستخدام لغة بايثون



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



2. تحليل الخوارزميات

كما رأينا في الفصل السابق، تُوفّر جافا تنفيذين implementations للواجهة List، هما ArrayList وLinkedList، حيث يكون النوع LinkedList أسرع بالنسبة لبعض التطبيقات، بينما يكون النوع ArrayList أسرع بالنسبة لتطبيقات أخرى.

وإذا أردنا أن نُحدّد أيهما أفضل للاستخدام في تطبيق معين، فيمكننا تجربة كلّ منهما على حدٍ لنرى الزمن الذي سيستغرقه. يُطلق على هذا الأسلوب اسم التشخيص profiling، ولكنّ له بعض الإشكاليات:

1. أننا سنضطر إلى تنفيذ الخوارزميتين كليهما لكي نتمكّن من الموازنة بينهما.
 2. قد تعتمد النتائج على نوع الحاسوب المُستخدَم، فقد تعمل خوارزمية معينة بكفاءةٍ عالية على حاسوب معين، في حين قد تَعْمَل خوارزميةٌ أخرى بكفاءةٍ عالية على حاسوب مختلف.
 3. قد تعتمد النتائج على حجم المشكلة أو البيانات المُدخلة.
- يُمكننا معالجة بعض هذه النقاط المُشكلة بالاستعانة بما يُعرّف باسم تحليل الخوارزميات، الذي يُمكننا من الموازنة بين عدة خوارزميات دون الحاجة إلى تنفيذها فعليًا، ولكننا سنضطر عندئذٍ لوضع بعض الافتراضات:
1. فلنكتفِ نتجّب التفاصيل المتعلقة بعتاد الحاسوب، سنُحدّد العمليات الأساسية التي تتألف منها أي خوارزمية مثل الجمع والضرب وموازنة عددين، ثم نحسب عدد العمليات التي تتطلبها كل خوارزمية.
 2. ولكي نتجّب التفاصيل المتعلقة بالبيانات المُدخلة، فإن الخيار الأفضل هو تحليل متوسط الأداء للمُدخلات التي نتوقع التعامل معها. فإذا لم يكن ذلك متاحًا، فسيكون تحليل الحالة الأسوأ هو الخيار البديل الأكثر شيوعًا.

3. أخيرًا، سيتعين علينا التعامل مع احتمالية أن يكون أداء خوارزمية معينة فعالاً عند التعامل مع مشكلات صغيرة وأن يكون أداء خوارزمية أخرى فعالاً عند التعامل مع مشكلات كبيرة. وفي تلك الحالة، عادةً ما تُركّز على المشكلات الكبيرة، لأن الاختلاف في الأداء لا يكون كبيراً مع المشكلات الصغيرة، ولكنه يكون كذلك مع المشكلات الكبيرة.

يقودنا هذا النوع من التحليل إلى تصنيف بسيط للخوارزميات. على سبيل المثال، إذا كان زمن تشغيل خوارزمية A يتناسب مع حجم المدخلات n ، وكان زمن تشغيل خوارزمية أخرى B يتناسب مع n^2 ، فيمكننا أن نقول إن الخوارزمية A أسرع من الخوارزمية B لقيم n الكبيرة على الأقل.

يُمكن تصنيف غالبية الخوارزميات البسيطة إلى إحدى التصنيفات التالية:

- **ذات زمن ثابت:** تكون الخوارزمية ثابتة الزمن إذا لم يعتمد زمن تشغيلها على حجم المدخلات. على سبيل المثال، إذا كان لدينا مصفوفة مكونة من عدد n من العناصر، واستخدمنا العامل [] لقراءة أيٍّ من عناصرها، فإن ذلك يتطلب نفس عدد العمليات بغض النظر عن حجم المصفوفة.
- **ذات زمن خطي:** تكون الخوارزمية خطية إذا تناسب زمن تشغيلها مع حجم المدخلات. فإذا كنا نحسب حاصل مجموع العناصر الموجودة ضمن مصفوفة مثلاً، فعلينا أن نسترجع قيمة عدد n من العناصر، وأن ننفذ عدد $n-1$ من عمليات الجمع، وبالتالي يكون العدد الكلي للعمليات (الاسترجاع والجمع) هو $2n-1$ ، وهو عدد يتناسب مع n .
- **ذات زمن تربيعي:** تكون الخوارزمية تربيعية أو من الدرجة الثانية إذا تناسب زمن تشغيلها مع n^2 . على سبيل المثال، إذا كنا نريد أن نفحص ما إذا كان هنالك أيُّ عنصر ضمن قائمة معينة مُكرّراً، فإن بإمكان خوارزمية بسيطة أن توازن كل عنصر ضمن القائمة بجميع العناصر الأخرى، وذلك نظراً لوجود عدد n من العناصر، والتي لا بُدّ من موازنة كلٍّ منها مع عدد $n-1$ من العناصر الأخرى، يكون العدد الكلي لعمليات الموازنة هو n^2-n ، وهو عدد يتناسب مع n^2 .

2.1 الترتيب الانتقائي Selection sort

تُنفذ الشيفرة المثال التالية خوارزمية بسيطة تُعرف باسم الترتيب الانتقائي (باللغة الإنجليزية):

```
public class SelectionSort {

    /**
     * بدل العنصرين الموجودين بالفهرس i والفهرس j
     */
    public static void swapElements(int[] array, int i, int j) {
        int temp = array[i];
```

```

        array[i] = array[j];
        array[j] = temp;
    }

    /**
     * اعثر على فهرس أصغر عنصر بدءًا من الفهرس المُمرَّر
     * عبر المعامل index وحتى نهاية المصفوفة
     */
    public static int indexLowest(int[] array, int start) {
        int lowIndex = start;
        for (int i = start; i < array.length; i++) {
            if (array[i] < array[lowIndex]) {
                lowIndex = i;
            }
        }
        return lowIndex;
    }

    /**
     * رتب المصفوفة باستخدام خوارزمية الترتيب الانتقائي
     */
    public static void selectionSort(int[] array) {
        for (int i = 0; i < array.length; i++) {
            int j = indexLowest(array, i);
            swapElements(array, i, j);
        }
    }
}

```

يُبدّل التابع الأول `swapElements` عنصرين ضمن المصفوفة، وتُستغرق عمليتا قراءة العناصر وكتابتها زمنًا ثابتًا؛ لأننا لو عَرَفنا حجم العناصر وموضع العنصر الأول ضمن المصفوفة، فسيكون بإمكاننا حساب موضع أي عنصرٍ آخر باستخدام عمليتي ضربٍ وجمعٍ فقط، وكلتا هاتين العمليتين تستغرقان زمنًا ثابتًا. ولمّا كانت جميع العمليات ضمن التابع `swapElements` تستغرق زمنًا ثابتًا، فإن التابع بالكامل يستغرق بدوره زمنًا ثابتًا.

يبحثُ التابع الثاني `indexLowest` عن فهرس `index` أصغر عنصرٍ في المصفوفة بدءًا من فهرسٍ معينٍ يُخصّصه المعامل `start`، ويقرأ كل تكرارٍ ضمن الحلقة التكراريّة عنصرين من المصفوفة ويوازن بينهما، ونظرًا

لأن كل تلك العمليات تستغرق زمنًا ثابتًا، فلا يهَمُّ أيُّها نَعْدَدُ. ونحن هنا بهدف التبسيط سنحسب عدد عمليات الموازنة:

1. إذا كان `start` يُساوي الصفر، فسيَمُرُّ التابع `indexLowest` عبر المصفوفة بالكامل، وبالتالي يكون عدد عمليات الموازنة المُجرَّاة مُساويًا لعدد عناصر المصفوفة، وليكن n .

2. إذا كان `start` يُساوي 1، فإن عدد عمليات الموازنة يُساوي $n-1$.

3. في العموم، يكون عدد عمليات الموازنة مساويًا لقيمة $n - \text{start}$ ، وبالتالي، يَسْتَغْرِقُ التابع `indexLowest` زمنًا خطيًّا.

يُرتَّبُ التابع الثالث `selectionSort` المصفوفة. ويُنَفَّذُ التابع حلقة تكرار من 0 إلى $n-1$ ، أي يُنفَّذُ الحلقة عدد n من المرات. وفي كل مرة يَسْتَدْعِي خلالها التابع `indexLowest`، ثم يُنفَّذُ العملية `swapElements` التي تَسْتَغْرِقُ زمنًا ثابتًا.

عند استدعاء التابع `indexLowest` لأوّل مرة، فإنه يُنفَّذُ عددًا من عمليات الموازنة مقداره n ، وعند استدعائه للمرة الثانية، فإنه يُنفَّذُ عددًا من عمليات الموازنة مقداره $n-1$ ، وهكذا. وبالتالي سيكون العدد الإجمالي لعمليات الموازنة هو:

$$n + n-1 + n-2 + \dots + 1 + 0$$

يُبلُغ مجموع تلك السلسلة مقدارًا يُساوي $n(n+1)/2$ ، وهو مقدارٌ يتناسب مع n^2 ، مما يَعْنِي أن التابع `selectionSort` يقع تحت التصنيف التربيعي.

يُمَكِّننا الوصول إلى نفس النتيجة بطريقة أخرى، وهي أن ننظر للتابع `indexLowest` كما لو كان حلقة تكرارٍ متداخلةً `nested`، ففي كل مرة نَسْتَدْعِي خلالها التابع `indexLowest`، فإنه يُنفَّذُ مجموعةً من العمليات يكون عددها متناسبًا مع n ، ونظرًا لأننا نَسْتَدْعِيه عددًا من المرات مقداره n ، فإن العدد الكلي للعمليات يكون متناسبًا مع n^2 .

2.2 ترميز Big O

تنتمي جميع الخوارزميات التي تَسْتَغْرِقُ زمنًا ثابتًا إلى مجموعة يُطلَق عليها اسم $O(1)$ ، فإذا قلنا إن خوارزميةً معينة تنتمي إلى المجموعة $O(1)$ ، فهذا يعني ضمنيًا أنها تستغرق زمنًا ثابتًا. وعلى نفس المنوال، تنتمي جميع الخوارزميات الخطيّة -التي تستغرق زمنًا خطيًّا- إلى المجموعة $O(n)$ ، بينما تنتمي جميع الخوارزميات التربيعية إلى المجموعة $O(n^2)$. تطلّق على تصنيف الخوارزميات بهذا الأسلوب تسمية ترميز Big O.

لقد عرّفنا هنا ترميز big O تعريفاً عارضاً، ولكن لو أردت التعمق في الجزء الرياضي منه فإمكانك الاطلاع على مقال ما هو ترميز Big O.

يُوقّر هذا الترميز أسلوباً سهلاً لكتابة القواعد العامة التي تسلكها الخوارزميات في العموم. فلو نفّذنا خوارزميةً خطيةً وتبعناها بخوارزميةً ثابتة الزمن على سبيل المثال، فإن زمن التشغيل الإجمالي يكون خطياً. وننبّه هنا إلى أن \in تعني "ينتمي إلى":

If $f \in O(n)$ and $g \in O(1)$, $f+g \in O(n)$

إذا أجرينا عمليتين خطيتين، فسيكون المجموع الإجمالي خطياً:

If $f \in O(n)$ and $g \in O(n)$, $f+g \in O(n)$

في الحقيقة، إذا أجرينا عمليةً خطيةً أي عددٍ من المرات، وليكن k ، فإن المجموع الإجمالي سيبقى خطياً طالما أن k قيمة ثابتة لا تعتمد على n :

If $f \in O(n)$ and k is constant, $kf \in O(n)$

في المقابل، إذا أجرينا عمليةً خطيةً عدد n من المرات، ts تكون النتيجة تربيعيةً:

If $f \in O(n)$, $nf \in O(n^2)$

وفي العموم، ما يهمنا هو أكبر أسّ للأساس n ، فإذا كان العدد الكليّ للعمليات يُساوي $2n+1$ ، فإنه إجمالاً ينتمي إلى $O(n)$ ، ولا أهمية للثابت 2 ولا للقيمة المضافة 1 في هذا النوع من تحليل الخوارزميات. وبالمثل، ينتمي $n^2+100n+1000$ إلى $O(n^2)$. ولا أهمية للأرقام الكبيرة التي تراها.

يُعدّ ترتيب النمو Order of growth طريقةً أخرى للتعبير عن نفس الفكرة، ويشير ترتيبُ نموٍّ معين إلى مجموعة الخوارزميات التي ينتمي زمن تشغيلها إلى نفس تصنيف ترميز big O، حيث تنتمي جميع الخوارزميات الخطية مثلاً إلى نفس ترتيب النمو؛ وذلك لأن زمن تشغيلها ينتمي إلى المجموعة $O(n)$.

ويُقصد بكلمة "ترتيب" ضمن هذا السياق "مجموعة"، مثل استخدامنا لتلك الكلمة في عبارة مثل "ترتيب فرسان المائدة المستديرة". ويُقصد بهذا أنهم مجموعة من الفرسان، وليس طريقة صقّهم أو ترتيبهم، أي يُمكنك أن تنظر إلى ترتيب الخوارزميات الخطية وكأنها مجموعة من الخوارزميات التي تتمتع بكفاءة عالية.

2.3 تمرين 2

يشتمل التمرين التالي على تنفيذ الواجهة List باستخدام مصفوفةٍ لتخزين عناصر القائمة.

ستجد الملفات التالية في مستودع الشيفرة الخاص بالكتاب -انظر القسم 0.1:-

- `MyArrayList.java`: يحتوي على تنفيذ جزئي للواجهة `List`، فهناك أربعة توابع غير مكتملة عليك أن تكمل كتابة شيفرتها.
- `MyArrayListTest.java`: يحتوي على مجموعة من اختبارات `JUnit`، والتي يُمكنك أن تستخدمها للتحقق من صحة عملك.

كما ستجد الملف `build.xml`، يُمكنك أن تُنفذ الأمر `ant MyArrayList`؛ لكي تتمكن من تشغيل الصنف `MyArrayList.java` وأنت ما تزال في المجلد `code` الذي يحتوي على عدة اختبارات بسيطة. ويُمكنك بدلاً من ذلك أن تُنفذ الأمر `ant MyArrayListTest` لكي تُشغل اختبارات `JUnit`.

عندما تُشغل تلك الاختبارات فسيُفشل بعضها، والسبب هو وجود توابع ينبغي عليك إكمالها. إذا نظرت إلى الشيفرة، فستجد 4 تعليقات `TODO` تشير إلى هذه موضع كل من هذه التوابع.

ولكن قبل أن تبدأ في إكمال تلك التوابع، دعنا نلق نظرة سريعة على بعض أجزاء الشيفرة. تحتوي الشيفرة التالية على تعريف الصنف ومتغيرات الكائنات المنشأة `instance variables` وباني الصنف `constructor`:

```
public class MyArrayList<E> implements List<E> {
    int size;                // احتفظ بعدد العناصر
    private E[] array;       // خزن العناصر

    public MyArrayList() {
        array = (E[]) new Object[10];
        size = 0;
    }
}
```

يحتفظ المتغير `size` -كما يوضح التعليق- بعدد العناصر التي يحملها كائن من النوع `MyArrayList`، بينما يُمثل المتغير `array` المصفوفة التي تحتوي على تلك العناصر ذاتها.

يُنشئ الباني مصفوفةً مكوَّنةً من عشرة عناصر تحمل مبدئياً القيمة الفارغة `null`، كما يضبط قيمة المتغير `size` إلى 0. غالباً ما يكون طول المصفوفة أكبر من قيمة المتغير `size`، مما يعني وجود أماكن غير مُستخدمة في المصفوفة.

ملاحظة متعلقة بقواعد لغة جافا

لا يُمكنك إنشاء مصفوفة باستخدام معامل نوع `type parameter`، وهكذا فالتعليمة التالية مثلاً لن تعمل:
`array = new E[10];`

لكي تتمكن من تخطي تلك العقبة، عليك أن تُنشئ مصفوفةً من النوع `Object`، ثم تُحوّل نوعها `typecast`. يُمكنك قراءة المزيد عن ذلك في ما المقصود بالأنواع المعممة (باللغة الإنجليزية).

ولتلق نظرةً الآن على التابع المسؤول عن إضافة العناصر إلى القائمة:

```
public boolean add(E element) {
    if (size >= array.length) {
        // أنشئ مصفوفة أكبر وانسخ إليها العناصر
        E[] bigger = (E[]) new Object[array.length * 2];
        System.arraycopy(array, 0, bigger, 0, array.length);
        array = bigger;
    }
    array[size] = element;
    size++;
    return true;
}
```

في حالة عدم وجود المزيد من الأماكن الشاغرة في المصفوفة، سنضطر إلى إنشاء مصفوفة أكبر نُنسخ إليها العناصر الموجودة سابقاً، وعندئذٍ ستمكن من إضافة العنصر الجديد إلى تلك المصفوفة، مع زيادة قيمة المتغير `size`.

يعيد ذلك التابع قيمةً من النوع `boolean`. قد لا يكون سبب ذلك واضحاً، فلربما تظن أنه سيعيد القيمة `true` دائماً. قد لا تتضح لنا الكيفية التي ينبغي أن نُحلّ أداء التابع على أساسها. في الأحوال العادية يستغرق التابع زمناً ثابتاً، ولكنه في الحالات التي نضطر خلالها إلى إعادة ضبط حجم المصفوفة سيستغرق زمناً خطياً. وسنتطرق إلى كيفية معالجة ذلك في القسم 3.2.

في الأخير، لتلق نظرةً على التابع `get`، وبعدها يُمكنك البدء في حل التمرين:

```
public T get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return array[index];
}
```

كما نرى، فالتابع `get` بسيط للغاية ويعمل كما يلي: إذا كان الفهرس المطلوب خارج نطاق المصفوفة، فسيُبلّغ التابع عن اعتراض `exception`؛ أما إذا ضمن نطاق المصفوفة، فإن التابع يسترجع عنصر المصفوفة.

ويعيده. لاحظ أن التابع يَفحص ما إذا كانت قيمة الفهرس أقل من قيمة size لا قيمة array.length، وبالتالي لا يعيد التابع قيم عناصر المصفوفة غير المُستخدمة.

ستجد التابع set في الملف MyArrayList.java على النحو التالي:

```
public T set(int index, T element) {
    // TODO: fill in this method.
    return null;
}
```

اقرأ توثيق set باللغة الإنجليزية، ثم أكمل متن التابع. لا بُدَّ أن ينجح الاختبار testSet عندما تُشغّل MyArrayListTest مرةً أخرى.

تجنّب تكرار الشيفرة المسؤولة عن فحص الفهرس.

الخطوة التالية هي إكمال التابع indexOf، وبالمثل نحيلك إلى مقالة توثيق التابع indexOf List لتقرأها أولاً وذلك لتعرف ما ينبغي عليك القيام به. وأعر انتبهاً لكيفية معالجته للقيمة الفارغة null.

وقرنا لك أيضاً التابع المساعد equals للموازنة بين قيمة عنصر ضمن المصفوفة وبين قيمة معينة أخرى. يعيد ذلك التابع القيمة true إذا كانت القيمتان متساويتين كما يُعالج القيمة الفارغة null بشكل سليم. لاحظ أن هذا التابع مُعرّف باستخدام المُعدّل private؛ لأنه ليس جزءاً من الواجهة List، ويُستخدم فقط داخل الصنف.

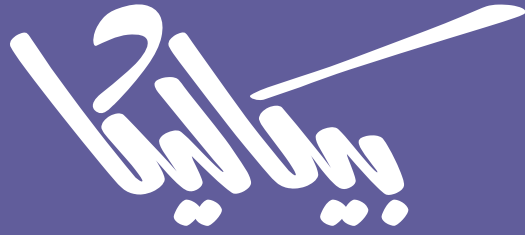
شغّل الاختبار MyArrayListTest مرةً أخرى عندما تنتهي، والآن ينبغي أن ينجح الاختبار testIndexOf وكذلك الاختبارات الأخرى التي تعتمد عليه.

ما يزال هناك تابعان آخران عليك إكمالهما لكي تنتهي من التمرين، حيث أن التابع الأول هو عبارة عن بصمة أخرى من التابع add. تستقبل تلك البصمة فهرساً وتُخزّن فيه قيمةً جديدة. قد تضطر أثناء ذلك إلى تحريك العناصر الأخرى لكي تُوفّر مكاناً للعنصر الجديد.

مثلما سبق، اقرأ التوثيق باللغة الإنجليزية أولاً ثم نفّذ التابع، بعدها شغّل الاختبارات لكي تتأكد من أنك تنفيذك سليم.

تجنّب تكرار الشيفرة المسؤولة عن زيادة/إعادة ضبط حجم المصفوفة.

لنتقل الآن إلى التابع الأخير: أكمل متن التابع remove. اقرأ أولاً التوثيق باللغة الإنجليزية <http://thinkdast.com/listrem>، وعندما تنتهي من إكمال هذا التابع، فالمتوقع أن تنجح جميع الاختبارات. بعد أن تُنهي جميع التوابع وتؤكد من أنها تعمل بكفاءة، يُمكنك الاطلاع على الشيفرة التي كتبها المؤلف.



هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

3. قائمة المصفوفة ArrayList

يُضرب هذا الفصل عصافورين بحجرٍ واحدٍ، حيث سنحل فيه تمرين الفصل السابق، وسنتطرق لوسيلة نصنّف من خلالها الخوارزميات باستخدام ما يسمّى التحليل بالتسديد amortized analysis.

3.1 تصنيف توابع الصنف MyArrayList

يُمكننا تحديد ترتيب نمو order of growth غالبية التوابع بالنظر إلى شيفرتها. على سبيل المثال، انظر إلى تنفيذ التابع `get` المُعرّف بالصنف `MyArrayList`:

```
public E get(int index) {  
    if (index < 0 || index >= size) {  
        throw new IndexOutOfBoundsException();  
    }  
    return array[index];  
}
```

تستغرق كل تعليمة من تعليمات التابع `get` زمناً ثابتاً، وبناءً على ذلك يستغرق التابع `get` في المجمل زمناً ثابتاً.

الآن وقد صنّفنا التابع `get`، يمكننا بنفس الطريقة أن نصنّف التابع `set` الذي يُستخدمه. انظر إلى تنفيذ التابع `set` من التمرين السابق الذي مرّ بنا في الفصل الثاني:

```
public E set(int index, E element) {  
    E old = get(index);  
    array[index] = element;  
}
```

```

        return old;
    }

```

ربما لاحظت أن التابع `set` لا يفحص نطاق المصفوفة صراحةً، فهو يعتمد في ذلك على استدعائه للتابع `get` الذي يُبلغ عن اعتراض `exception` عندما لا يكون الفهرس صالحًا.

تستغرق كل تعليمة من تعليمات التابع `set` -بما في ذلك استدعاؤه للتابع `get`- زمنًا ثابتًا، وعليه يُعدّ التابع `set` ثابت الزمن أيضًا.

ولنتقل الآن إلى بعض التوابيع الخاطئة. انظر مثلاً إلى تنفيذنا للتابع `indexOf`:

```

public int indexOf(Object target) {
    for (int i = 0; i < size; i++) {
        if (equals(target, array[i])) {
            return i;
        }
    }
    return -1;
}

```

يحتوي التابع `indexOf` على حلقة تكرارية كما نرى، وفي كل مرورٍ تكراريٍّ في تلك الحلقة يستدعي التابع `equals`. علينا إذاً أن نُصنّف التابع `equals` أولاً لنتمكن من تصنيف التابع `indexOf`. لننظر إلى تعريف ذلك التابع:

```

private boolean equals(Object target, Object element) {
    if (target == null) {
        return element == null;
    } else {
        return target.equals(element);
    }
}

```

يستدعي التابع السابق التابع `target.equals` الذي يعتمد زمن تنفيذه على حجم المتغير `target` و `element`، ولكنه لا يعتمد على حجم المصفوفة، ولذلك سنعدّه ثابت الزمن لكي نُكمل تحليل التابع `indexOf`.

لنعد الآن إلى التابع `indexOf`. تستغرق كل تعليمة ضمن الحلقة زمنًا ثابتًا، مما يقودنا إلى السؤال التالي: كم عدد مرات تنفيذ الحلقة؟

إذا حالفنا الحظ، قد نجد الكائن المطلوب مباشرةً ونعود بعد اختبار عنصر واحد فقط؛ أما إذا لم نكن محظوظين، فقد نضطرّ لاختبار جميع العناصر. لنقلُ إننا سنحتاج وسطياً إلى اختبار نصف عدد العناصر، ومن ثم يمكن القول بأن هذا التابع يصنّف بأنه تابع خطّي أيضاً باستثناء الحالة الأقل احتمالاً، والتي يكون فيها العنصر المطلوب هو أول عنصر في المصفوفة.

وهكذا يتشابه تحليل التابع remove مع التابع السابق. وفيما يلي تنفيذه:

```
public E remove(int index) {
    E element = get(index);
    for (int i=index; i<size-1; i++) {
        array[i] = array[i+1];
    }
    size--;
    return element;
}
```

يُستدعي التابع السابق التابع get ذا الزمن الثابت، ثم يُمرّر عبر عناصر المصفوفة بدايةً من الفهرس index. وإذا حذفنا العنصر الموجود في نهاية القائمة، فلن يُنفذ التابع حلقة التكرار على الإطلاق، وسيستغرق التابع عندئذٍ زمناً ثابتاً. في المقابل، إذا حذفنا العنصر الأول فسيمرّ التابع عبر جميع العناصر المتبقية، وبالتالي سيستغرق التابع زمناً خطياً. لذلك يُمكننا أن نعدّ التابع خطياً في المجمل، باستثناء الحالة الخاصة التي يكون خلالها العنصر المطلوب حذفه واقعاً في نهاية المصفوفة أو على بعد مسافة ثابتة من نهايتها.

3.2 تصنيف التابع add

تستقبل النسخة التالية من التابع add فهرساً وعنصرًا كعواملات parameters:

```
public void add(int index, E element) {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException();
    }
    // أضف عنصرًا للتأكد من ضبط حجم المصفوفة
    add(element);

    // حرك العناصر الأخرى
    for (int i=size-1; i>index; i--) {
        array[i] = array[i-1];
    }
}
```

```
// ضع العنصر الجديد في المكان الصحيح
array[index] = element;
}
```

تستدعي النسخة ذات المعاملين `add(int, E)` النسخة ذات المعامل الواحد `add(E)` أولاً لكي تضع العنصر الجديد في نهاية المصفوفة، وبعد ذلك تُحرّك العناصر الأخرى إلى اليمين، وتضع العنصر الجديد في المكان الصحيح.

سُحلّل أولاً زمن النسخة ذات المعامل الواحد `add(E)` قبل أن نتقل إلى تحليل النسخة ذات المعاملين `add(int, E)`:

```
public boolean add(E element) {
    if (size >= array.length) {
        // أنشئ مصفوفة أكبر وانسخ العناصر إليها
        E[] bigger = (E[]) new Object[array.length * 2];
        System.arraycopy(array, 0, bigger, 0, array.length);
        array = bigger;
    }
    array[size] = element;
    size++;
    return true;
}
```

تتضح لنا هنا صعوبة تحليل زمن النسخة ذات المعامل الواحد؛ لأنه لو كانت هناك مساحة غير مُستخدمة في المصفوفة، فسيستغرق التابع زمناً ثابتاً؛ أما لو اضطررنا لإعادة ضبط حجم المصفوفة، فسيستغرق التابع زمناً خطياً؛ لأن التابع `System.arraycopy` يستغرق بدوره زمناً يتناسب مع حجم المصفوفة.

إذاً فهل هذا التابع ثابت أم خطي؟ يُمكننا أن نُصنّفه بالتفكير في متوسط عدد العمليات التي تتطلبها عملية الإضافة خلال عدد من الإضافات مقداره n . وسنفترض للتبسيط بأن لدينا مصفوفةً بإمكانها تخزين عنصرين فقط.

- في المرة الأولى التي سنستدعي خلالها `add`، سيجد التابع مساحةً شاغرةً في المصفوفة، وسيُخزّن عنصراً واحداً.
- في المرة الثانية، سيجد التابع مساحةً شاغرةً في المصفوفة، وسيُخزّن عنصراً واحداً.
- في المرة الثالثة، سيعيد التابع ضبط حجم المصفوفة، وينسخ العنصرين السابقين، ثم يخزن العنصر الجديد وهو الثالث، وسيُصبح بإمكان المصفوفة تخزين 4 عناصر في المجل.

- ستُخزَّن المرة الرابعة عنصرًا واحدًا.
- ستعيد المرة الخامسة ضبط حجم المصفوفة، وتنسخ أربعة العناصر السابقة، وتخزَّن عنصرًا جديدًا وهو الخامس، وسيكون في إمكان المصفوفة تخزين ثمانية عناصر إجمالاً.
- ستُخزَّن المرات الثلاث التالية ثلاثة عناصر.
- ستنسخ المرة التالية ثمانية العناصر السابقة وتُخزَّن عنصرًا جديدًا وهو التاسع، وسيُصبح بإمكان المصفوفة تخزين ستة عشر عنصرًا.
- ستُخزَّن سبع المرات التالية سبعة عناصر وهكذا دواليك.

وإذا أردنا أن نلخص ما سبق:

- فإننا بعد 4 إضافات، سنكون قد خزَّنا 4 عناصر ونسخنا عنصرين.
 - بعد 8 إضافات، سنكون قد خزَّنا 8 عناصر ونسخنا 6 عناصر.
 - بعد 16 إضافةً، سنكون قد خزَّنا 16 عنصرًا ونسخنا 14 عنصرًا.
- يُفترض أن تكون قد استقرأت سير العملية وحصلت على ما يلي: لكي نُضيف عدد مقداره n من العناصر، سنضطرّ إلى تخزين عدد n من العناصر ونسخ عدد $n-2$ من العناصر، وبالتالي يكون عدد العمليات الإجمالي هو $n+n-2$ أي $2n-2$.

لكي نحسب متوسط عدد العمليات المطلوبة لعملية الإضافة، ينبغي أن نقسّم العدد الكلي للعمليات على عدد الإضافات n ، وبذلك ستكون النتيجة هي $2-2/n$. لاحظ أنه كلما ازدادت قيمة n ، ستقل قيمة الجزء الثاني $2/n$. ونظرًا لأن ما يهمنا هنا هو الأس الأكبر للأساس n ، فيمكننا أن نعدّ التابع add ثابت الزمن.

قد يبدو من الغريب لخوارزمية تحتاج إلى زمن خطي أحيانًا أن تكون ثابتة الزمن في المتوسط. والفكرة هي أننا نضاعف طول المصفوفة في كل مرة نضطرّ فيها إلى إعادة ضبط حجمها. يُقلّل ذلك عدد المرات التي ننسخ خلالها جميع العناصر، أما لو كنا نضيف مقدارًا ثابتًا إلى طول المصفوفة بدلًا من مضاعفتها بمقدار ثابت، فإنّ هذا التحليل لا يصلح.

يُطلق على تصنيف الخوارزميات وفقًا لتلك الطريقة -أي بحساب متوسط الزمن الذي تستغرقه متتالية من الاستدعاءات- باسم التحليل بالتسديد، والذي يُمكنك قراءة المزيد عنه في مقال [ما هو التحليل بالتسديد؟](#) (باللغة الإنجليزية). تلخص فكرته الأساسية في توزيع/تسديد التكلفة الإضافية لنسخ المصفوفة عبر سلسلة من الاستدعاءات.

الآن وقد عرفنا أنّ التابع $\text{add}(E)$ ثابت الزمن، ماذا عن التابع $\text{add}(int, E)$ ؟ يُنفَّذ التابع $\text{add}(int, E)$ -بعد استدعائه للتابع $\text{add}(E)$ - حلقة تمرّ عبر جزء من عناصر المصفوفة وتُحرّكها. تستغرق تلك

الحلقة زمناً خطئاً باستثناء الحالة التي نضيف خلالها عنصراً إلى نهاية المصفوفة، وعليه يكون التابع `add(int, E)` بدوره خطئاً.

3.3 حجم المشكلة

ولنتقل الآن إلى المثال الأخير في هذا الفصل. انظر فيما يلي إلى تنفيذ التابع `removeAll` ضمن الصنف `MyArrayList`:

```
public boolean removeAll(Collection<?> collection) {
    boolean flag = true;
    for (Object obj: collection) {
        flag &= remove(obj);
    }
    return flag;
}
```

يستدعي التابع `removeAll` في كل تكرار ضمن الحلقة التابع `remove` الذي يستغرق زمناً خطئاً. قد يدفعك ذلك إلى الظن بأنّ التابع `removeAll` تربيعي، ولكن ليس بالضرورة أن يكون كذلك.

يُنَفَّذ التابع `removeAll` الحلقة مرةً واحدةً لكل عنصر في المتغير `collection`. فإذا كان المتغير يحتوي على عدد m من العناصر، وكانت القائمة التي نحذف منها العنصر مكوّنةً من عدد n من العناصر، فإن هذا التابع ينتمي إلى المجموعة $O(nm)$. لو افترضنا أن حجم `collection` ثابت، فسيكون التابع `removeAll` خطئاً بالنسبة لـ n ، ولكن إذا كان حجم `collection` متناسباً مع n ، فسيكون التابع `removeAll` تربيعياً. على سبيل المثال، إذا كان `collection` يحتوي دائماً على 100 عنصر أو أقل، فإن التابع `removeAll` يستغرق زمناً خطئاً؛ أما إذا كان `collection` يحتوي في العموم على 1% من عناصر القائمة، فإن التابع `removeAll` يستغرق زمناً تربيعياً.

عند الحديث عن حجم المشكلة، ينبغي أن ننتبه إلى ماهية الحجم أو الأحجام التي نحن بصددّها. يبين هذا المثال إحدى مشاكل تحليل الخوارزميات، وهي الاختصار المغري الناجم عن عدّ الحلقات، ففي حالة وجود حلقة واحدة، غالباً ما تكون الخوارزمية خطية، وفي حالة وجود حلقتين متداخلتين، فغالباً ما تكون الخوارزمية تربيعية، ولكن انتبه وفكر أولاً في عدد مرات تنفيذ كل حلقة، فإذا كان عددها يتناسب مع n لجميع الحلقات، فيمكنك الاكتفاء بعدّ الحلقات؛ أما إذا لم يكن عددها متناسباً دائماً مع n - كما هو الحال في هذا المثال - فعليك أن تتريث وتمنح الأمر مزيداً من التفكير.

3.4 هياكل البيانات المترابطة linked data structures

سنقدم في التمرين التالي تنفيذًا جزئيًا للواجهة `List`. يُستخدم هذا التنفيذ قائمةً مترابطةً `linked list` لتخزين العناصر، وإذا لم تكن لديك فكرة عن القوائم المترابطة، فيمكنك القراءة عنها في مقال القوائم المترابطة، وسنتطرق لها باختصار هنا.

يُعدّ هياكل البيانات مترابطًا إذا كان مُؤلَّفًا من كائنات يُطلق عليها عادةً اسم عُقد `nodes`، حيث تحتوي العقد على مراجع `references` تشير إلى عقد أخرى. وفي القوائم المترابطة، تحتوي كل عقدة على مرجع إلى العقدة التالية في القائمة. قد تحتوي العقد في أنواع أخرى من هياكل البيانات المترابطة على مراجع تشير إلى عدة عقد، مثل الأشجار `trees` والشُعب `graphs`.

تعرض الشيفرة التالية تنفيذًا بسيطًا لصنف عقدة:

```
public class ListNode {

    public Object data;
    public ListNode next;

    public ListNode() {
        this.data = null;
        this.next = null;
    }

    public ListNode(Object data) {
        this.data = data;
        this.next = null;
    }

    public ListNode(Object data, ListNode next) {
        this.data = data;
        this.next = next;
    }

    public String toString() {
        return "ListNode(" + data.toString() + ")";
    }
}
```


يتضمن الصنف `ListNode` متغيري نسخة هما: `data` و `next`. يحتوي `data` على مرجع يشير إلى كائن ما من النوع `Object`، بينما يحتوي `next` على مرجع يشير إلى العقدة التالية في القائمة. ويحتوي `next` في العقدة الأخيرة من القائمة على القيمة الفارغة `null` كما هو متعارف عليه.

يُعرّف الصنف `ListNode` مجموعةً من البواني `constructors` التي تُمكنك من تمرير قيمٍ مبدئيةٍ للمتغيرين `data` و `next`، أو تمكّنك من مجرد تهيئتهما إلى القيمة الافتراضية `null`. ويُمكنك أن تُفكر في عقدةٍ واحدةٍ من النوع `ListNode` كما لو أنها قائمةٌ مُكوّنةٌ من عنصرٍ واحدٍ، ولكن على العموم، يُمكن لأي قائمة أن تحتوي على أي عدد من العقد.

هناك الكثير من الطرق المستخدمة لإنشاء قائمة جديدة، وتتكوّن إحداها من إنشاء مجموعة من كائنات الصنف `ListNode` على النحو التالي:

```
ListNode node1 = new ListNode(1);
ListNode node2 = new ListNode(2);
ListNode node3 = new ListNode(3);
```

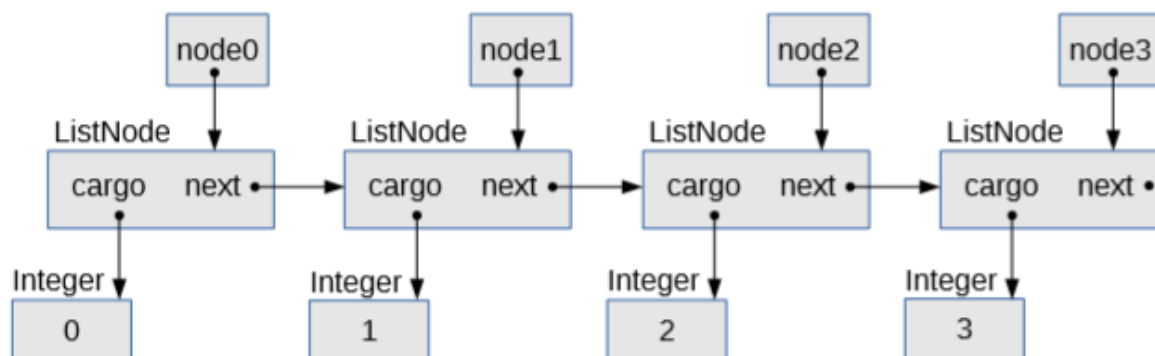
ثم ربطها ببعض:

```
node1.next = node2;
node2.next = node3;
node3.next = null;
```

وهناك طريقة أخرى هي أن تُنشئ عقدةً وتربطها في نفس الوقت. على سبيل المثال، إذا أردت أن تضيف عقدةً جديدةً إلى بداية قائمة، يُمكنك كتابة ما يلي:

```
ListNode node0 = new ListNode(0, node1);
```

والآن، بعد تنفيذ سلسلة التعليمات السابقة، أصبح لدينا أربعة عقدٍ تحتوي على الأعداد الصحيحة 0 و 1 و 2 و 3 مثل بيانات، ومربوطةً معًا بترتيب تصاعدي. لاحظ أن قيمة `next` في العقدة الأخيرة تحتوي على القيمة الفارغة `null`.



الرسم التوضيحية السابقة هي رسم بياني لكائن يُوضّح المتغيرات والكائنات التي تشير إليها تلك المتغيرات. تظهر المتغيرات بهيئة أسماء داخل صناديق مع أسهم تُوضّح ما تشير إليه المتغيرات، بينما تظهر الكائنات بهيئة صناديق تجدها خارج النوع الذي تنتمي إليه (مثل Integer و ListNode)، وداخلها متغيرات النسخ المُعرّفة بها.

3.5 تمرين 3

ستجد ملفات الشيفرة المطلوبة لهذا التمرين في مستودع الكتاب.

- `MyLinkedList.java`: يحتوي على تنفيذ جزئي للواجهة `List`، ويستخدم قائمة مترابطة لتخزين العناصر.

- `MyLinkedListTest.java`: يحتوي على اختبارات `JUnit` للصف `MyLinkedList`.

نفذ الأمر `ant MyArrayList` لتشغيل `MyArrayList.java` الذي يحتوي على عدة اختبارات بسيطة. ثم نفذ `ant MyArrayListTest` لتشغيل اختبارات `JUnit` التي سيفشل البعض منها. إذا نظرت إلى الشيفرة، ستجد ثلاثة تعليقات `TODO` للإشارة إلى التوابع التي ينبغي عليك إكمالها.

لننظر إلى بعض أجزاء الشيفرة قبل أن تبدأ. انظر إلى المتغيرات والبواني المُعرّفة في الصف `MyLinkedList`:

```

public class MyLinkedList<E> implements List<E> {

    private int size;           // احتفظ بعدد العناصر
    private Node head;          // مرجع إلى العقدة الأولى

    public MyLinkedList() {
        head = null;
    }
}

```

```

        size = 0;
    }
}

```

يحتفظ المتغير `size` -كما يُوضَّح التعليق- بعدد العناصر التي يحملها كائن من النوع `MyLinkedList`، بينما يشير المتغير `head` إلى العقدة الأولى في القائمة أو يحمل القيمة الفارغة `null` إذا كانت القائمة فارغة. ليس من الضروري تخزين عدد العناصر. ولا يُعدّ الاحتفاظ بمعلومات إضافية في العموم أمرًا جيدًا؛ والسبب هو أن المعلومات الإضافية تحمّلنا عبء التحديث المستمر لها، ما قد يتسبّب في وقوع أخطاء، كما أنها تحتل حيزًا إضافيًا من الذاكرة.

لكننا لو خزّنا `size` صراحةً، فإننا سنتمكّن من كتابة تنفيذٍ للتابع `size`، بحيث يستغرق تنفيذه زمنًا ثابتًا؛ أما لو لم نفعل ذلك، فسنضطرّ إلى المرور عبر القائمة وعدّ عناصرها في كل مرة، وهذا يتطلب زمنًا خطيًا. من جهة أخرى، نظرًا لأننا نخزّن `size` صراحةً، فإننا سنضطرّ إلى تحديثه في كل مرة نضيف فيها عنصرًا إلى القائمة أو نحذفه منها. يؤدي ذلك إلى إبطاء تلك التتابع نوعًا ما، ولكنه لن يؤثّر على ترتيب النمو الخاص بها، ولذلك فلربما الأمر يستحق.

يضبط الباني قيمة `head` إلى `null`، مما يشير إلى كون القائمة فارغة، كما يضبط `size` إلى صفر. يستخدم هذا الصنف معامل نوع `parameter type` اسمه `E` لتخصيص نوع العناصر. إذا لم تكن على معرفة بمعاملات الأنواع، اقرأ هذا الدرس (باللغة التي تريد).

يظهر معامل النوع أيضًا بتعريف الصنف `Node` المُضمّن داخل الصنف `MyLinkedList`. انظر تعريفه:

```

private class Node {
    public E data;
    public Node next;

    public Node(E data, Node next) {
        this.data = data;
        this.next = next;
    }
}

```

أضف إلى هذا أن الصنف `Node` مشابه تمامًا للصنف `ListNode` في الأعلى.

والآن، انظر إلى تنفيذ التابع `add`:

```

public boolean add(E element) {
    if (head == null) {
        head = new Node(element);
    } else {
        Node node = head;
        // نفذ الحلقة حتى تصل إلى العقدة الأخيرة
        for ( ; node.next != null; node = node.next) {}
        node.next = new Node(element);
    }
    size++;
    return true;
}

```

يُوضّح هذا المثال نمطين ستحتاج لمعرفةهما لإكمال حل التمرين:

1. في كثير من التوابع، عادةً ما نضطرّ إلى معالجة أول عنصرٍ في القائمة بطريقةٍ خاصة. وفي هذا المثال، إذا كنا نضيف العنصر الأول إلى القائمة، فعلينا أن نُعدّل قيمة head، أما في الحالات الأخرى، فعلينا أن نجتاز القائمة، حتى نصل إلى نهايتها، ثم نضيف العقدة الجديدة.

2. يُبيّن ذلك التابع طريقة استخدام حلقة التكرار for من أجل اجتياز أو التنقل بين العقد الموجودة في القائمة. في مثالنا هذا لدينا حلقة واحدة. ولكن في الواقع قد تحتاج إلى كتابة نسخٍ عديدةٍ من تلك الحلقة ضمن الحلول الخاصة بك. من جهةٍ أخرى، لاحظ كيف صرّحنا عن node قبل بداية الحلقة؛ والهدف من ذلك هو أن نتمكّن من استرجاعها بعد انتهاء الحلقة.

والآن حان دورك، أكمل متن التابع indexOf. ويجب أن تقرأ مقال توثيق List indexOf لكي تعرف ما ينبغي عليك القيام به. انتبه تحديداً للطريقة التي يُفترض له معالجة القيمة الفارغة null بها.

كما هو الحال في تمرين الفصل السابق، وقّرنّا التابع المساعد equals للموازنة بين قيمة عنصرٍ ضمن المصفوفة وبين قيمةٍ معينةٍ أخرى، وفحص ما إذا كانت القيمتان متساويتين. يُعالج التابع القيمة الفارغة معالجةً سليمة. لاحظ أن التابع مُعرّف باستخدام المُعدّل private؛ لأنه ليس جزءاً من الواجهة List، ويُستخدم فقط داخل الصنف.

شغلّ الاختبارات مرةً أخرى عندما تنتهي. ينبغي أن ينجح الاختبار testIndexOf وكذلك الاختبارات الأخرى التي تعتمد عليه.

والآن، عليك أن تكمل نسخة التابع add ذات المعاملين. تستقبل تلك النسخة فهرساً وتُخزّن القيمة الجديدة في الفهرس المُمرّر. وبالمثل، اقرأ أولاً التوثيق List add ثم نفذ التابع، وأخيراً، شغلّ الاختبارات لكي تتأكّد من أنك نفّذتها بشكل سليم.

لننتقل الآن إلى التابع الأخير: أكمل متن التابع `remove`. اقرأ توثيق التابع `List remove`. بعدما تنتهي من إكمال هذا التابع، ينبغي أن تنجح جميع الاختبارات.

بعد أن تُنهي جميع التوابع وتؤكد من أنها تعمل بكفاءة، يُمكنك مقابقتها مع النسخ المتاحة في مجلد `solution` في مستودع الكتاب.

3.6 ملحوظة متعلقة بكنس المهملات garbage collection

تنمو المصفوفة في الصنف `MyArrayList` -من التمرين المشار إليه- عند الضرورة، ولكنها لا تقلص أبدًا، وبالتالي لا تُكنس المصفوفة ولا يُكنس أي من عناصرها حتى يحين موعد تدمير القائمة ذاتها. في المقابل، تقلص القائمة المترابطة عند حذف العناصر منها، كما تُكنس العقد غير المُستخدمة مباشرةً، وهو ما يُمثل واحدةً من مميزات هذا النوع من هياكل البيانات.

انظر إلى تنفيذ التابع `clear`:

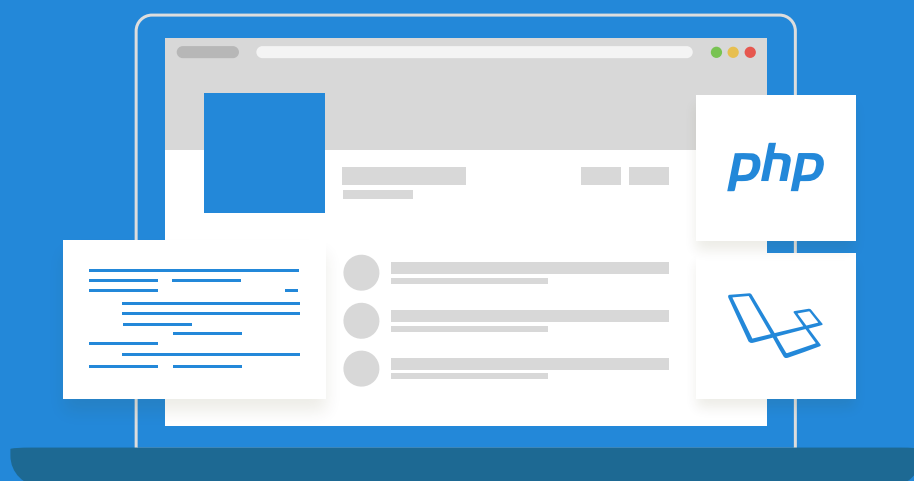
```
public void clear() {
    head = null;
    size = 0;
}
```

عندما نضبط قيمة الرأس `head` بالقيمة `null`، فإننا نحذف المرجع إلى العقدة الأولى. إذا لم تكن هناك أي مراجع أخرى إلى ذلك الكائن (من المفترض ألا يكون هناك أي مراجع أخرى)، فسيُكنس الكائن مباشرةً. في تلك اللحظة، سيُحذف المرجع إلى الكائن المُمثل للعقدة الثانية، وبالتالي يُكنس بدوره أيضًا. وستستمر تلك العملية إلى أن تُكنس جميع العقد.

بناءً على ما سبق، ما هو تصنيف التابع `clear`؟ يتضمّن التابع عمليتين ثابتتي الزمن، ويبدو لهذا كما لو أنه يستغرق زمنًا ثابتًا، ولكنك عندما تستدعيه ستُحفر كنس المهملات على إجراء عملية تتناسب مع عدد العناصر، ولذلك ربما علينا أن نُعدّه خطّي الزمن.

يُعدّ هذا مثالاً على ما يُسميه أحيانًا بمشكلة برمجية في الأداء، أي أن البرنامج يفعل الشيء الصحيح، ولكنه لا ينتمي إلى ترتيب النمو المُتوقع. يصعب العثور على هذا النوع من الأخطاء خاصةً في اللغات التي تُنفّذ أعمالاً كثيرةً وراء الكواليس مثل عملية كنس المهملات مثلاً، وتُعدّ لغة جافا واحدةً من تلك اللغات.

دورة تطوير تطبيقات الويب باستخدام لغة PHP



احترف تطوير النظم الخلفية وتطبيقات الويب
من الألف إلى الياء دون الحاجة لخبرة برمجية مسبقة

التحق بالدورة الآن



4. القائمة المترابطة LinkedList

سنتناول في هذا الفصل حل تمرين الفصل الثالث، ثم نتابع مناقشة تحليل الخوارزميات.

4.1 تصنيف توابع الصنف MyLinkedList

تُعرض الشيفرة التالية تنفيذ التابع `indexOf`. اقرأها وحاول تحديد ترتيب نمو `order of growth` التابع قبل المتابعة:

```
public int indexOf(Object target) {  
    Node node = head;  
    for (int i=0; i<size; i++) {  
        if (equals(target, node.data)) {  
            return i;  
        }  
        node = node.next;  
    }  
    return -1;  
}
```

تُسند `head` إلى `node` أولاً، ويعني هذا أنّ كليهما يشيران الآن إلى نفس العقدة. يُعدّ المُتغيّر `i` هو المُتحكّم بالحلقة من 0 إلى `size-1`، ويُسندعي في كل تكرار التابع `equals` ليفحص ما إذا كان قد وجد القيمة المطلوبة. فإذا وجدها، فسيعيد قيمة `i` على الفور؛ أما إذا لم يجدها، فسينتقل إلى العقدة التالية ضمن القائمة.

عادةً ما نتأكد أولاً مما إذا كانت العقدة التالية لا تحتوي على قيمة فارغة `null`، ولكن ليس هذا ضروريًا في حالتنا؛ لأن الحلقة تنتهي بمجرد وصولنا إلى نهاية القائمة (بفرض أن قيمة `size` مُتّسقة مع العدد الفعلي للعقد الموجودة ضمن القائمة).

إذا نفّذنا الحلقة بالكامل دون العثور على القيمة المطلوبة، فسيعيد التابع القيمة -1.

والآن، ما هو ترتيب نمو هذا التابع؟

1. إننا نستدعي في كل تكرار التابع `equals` الذي يستغرق زمنًا ثابتًا (قد يعتمد على حجم `target` أو `data`، ولكنه لا يعتمد على حجم القائمة). تستغرق جميع التعليمات الأخرى ضمن الحلقة زمنًا ثابتًا أيضًا.

2. تُنفّذ الحلقة عددًا من المرات مقداره `n` لأننا قد نضطر إلى التنقل في القائمة بالكامل في الحالة الأسوأ.

وبالتالي يتناسب زمن تنفيذ ذلك التابع مع طول القائمة.

والآن، انظر إلى تنفيذ التابع `add` ذي المعاملين، وحاول تصنيفه قبل متابعة القراءة.

```
public void add(int index, E element) {
    if (index == 0) {
        head = new Node(element, head);
    } else {
        Node node = getNode(index-1);
        node.next = new Node(element, node.next);
    }
    size++;
}
```

إذا كان `index` يُساوي الصفر، فإننا نضيف العقدة الجديدة إلى بداية القائمة، ولهذا علينا أن نعالج ذلك مثل حالة خاصة. وبخلاف ذلك سنضطر إلى التنقل في القائمة إلى أن نصل إلى العنصر الموجود في الفهرس `index-1`، كما سنستخدم لهذا الهدف التابع المساعد `getNode`، وفيما يلي شيفرته:

```
private Node getNode(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    Node node = head;
    for (int i=0; i<index; i++) {
        node = node.next;
    }
}
```



```

    }
    return node;
}

```

يفحص التابع `getNode` ما إذا كانت قيمة `index` خارج النطاق المسموح به، فإذا كانت كذلك، فإنه يُبلِّغ عن اعتراض `exception`؛ أما إذا لم تكن كذلك، فإنه يمرّ عبر عناصر القائمة ويعيد العقدة المطلوبة. الآن وقد حصلنا على العقدة المطلوبة، نعود إلى التابع `add` وننشئ عقدةً جديدةً، ونضعها بين العقدين `node` و `node.next`. قد يساعدك رسم هذه العملية على التأكد من فهمها بوضوح.

والآن، ما هو ترتيب نمو التابع `add`؟

1. يشبه التابع `getNode` التابع `indexOf`، وهو خطّي لنفس السبب.
2. تستغرق جميع التعليمات زمنًا ثابتًا سواءً قبل استدعاء التابع `getNode` أو بعد استدعائه ضمن التابع `add`.

وعليه، يكون التابع `add` خطّيًا.

وأخيرًا، لنلق نظرةً على التابع `remove`:

```

public E remove(int index) {
    E element = get(index);
    if (index == 0) {
        head = head.next;
    } else {
        Node node = getNode(index-1);
        node.next = node.next.next;
    }
    size--;
    return element;
}

```

يُستدعى `remove` التابع `get` للعثور على العنصر الموجود في الفهرس `index` ثم عندما يجده يحذف العقدة التي تتضمنه.

إذا كان `index` يُساوي الصفر، نُعالج ذلك مثل حالة خاصة. وإذا لم يكن يساوي الصفر فسنذهب إلى العقدة الموجودة في الفهرس `index-1`، وهي العقدة التي تقع قبل العقدة المستهدفة بالحذف، ونُعدّل حقل `next` فيها ليشير مباشرةً إلى العقدة `node.next.next`، وبذلك نكون قد حذفنا العقدة `node.next` من

القائمة، ومن ثم تُحرّر الذاكرة التي كانت تحتلها عن طريق الكنس garbage collection. وأخيرًا، يُنقص التابع قيمة size ويُعيد العنصر المُسترجع في البداية.

والآن بناءً على ما سبق، ما هو ترتيب نمو التابع remove؟ تستغرق جميع التعليمات في ذلك التابع زمنًا ثابتًا باستثناء استدعائه للتابعين get وgetNode اللذين يستغرقان زمنًا خطيًا. وبناءً على ذلك يكون التابع remove خطيًا هو الآخر.

يظن بعض الأشخاص عندما يرون عمليتين خطيتين أن النتيجة الإجمالية ستكون تربيعية، ولكن هذا ليس صحيحًا إلا إذا كانت إحدهما داخل الأخرى؛ أما إذا استدعيت إحدهما تلو الأخرى، فستُحسب المُحصلة بجمع زمنيها، ولأن كليهما ينتميان إلى المجموعة $O(n)$ ، فسينتمي المجموع إلى $O(n)$ أيضًا.

4.2 الموازنة بين الصنفين MyLinkedList و MyArrayList

يُلخّص الجدول التالي الاختلافات بين الصنفين MyArrayList و MyLinkedList. يشير 1 إلى المجموعة $O(1)$ أو الزمن الثابت، بينما يشير n إلى المجموعة $O(n)$ أو الزمن الخطي:

MyLinkedList	MyArrayList	
n	1	add (في النهاية)
1	n	add (في البداية)
n	n	add (في العموم)
n	1	get / set
n	n	indexOf / lastIndexOf
1	1	isEmpty / size
n	1	remove (من النهاية)
1	n	remove (من البداية)
n	n	remove (في العموم)

في حالتي إضافة عنصر أو حذفه من نهاية القائمة، فإنّ الصنف MyArrayList هو أفضل من نظيره MyLinkedList، وكذلك في عمليتي الاسترجاع والتعديل؛ أمّا في حالتي إضافة عنصر أو حذفه من مقدّمة القائمة، فإنّ الصنف MyLinkedList هو أفضل من نظيره MyArrayList.

يحظى الصنفان بنفس ترتيب النمو بالنسبة للعمليات الأخرى.

إدًا، أيهما أفضل؟ يعتمد ذلك على العمليات التي يُحتمل استخدامها أكثر، وهذا السبب هو الذي يجعل جافا تُوفّر أكثر من تنفيذٍ implementation وحيد.

4.3 التشخيص Profiling

ستحتاج إلى الصنف Profiler في التمرين التالي. يحتوي هذا الصنف على شيفرة بإمكانها أن تُشغَّل تبعًا ما على مشاكل ذات أحجام متفاوتة، وتقيس زمن التشغيل لكلٍّ منها، وتعرض النتائج.

ستستخدم الصنف Profiler لتصنيف أداء التابع add المُعرَّف في كلٍّ من الصنفين ArrayList وLinkedList اللذين تُوفرهما لغة جافا.

توضِّح الشيفرة التالية طريقة استخدام ذلك الصنف:

```
public static void profileArrayListAddEnd() {
    Timeable timeable = new Timeable() {
        List<String> list;

        public void setup(int n) {
            list = new ArrayList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add("a string");
            }
        }
    };

    String title = "ArrayList add end";
    Profiler profiler = new Profiler(title, timeable);

    int startN = 4000;
    int endMillis = 1000;
    XYSeries series = profiler.timingLoop(startN, endMillis);
    profiler.plotResults(series);
}
```

يقيس التابع السابق الزمن الذي يستغرقه تشغيل التابع add الذي يُضيف عنصرًا جديدًا في نهاية قائمة من النوع ArrayList. سنشرح الشيفرة أولًا ثم نعرض النتائج.

لكي يتمكن الصنف `Profiler` من أداء عمله، سنحتاج أولاً إلى إنشاء كائنٍ من النوع `Timeable`. يُوفّر هذا الكائنُ التابعين `setup` و `timeMe`، حيث يُنفَّذ التابع `setup` كل ما ينبغي فعله قبل بدء تشغيل المؤقت، وفي هذا المثال سيُنشئ قائمةً فارغة، أمّا التابع `timeMe` فيُنفَّذ العملية التي نحاول قياس أدائها. في هذا المثال، سنجعله يُضيف عددًا من العناصر مقداره `n` إلى القائمة.

لقد عرّفنا الشيفرة المسؤولة عن إنشاء المتغير `timeable` ضمن صنفٍ مجهول الاسم `anonymous`، حيث يُعرّف ذلك الصنف تنفيذاً جديداً للواجهة `Timeable`، ويُنشئ نسخةً من الصنف الجديد في نفس الوقت. إذا لم يكن لديك فكرة عن الأصناف مجهولة الاسم، فسنحيلك إلى المقالة ما هي الأصناف مجهولة الاسم؟ (باللغة الإنجليزية) والفصل الأصناف المتداخلة `Nested Classes` في جافا.

ولكنك على كل حالٍ لست في حاجةٍ إلى معرفة الكثير عنها لحل التمرين التالي، ويُمكنك نسخ شيفرة المثال وتعديلها.

والآن سننتقل إلى الخطوة التالية، وهي إنشاء كائنٍ من الصنف `Profiler` مع تمرير معامليْن له هما: العنوان `title` وكائنٍ من النوع `Timeable`.

يحتوي الصنف `Profiler` على التابع `timingLoop`. يستخدم ذلك التابع الكائن `Timeable` المُخزّن مثل متغيّرٍ نُسخةٍ `instance variable`، حيث يَستدعيّ تابعه `timeMe` عدة مراتٍ مع قيم مختلفةٍ لحجم المشكلة `n` في كل مرة. ويَستقبِل التابع `timingLoop` المعامليْن التاليين:

- `startN`: هي قيمة `n` التي تبدأ منها الحلقة.
- `endMillis`: عبارة عن قيمة قصوى بوحدة الملي ثانية. يزداد زمن التشغيل عندما يُزيد التابع `timingLoop` حجم المشكلة، وعندما يتجاوز ذلك الزمنُ القيمةَ القصوى المُحدّدة، فينبغي أن يتوقف التابع `timingLoop`.

قد تضطرّ إلى ضبط قيم تلك المعاملات عند إجراء تلك التجارب، فإذا كانت قيمة `startN` ضئيلةً للغاية، فلن يكون زمن التشغيل طويلاً بما يكفي لقياسه بدقّة، وإذا كانت قيمة `endMillis` صغيرةً للغاية، فقد لا تحصل على بياناتٍ كافيةٍ لاستنباط العلاقة بين حجم المشكلة وزمن التشغيل.

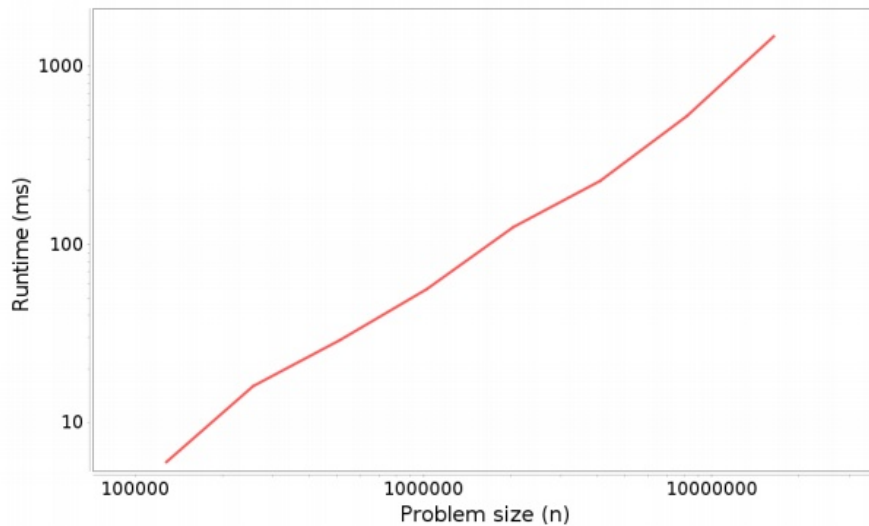
ستجد تلك الشيفرة -التي ستحتاج إليها في التمرين التالي- في الملف `ProfileListAdd.java`، والتي حصلنا عند تشغيلها على الخرج التالي:

```
3
0
1
2
3
```

6
18
30
88
185
242
544
1325

يُمثل العمود الأول حجم المشكلة n ، أما العمود الثاني فيُمثل زمن التشغيل بوحدة الملي ثانية. كما تلاحظ، فالقياسات القليلة الأولى ليست دقيقة تمامًا وليس لها مدلول حقيقي، ولعلّه كان من الأفضل ضبط قيمة startN إلى 64000.

يُعبد التابع timingLoop النتائج بهيئة كائن من النوع XYSeries، ويُمثل متتاليةً تحتوي على القياسات. إذا مرّرتها إلى التابع plotResults، فإنه يرسم شكلًا بيانيًا -مشابها للموجود في الصورة التالية- وسنشرحه طبعًا في القسم التالي.



4.4 تفسير النتائج

بناءً على فهمنا لكيفية عمل الصف ArrayList، فإننا نتوقع أن يستغرق التابع add زمنًا ثابتًا عندما نضيف عناصر إلى نهاية القائمة، وبالتالي ينبغي أن يكون الزمن الكلي لإضافة عددٍ من العناصر مقداره n زمنًا خطيًا.

لكي نختبر صحة تلك النظرية، سنعرض تأثير زيادة حجم المشكلة على زمن التشغيل الكلي. من المفترض أن نحصل على خطّ مستقيم على الأقل لأحجام المشكلة $problem\ size$ الكبيرة بالقدر الكافي لقياس زمن التشغيل runtime بدقة. ويُمكننا كتابة دالة هذا الخط المستقيم رياضياً على النحو التالي:

$$runtime = a + b*n$$

حيث يشير a إلى إزاحة الخط وهو قيمة ثابتة و b إلى ميل الخط.

في المقابل، إذا كان التابع add خطياً، فسيكون الزمن الكلي لتنفيذ عددٍ من الإضافات بمقدار n تربيعياً. وعندها إذا نظرنا إلى تأثير زيادة حجم المشكلة على زمن التشغيل، فسننتوقع الحصول على قطعٍ مكافئٍ $parabola$ ، والذي يُمكن كتابته معادلته رياضياً على النحو التالي:

$$runtime = a + b*n + c*n^2$$

إذا كانت القياسات التي حصلنا عليها دقيقةً فسنتمكن بسهولةٍ من التمييز بين الخط المستقيم والقطع المكافئ، أمّا إذا لم تكن دقيقةً تمامًا، فقد يكون تمييز ذلك صعباً إلى حدٍّ ما، وعندئذٍ يُفَضَّلُ استخدام مقياس لوغاريتمي-لوغاريتمي $\log\text{-}\log$ لعرض تأثير حجم المشكلة على زمن التشغيل.

ولنعرف السبب، لنفترض أن زمن التشغيل يتناسب مع n^k ، ولكننا لا نعلم قيمة الأس k . يُمكننا كتابة تلك العلاقة على النحو التالي:

$$runtime = a + b*n + \dots + c*n^k$$

كما ترى في المعادلة السابقة، فإنّ الأساس ذا الأس الأكبر هو الأهم من بين القيم الكبيرة لحجم المشكلة، وبالتالي يُمكن تقريباً إهمال باقي الحدود وكتابة العلاقة على النحو التالي:

$$runtime \approx c * n^k$$

حيث نعني بالرمز \approx "يساوي تقريباً"، فإذا حسبنا اللوغاريتم لطرفي المعادلة، فستصبح مثل الآتي:

$$\log(runtime) \approx \log(c) + k*\log(n)$$

تعني المعادلة السابقة أنه لو رسمنا زمن التشغيل مقابل حجم المشكلة n باستخدام مقياس لوغاريتمي-لوغاريتمي، فسنرى خطاً مستقيماً بثابتٍ -يمثل الإزاحة- يساوي $\log(c)$ وبميل يساوي k . لا يهمنا الثابت هنا وإنما يهمنا الميل k ، فهو الذي يشير إلى ترتيب النمو. وزبدة الكلام أنه إذا كانت قيمة k تساوي 1، فالخوارزمية خطية؛ أما إذا كانت تساوي 2، فالخوارزمية تربيعية.

إذا تأملنا في الرسم البياني السابق، يُمكننا أن نُقدِّر قيمة الميل تقريبيًا، في حين لو استدعينا التابع `plotResults`، فسيحسب قيمة الميل بتطبيق طريقة المربعات الدنيا `least squares fit` على القياسات، ثم يطبعه. و قد كانت قيمة الميل التي حصل عليها التابع:

Estimated slope = **1.06194352346708**

أي تقريبًا يساوي 1. إذاً فالزمن الكلي لإجراء عدد مقداره n من الإضافات هو زمن خطي، وزمن كل إضافة منها ثابت كما توقعنا.

إذا رأيت خطأ مستقيمًا في رسمة مشابهة للرسم السابقة، فهذا لا يعني بالضرورة أن الخوارزمية خطية. إذا كان زمن التشغيل متناسبًا مع n^k لأي أس k ، فمن المتوقع أن نرى خطأ مستقيمًا ميله يساوي k . فإذا كان الميل أقرب للواحد الصحيح، فمن المرجح أن تكون الخوارزمية خطية؛ أما إذا كان أقرب لاثنين، فيُحتمل أن تكون تربيعية.

4.5 تمرين 4

ستجد ملفات الشيفرة المطلوبة لهذا التمرين في مستودع الكتاب.

1. `Profiler.java`: يحتوي على تنفيذ الصنف `Profiler` الذي شرحناه فيما سبق. ستستخدم ذلك الصنف، ولن تحتاج لفهم طريقة عمله، ومع ذلك يُمكنك الاطلاع على شيفرته إن أردت.
2. `ProfileListAdd.java`: يحتوي على شيفرة تشغيل التمرين، بما في ذلك المثال العلوي الذي شخّصنا خلاله التابع `ArrayList.add`. ستُعدّل هذا الملف لتُجري التشخيص على القليل من التوابع الأخرى.

ستجد ملف البناء `build.xml` في المجلد `code` أيضًا.

نُفذ الأمر `ant ProfileListAdd` لكي تُشغّل الملف `ProfileListAdd.java`. ينبغي أن تحصل على نتائج مشابهة للصورة المرفقة في الأعلى، ولكنك قد تضطر إلى ضبط قيمة المعاملين `startN` و `endMillis`. ينبغي أن يكون الميل المُقدَّر قريبًا من 1، مما يعني أن تنفيذ عدد n من عمليات الإضافة يستغرق زمنًا متناسبًا مع n مرفوعة للأس 1، أي ينتمي إلى المجموعة $O(n)$. ستجد تابعًا فارغًا اسمه `profileArrayListAddBeginning` في الملف `ProfileListAdd.java`. املاه بشيفرة تفحص التابع `ArrayList.add` جاعلاً إياه يُضيف العنصر الجديد دائمًا إلى بداية القائمة. إذا نسخت شيفرة التابع `profileArrayListAddEnd`، فستحتاج فقط إلى إجراء القليل من التعديلات. في الأخير، أضف سطرًا ضمن التابع `main` لاستدعاء تلك الدالة.

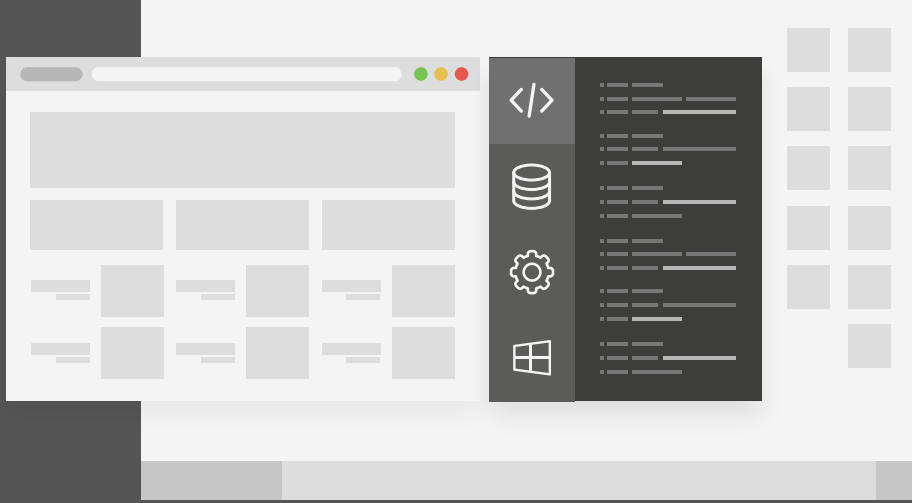
نُفذ الأمر `ant ProfileListAdd` مرةً أخرى وفُسّر النتائج. بناءً على فهمنا لطريقة عمل الصنف `ArrayList`، فإننا نتوقع أن تستغرق عملية الإضافة الواحدة زمنًا خطيًا، وبالتالي سيكون الزمن الكلي الذي

يَستغرقه عدد مقداره n من الإضافات تربيعيًا. إن كان ذلك صحيحًا، فإن الميل المُقدَّر للخط بمقياس لوغاريتمي-لوغاريتمي ينبغي أن يكون قريبًا من 2.

بعد ذلك، وازن ذلك الأداء مع أداء الصنف `LinkedList`. املأ متن التابع `profileLinkedListAddBeginning` واستخدمه لتصنيف التابع `LinkedList.add` بينما يُضيف عنصرًا جديدًا إلى بداية القائمة. ما الأداء الذي تتوقعه؟ وهل تتوافق النتائج مع تلك التوقعات؟

أخيرًا، املأ متن التابع `profileLinkedListAddEnd`، واستخدمه لتصنيف التابع `LinkedList.add` بينما يُضيف عنصرًا جديدًا إلى نهاية القائمة. ما الأداء الذي تتوقعه؟ وهل تتوافق النتائج مع تلك التوقعات؟ سنَعرض النتائج ونجيب على تلك الأسئلة في الفصل التالي.

دورة علوم الحاسوب



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حاسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



5. القائمة ازدواجية الترابط

Doubly-Linked List

سنراجع في هذا الفصل نتائج تمرين الفصل الرابع السابق، ثم سنُقدِّم تنفيذًا آخرًا للواجهة `List`، وهو القائمة ازدواجية الترابط `doubly-linked list`.

5.1 نتائج تشخيص الأداء

إستخدَمنا الصنف `Profiler.java` -في التمرين المشار إليه- لكي نُطبِّق عمليات الصنفين `ArrayList` و `LinkedList` على أحجام مختلفة من المشكلة، ثم عرضنا زمن التشغيل مقابل حجم المشكلة بمقياس لوغاريتمي-لوغاريتمي `log-log`، وقدَرنا ميل المنحني الناتج. يُوضِّح ذلك الميل العلاقة بين زمن التشغيل وحجم المشكلة.

فعلى سبيل المثال، عندما استخدمنا التابع `add` لإضافة عناصر إلى نهاية قائمة من النوع `ArrayList`، وجدنا أن الزمن الكلي لتنفيذ عدد n من الإضافات يتناسب مع n ، أي أن الميل المُقدَّر كان قريبًا من 1، وبناءً على ذلك استنتجنا أن تنفيذ عدد n من الإضافات ينتمي إلى المجموعة $O(n)$ ، وأن تنفيذ إضافة واحدة يتطلب زمنًا ثابتًا في المتوسط، أي أنه ينتمي إلى المجموعة $O(1)$ ، وهو نفس ما توصلنا إليه باستخدام تحليل الخوارزميات.

كان المطلوب من ذلك التمرين هو إكمال متن التابع `profileArrayListAddBeginning` الذي يُشخِّص عملية إضافة عناصر جديدة إلى بداية قائمة من النوع `ArrayList`. وبناءً على تحليلنا للخوارزمية، فقد توقَّعنا أن يتطلب تنفيذ إضافة واحدة زمنًا خطيًا بسبب تحريك العناصر الأخرى إلى اليمين، وعليه توقَّعنا أن يتطلب تنفيذ عدد n من الإضافات زمنًا تربيعيًا.

انظر إلى حل التمرين الذي ستجده في مجلد الحل داخل مستودع الكتاب:

```

public static void profileArrayListAddBeginning() {
    Timeable timeable = new Timeable() {
        List<String> list;

        public void setup(int n) {
            list = new ArrayList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add(0, "a string");
            }
        }
    };
    int startN = 4000;
    int endMillis = 10000;
    runProfiler("ArrayList add beginning", timeable, startN,
endMillis);
}

```

يتطابق هذا التابع تقريبًا مع التابع `profileArrayListAddEnd`، فالفارق الوحيد موجود في التابع `timeMe`، حيث يُستخدم نسخةً ثنائيةً المعامل من التابع `add` لكي يضع العناصر الجديدة في الفهرس 0، كما أنه يزيد من قيمة `endMillis` لكي يحصل على نقطة بياناتٍ إضافية.

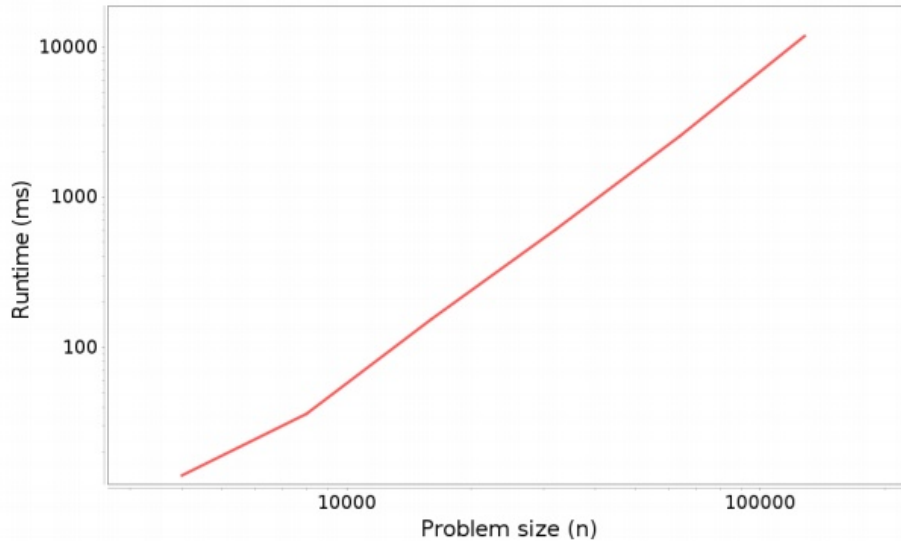
انظر إلى النتائج (حجم المشكلة على اليسار وزمن التشغيل بوحدة الملي ثانية على اليمين):

```

14
35
150
604
2518
11555

```

تُعرض الصورة التالية رسمًا بيانيًا لزمن التشغيل runtime مقابل حجم المشكلة problem size.



لا يعني ظهور خط مستقيم في هذا الرسم البياني أن الخوارزمية خطية، وإنما يعني أنه إذا كان زمن التشغيل متناسبًا مع n^k لأي أس k ، فإنه من المتوقع أن نرى خطًا مستقيمًا ميله يساوي k . نتوقع في هذا المثال أن يكون الزمن الكلي لعدد n من الإضافات متناسبًا مع n^2 ، وأن نحصل على خط مستقيم بميل يساوي 2، وفي الحقيقة يساوي الميل المُقدَّر 1.992 تقريبًا، وهو في الحقيقة دقيق جدًا لدرجة تجعلنا لا نرغب في تزوير بيانات بهذه الجودة.

5.2 تشخيص توابع الصنف LinkedList

طلب التمرين المشار إليه منك أيضًا تشخيص أداء عملية إضافة عناصر جديدة إلى بداية قائمة من النوع `LinkedList`. وبناءً على تحليلنا للخوارزمية، توقعنا أن يتطلب تنفيذ إضافة واحدة زمنًا ثابتًا؛ لأننا لا نضطر إلى تحريك العناصر الموجودة في هذا النوع من القوائم، وإنما نضيف فقط عقدة جديدة إلى بداية القائمة، وعليه توقعنا أن يتطلب تنفيذ عدد n من الإضافات زمنًا خطيًا. انظر شيفرة الحل:

```
public static void profileLinkedListAddBeginning() {
    Timeable timeable = new Timeable() {
        List<String> list;

        public void setup(int n) {
            list = new LinkedList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add(0, "a string");
            }
        }
    };
}
```

```

    }
}
};
int startN = 128000;
int endMillis = 2000;
runProfiler("LinkedList add beginning", timeable, startN,
endMillis);
}

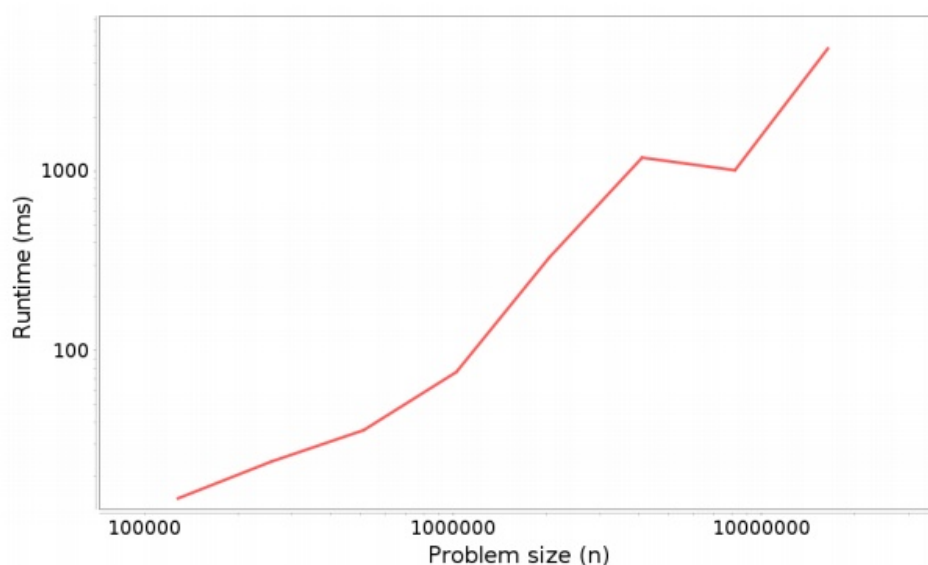
```

اضطررنا إلى إجراء القليل من التعديلات، فعدّلنا الصنف ArrayList إلى الصنف LinkedList، كما ضبطنا قيمة المعاملين startN و endMillis لكي نحصل على قياسات مناسبة، فقد لاحظنا أن القياسات ليست بدقة القياسات السابقة. انظر إلى النتائج:

```

16
19
28
77
330
892
1047
4755

```



لم نحصل على خط مستقيم تمامًا، وميل الخيط لا يساوي 1 بالضبط، وقد قَدَّرت المربعات الدنيا least squares fit الميل بحوالي 1.23، ومع ذلك تشير تلك النتائج إلى أن الزمن الكلي لعدد n من الإضافات ينتمي إلى المجموعة $O(n)$ على الأقل، وبالتالي يتطلب تنفيذ إضافة واحدة زمنًا ثابتًا.

5.3 الإضافة إلى نهاية قائمة من الصنف LinkedList

تُعدّ إضافة العناصر إلى بداية القائمة واحدةً من العمليات التي نتوقع أن يكون الصنف `LinkedList` أثناء تنفيذها أسرع من الصنف `ArrayList`؛ وفي المقابل، بالنسبة لإضافة العناصر إلى نهاية القائمة، فإننا نتوقع أن يكون الصنف `LinkedList` أبطأ، حيث يضطر تابع الإضافة إلى المرور عبر قائمة العناصر بالكامل لكي يتمكن من إضافة عنصر جديد إلى النهاية، مما يعني أن العملية خطية، وعليه نتوقع أن يكون الزمن الكلي لعدد n من الإضافات تربيعيًا.

في الواقع هذا ليس صحيحًا، ويمكنك الاطلاع إلى الشيفرة التالية:

```
public static void profileLinkedListAddEnd() {
    Timeable timeable = new Timeable() {
        List<String> list;

        public void setup(int n) {
            list = new LinkedList<String>();
        }

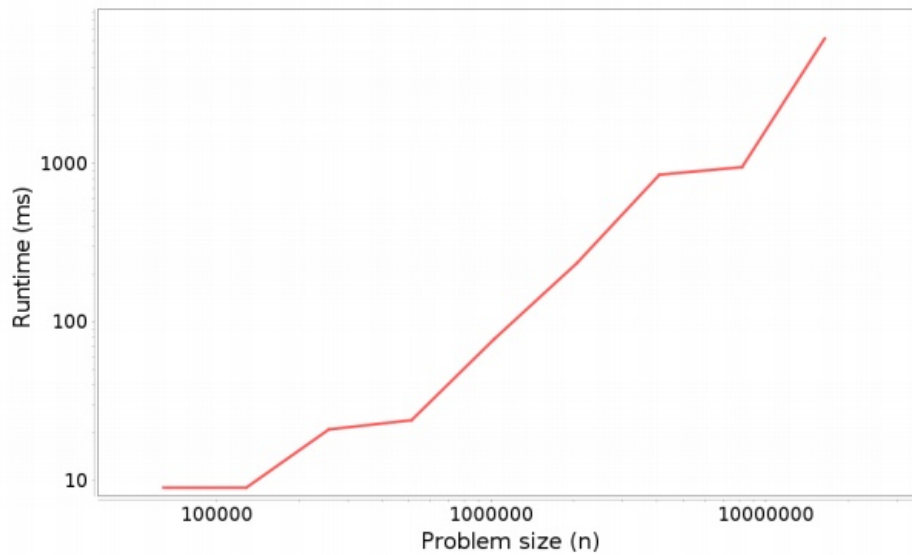
        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add("a string");
            }
        }
    };
    int startN = 64000;
    int endMillis = 1000;
    runProfiler("LinkedList add end", timeable, startN,
endMillis);
}
```

ها هي النتائج التي حصلنا عليها:

9

9

21
24
78
235
851
950
6160



كما ترى هنا فالقياسات غير دقيقة أيضًا، كما أن الخط ليس مستقيمًا تمامًا، والميل المُقدَّر يُساوي 1.19، وهو قريب لما حصلنا عليه عند إضافة العناصر إلى بداية القائمة، وليس قريبًا من 2 الذي توقعنا أن نحصل عليه بناءً على تحليلنا للخوارزمية. في الواقع، هو أقرب إلى 1، مما قد يشير إلى أن إضافة العناصر إلى نهاية القائمة يستغرق زمنًا ثابتًا.

5.4 القوائم ازدواجية الترابط Doubly-linked list

الفكرة هي أن الصنف `MyLinkedList` الذي نَقَّذناه يَستخدِم قائمة مترابطة أحادية، أي أن كل عنصرٍ يحتوي على رابطٍ واحدٍ إلى العنصر التالي، في حين يحتوي الكائن `MyArrayList` نفسه على رابطٍ إلى العقدة الأولى.

في المقابل، إذا اطلعت على توثيق الصنف `LinkedList` باللغة الإنجليزية الذي تُوقِّره جافا، فإننا نجد ما يلي:

تنفيذ قائمة ازدواجية الترابط للواجهتين `List` و `Deque`. تَعْمَل جميع العمليات بالشكل المُتَوَقَّع من قائمةٍ ازدواجية الترابط، أي تؤدي عمليات استرجاع فهرس معين إلى اجتياز أو التنقل في عناصر القائمة من البداية أو من النهاية بناءً على أيهما أقرب لذلك الفهرس.

يمكنك -إذا أردت- معرفة المزيد عن القوائم ازدواجية الترابط، وفيما يلي نذكر الفكرة العامة عنها، إذ فيها:

- تحتوي كل عقدة على رابط إلى العقدة التالية ورابط إلى العقدة السابقة.
 - تحتوي كائنات الصنف `LinkedList` على روابط إلى العنصر الأول والعنصر الأخير في القائمة.
- بناءً على ما سبق، يُمكننا أن نبدأ من أي طرف، وأن نجتاز القائمة بأي اتجاه، وعليه تتطلب إضافة العناصر وحذفها من بداية القائمة أو نهايتها زمنًا ثابتًا.

يُلخّص الجدول التالي الأداء المُتوقَّع من الصنف `ArrayList` والصنف المُخصَّص `MyLinkedList` الذي يحتوي عقده على رابط واحد والصنف `LinkedList` الذي يحتوي عقده على رابطين:

LinkedList	MyLinkedList	MyArrayList	
1	n	1	add (بالنهاية)
1	1	n	add (بالبداية)
n	n	n	add (في العموم)
n	n	1	get / set
n	n	n	indexOf / lastIndexOf
1	1	1	isEmpty / size
1	n	1	remove (من النهاية)
1	1	n	remove (من البداية)
n	n	n	remove (في العموم)

5.5 اختيار هيكل البيانات الأنسب

يُعدّ التنفيذ مزدوج الروابط أفضل من التنفيذ `ArrayList` فيما يتعلّق بعملياتي الإضافة والحذف من بداية القائمة، ويتمتعان بنفس الكفاءة فيما يتعلّق بعملياتي الإضافة والحذف من نهاية القائمة، وبالتالي تقتصر أفضلية الصنف `ArrayList` عليه بعملياتي `get` و `set`، لأنهما تتطلبان زمنًا خطيًا في القوائم المترابطة حتى لو كانت مزدوجة.

إذا كان زمن تشغيل التطبيق الخاص بك يعتمد على الزمن الذي تتطلبه عمليتا `get` و `set`، فقد يكون التنفيذ `ArrayList` هو الخيار الأفضل؛ أما إذا كان يعتمد على عملية إضافة العناصر وحذفها إلى بداية القائمة ونهايتها، فلربما التنفيذ `LinkedList` هو الخيار الأفضل.

ولكن تذكر أن هذه التوصيات مبنية على ترتيب النمو `order of growth` للأحجام الكبيرة من المشكلات. هنالك عوامل أخرى ينبغي أن تأخذها في الحسبان أيضًا:

• لو لم تكن تلك العمليات تستغرق جزءًا كبيرًا من زمن تشغيل التطبيق الخاص بك -أي لو كان التطبيق يقضي غالبية زمن تشغيله في تنفيذ أشياء أخرى-، فإن اختيارك لتنفيذ الواجهة List غير مهم لتلك الدرجة.

• إذا لم تكن القوائم التي تُعالجها كبيرةً بدرجة كافية، فلربما لن تحصل على الأداء الذي تتوقعه، فبالنسبة للمشكلات الصغيرة، قد تكون الخوارزمية التربيعية أسرع من الخوارزمية الخطية، وقد تكون الخوارزمية الخطية أسرع من الخوارزمية ذات الزمن الثابت، كما أن الاختلاف بينها في العموم لا يُهم كثيرًا.

• لا تنسى عامل المساحة. ركّزنا حتى الآن على زمن التشغيل، ولكن عامل المساحة مهم أيضًا، إذ تتطلب التنفيذات المختلفة مساحاتٍ مختلفةً من الذاكرة، وتُخزّن العناصر في قائمةٍ من الصنف ArrayList إلى جانب بعضها البعض ضمن قطعة واحدة من الذاكرة، وبالتالي لا تُبدّد مساحة الذاكرة، كما أن الحاسوب عادةً ما يكون أسرع عندما يتعامل مع أجزاء متصلة من الذاكرة. في المقابل، يتطلب كل عنصر في القوائم المترابطة عقدةً مكوّنةً من رابطٍ أو رابطتين.

تحتل تلك الروابط حيزًا من الذاكرة - أحيانًا ما يكون أكبر من الحيز الذي تحتله البيانات نفسها-، كما تكون تلك العقدة مبعثرة ضمن أجزاءٍ مختلفةٍ من الذاكرة، مما يجعل الحاسوب أقلّ كفاءةً في تعامله معها.

خلاصة القول هي أن تحليل الخوارزميات يُوفّر بعض الإرشادات التي قد تساعدك على اختيار هياكل البيانات الأنسب، ولكن بشروط:

1. زمن تشغيل التطبيق مهم.
 2. زمن تشغيل التطبيق يعتمد على اختيارك لهيكل البيانات.
 3. حجم المشكلة كبيرٌ بالقدر الكافي بحيث يتمكن ترتيب النمو من توقع هيكل البيانات الأنسب.
- في الحقيقة، يُمكنك أن تتمتع بحياةٍ مهنيّةٍ طويلةٍ أثناء عملك كمهندس برمجيات دون أن تتعرّض لهذا الموقف على الإطلاق.



أكبر موقع توظيف عن بعد في العالم العربي

ابحث عن الوظيفة التي تحقق أهدافك وطموحاتك
المهنية في أكبر موقع توظيف عن بعد

تصفح الوظائف الآن

6. التنقل في الشجرة Tree Traversal

سنتناول في هذا الفصل مقدمة سريعة عن تطبيق محرك البحث الذي ننوي بناءه، حيث سنصف مكوّناته ونشرح أُولاهها، وهي عبارة عن زاحف ويب crawler يُحمّل صفحات موقع ويكيبيديا ويحلّلها. سنتناول أيضًا تنفيذًا تعاوديًا recursive لأسلوب البحث بالعمق أولاً depth-first وكذلك تنفيذًا تكراريًا للمكدّس stack (الداخل آخرًا، يخرج أولاً LIFO) باستخدام Deque.

6.1 محركات البحث

تستقبل محركات البحث -مثل محرك جوجل وبينغ- مجموعة من كلمات البحث، وتعيد قائمةً بصفحات الإنترنت المرتبطة بتلك الكلمات (سنناقش ما تعنيه كلمة مرتبطة لاحقًا). يُمكنك قراءة المزيد عن محركات البحث (باللغة الإنجليزية)، ولكننا سنشرح هنا كل ما قد تحتاج إليه.

يتكوّن أي محرك بحث من عدة مكوناتٍ أساسيةٍ نستعرضها فيما يلي:

- الزحف crawling: برنامج بإمكانه تحميل صفحة إنترنت وتحليلها واستخراج النص وأي روابط إلى صفحات أخرى.
- الفهرسة indexing: هيكل بيانات data structure بإمكانه البحث عن كلمةٍ والعثور على الصفحات التي تحتوي على تلك الكلمة.
- الاسترجاع retrieval: طريقة لتجميع نتائج المُفهرس واختيار الصفحات الأكثر صلةً بكلمات البحث.

سنبدأ بالزاحف، والذي تتلخص مهمته في اكتشاف مجموعة من صفحات الويب وتحميلها، في حين تهدف محركات البحث مثل جوجل وبينغ إلى العثور على جميع صفحات الإنترنت، لكن المعتاد أيضًا أن يكون الزاحف مقتصرًا على نطاق أصغر. وفي حالتنا هذه، سنقتصر على صفحات موقع ويكيبيديا فقط.

في البداية، سنبنّي زاحفًا يقرأ صفحةً من موقع ويكيبيديا، ويبحث عن أول رابطٍ ضمن الصفحة، وينتقل إلى الصفحة التي يشير إليها الرابط، ثم يكرر الأمر. سنستخدم ذلك الزاحف لاختبار صحة فرضيّة فرضيّة الطريق إلى الفلسفة، الموجودة في صفحات ويكيبيديا والتي تنصّ على ما يلي:

إذا نقرت على أول رابطٍ مكتوبٍ بأحرفٍ صغيرة في أي مقالةٍ في موقع ويكيبيديا، وكوّرت ذلك على المقالات التالية، فسينتهي بك المطاف إلى مقالة الفلسفة في موقع ويكيبيديا.

سيسمح لنا اختبار تلك الفرضيّة ببناء القطع الأساسية للزاحف بدون الحاجة إلى الزحف عبر الإنترنت بأكمله أو حتى عبر كل صفحات موقع ويكيبيديا، كما أن هذا التمرين ممتعٌ نوعًا ما. أمّا المُفهرس والمُسترجع فسنبنّي كلّ منهما في فصل مستقلٍّ لاحقًا.

6.2 تحليل مستند HTML

عندما تُحمّل صفحة إنترنت، فإن محتوياتها تكون مكتوبةً بلغة ترميز النص الفائق HyperText Markup Language، التي تُختصر عادةً إلى HTML. على سبيل المثال، انظر إلى مستند HTML التالي:

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

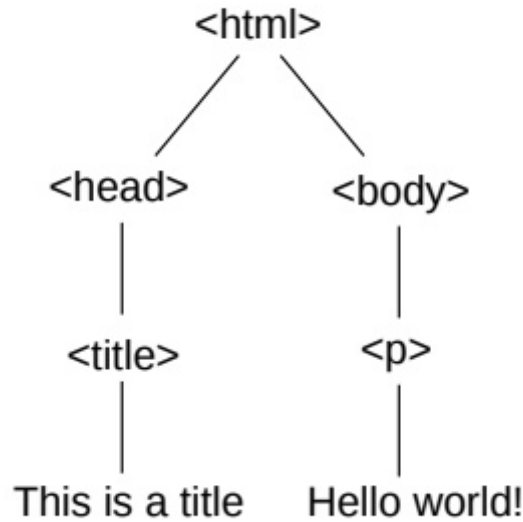
تُمثّل العبارات "This is a title" و "Hello world!" النصّ الفعليّ المعروض في الصفحة، أما بقية العناصر فهي عبارة عن وسوم tags تشير إلى الكيفية التي ستُعرض بها تلك النصوص.

بعد أن يُحمّل الزاحف صفحة إنترنت، يُحلّل محتوياتها المكتوبة بلغة HTML ليتِمكّن من استخراج النص وإيجاد الروابط. سنستخدم مكتبة jsoup مفتوحة المصدر من لغة جافا لإجراء ذلك، حيث تستطيع تلك المكتبة تحميل صفحات HTML وتحليلها.

ينتج عن تحليل مستندات HTML شجرة نموذج كائن المستند Document Object Model التي تُختصر إلى DOM، حيث تحتوي تلك الشجرة على ما يتضمنه المستند من عناصر بما في ذلك النصوص والوسوم، تمثّل هيكل بياناتٍ مترابطًا linked يتألف من عقد nodes تُمثّل كلّ من النصوص والوسوم والعناصر الأخرى.

تُحدّد بنية المستند العلاقات بين العقد. يُعدّ الوسم `<html>` -في المثال المُوضَّح في الأعلى مثلاً، العقدة الأولى التي يُطلَق عليها اسم الجذر root، وتحتوي تلك العقدة على روابط تشير إلى العقد التي تتضمنها وفي حالتنا هما العقدتان `<head>` و `<body>`، وتُعدّ كلٌّ منهما ابناً للعقدة الجذر.

تملك العقدة `<head>` ابناً واحداً هو العقدة `<title>`، وبالمثل، تملك العقدة `<body>` ابناً واحداً هو العقدة `<p>` (اختصار لكلمة paragraph). تُوضَّح الصورة التالية تلك الشجرة بيانياً.



تحتوي كلّ عقدة على روابط إلى عقد الأبناء، كما تحتوي على رابط إلى عقدة الأب الخاصة بها، وبالتالي يمكننا أن نبدأ من أي عقدة في الشجرة، ثم نتنقل إلى أعلاها أو أسفلها. عادةً ما تكون أشجار DOM المُمثلة للصفحات الحقيقية أعقد بكثير من هذا المثال.

تُوفّر غالبية متصفحات الإنترنت أدوات للتحقق من نموذج DOM الخاص بالصفحة المعروضة. ففي متصفح كروم مثلاً، يُمكنك النقر بزر الفأرة الأيمن على أي مكان من الصفحة، واختيار "Inspect" من القائمة؛ أما في متصفح فايرفوكس، فيُمكنك أيضاً النقر بزر الفأرة الأيمن على أي مكان واختيار "Inspect Element" من القائمة. يُمكنك القراءة عن أداة Web Inspector التي يُوفّرها متصفح سفاري أو كروم Chrome.



تعرض الصورة السابقة لقطة شاشة لنموذج DOM الخاص بمقالة ويكيبيديا عن لغة جافا، حيث يُمثّل العنصر المظلل أول فقرة في النص الرئيسي من المقالة. لاحظ أن الفقرة تقع داخل عنصر `<div>` الذي يملك السمة `id="mw-content-text"`، والتي سنستخدمها للعثور على النص الرئيسي في أي مقالة نُحمّلها.

6.3 استخدام مكتبة jsoup

تُسهّل مكتبة jsoup من تحميل صفحات الإنترنت وتحليلها، وكذلك التنقل عبر شجرة DOM. انظر إلى المثال التالي:

```
String url =
"http://en.wikipedia.org/wiki/Java_(programming_language)";

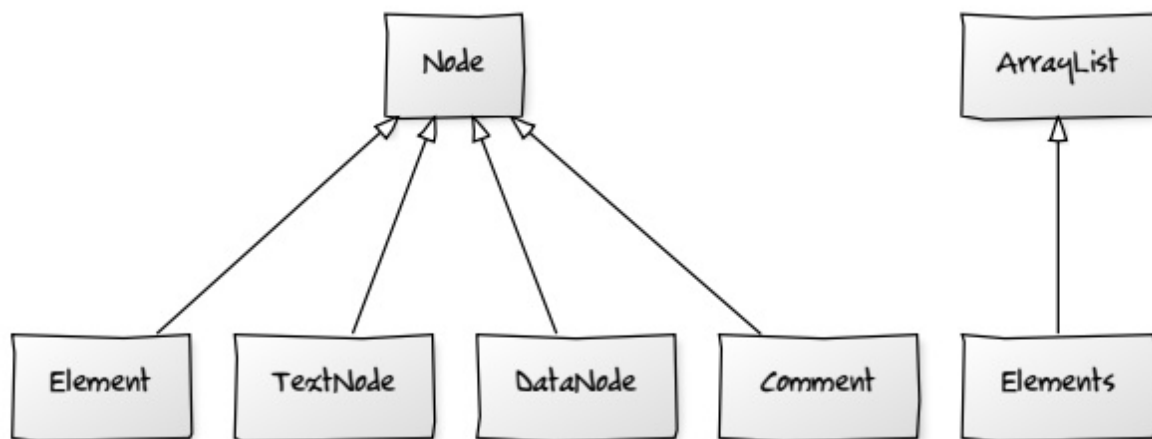
// حمّل المستند وحلّله
Connection conn = Jsoup.connect(url);
Document doc = conn.get();

// اختر المحتوى النصي واسترجع الفقرات
Element content = doc.getElementById("mw-content-text");
Elements paragraphs = content.select("p");
```

يُستقيل التابع `Jsoup.connect` مُحدّد موارد موحّدًا من النوع `String`، ويُنشئ اتصالًا مع خادم الويب. بعد ذلك يُحمّل التابع `get` مستند HTML ويُحلّله، ويعيد كائنًا من النوع `Document` يُمثّل شجرة DOM.

يُوفّر الصنف Document توابعًا للتنقل عبر الشجرة واختيار العقد. في الواقع، إنه يُوفّر توابع كثيرة جدًا لدرجة تُصيبك بالحيرة. وسيعرض المثال التالي طريقتين لاختيار العقد:

- `getElementById`: يستقبل سلسلة نصية من النوع `String`، ويبحث ضمن الشجرة عن عنصر يملك نفس قيمة حقل `id` المُمرّرة. يختار التابع في هذا المثال العقدة `<div id="mw-content-text" lang="en" dir="ltr" class="mw-content-ltr">` التي تظهر في أيّ مقالة من موقع ويكيبيديا لكي تُميّز عنصر `<div>` المُتضمّن للنص الرئيسي للصفحة، عن شريط التنقل الجانبي والعناصر الأخرى. يعيد التابع `getElementById` كائنًا من النوع `Element` يُمثل عنصر `<div>` ذاك، ويحتوي على العناصر الموجودة داخله بهيئة أبناءٍ وأحفادٍ وغيرها.
 - `select`: يستقبل سلسلة نصية من النوع `String`، ويتنقّل عبر الشجرة، ثم يُعيد جميع العناصر التي يتوافق الوسم `tag` الخاص بها مع تلك السلسلة النصية. يعيد التابع في هذا المثال جميع وسوم الفقرات الموجودة في الكائن `content`. تكون القيمة المعادة عبارة عن كائن من النوع `Elements`.
- قبل أن تُكمل القراءة، يُفضّل أن تلقي نظرةً على توثيق كلٍّ من الأصناف المذكورة لكي تتعرف على إمكانيات كلٍّ منها. تجدر الإشارة إلى أنّ الأصناف `Element` و `Elements` و `Node` هي الأصناف الأهم.
- يُمثل الصنف `Node` عقدةً في شجرة DOM. وتمتد منه أصناف فرعيةٌ كثيرةٌ مثل `Element` و `TextNode` و `Comment` و `DataNode`. يُعدّ الصنف `Elements` تجميعاً من النوع `Collection` التي تحتوي على كائناتٍ من النوع `Element`.



تحتوي الصورة السابقة على مخطط UML يوضح العلاقة بين تلك الأصناف. يشير الخط ذو الرأس الأجوف إلى أن هناك صنفًا يمتد من صنفٍ آخر، إذ يمتد الصنف `Elements` مثلاً، من الصنف `ArrayList`. وسنعود لاحقاً للحديث عن مخططات UML.

6.4 التنقل في شجرة DOM

يَسْمَح لك الصنف `WikiNodeIterable` -الذي كتبه المؤلف- بالمرور عبر عقد شجرة DOM. انظر إلى المثال التالي الذي يبين طريقة استخدامه:

```
Elements paragraphs = content.select("p");
Element firstPara = paragraphs.get(0);

Iterable<Node> iter = new WikiNodeIterable(firstPara);
for (Node node: iter) {
    if (node instanceof TextNode) {
        System.out.print(node);
    }
}
```

يُكْمِل هذا المثال ما وصلنا إليه في المثال السابق، فهو يختار الفقرة الأولى في الكائن `paragraphs` أولاً، ثم يُنشئ كائنًا من النوع `WikiNodeIterable` لِيُنْفِذَ الواجهة `Iterable<Node>`. يُجْري `WikiNodeIterable` بحثًا بتقنية العمق أولاً `depth-first`، ويُولِّد العقد بنفس ترتيب ظهورها بالصفحة. تَطْبَع الشيفرة العقد إذا كانت من النوع `TextNode` وتتجاهلها إذا كانت من أي نوع آخر، والتي تُمَثِّل وسومًا من الصنف `Element` في هذا المثال. ينتج عن ذلك طباعة نص الفقرة بدون أي ترميزات. وقد كان الخرج في هذا المثال كما يلي:

```
Java is a general-purpose computer programming language that is
concurrent, class-based, object-oriented,[13] and specifically
designed ...
```

6.5 البحث بالعمق أولاً Depth-first search

تتوفّر العديد من الطرائق للتنقل في الأشجار، ويتلاءم كلٌّ منها مع أنواعٍ مختلفةٍ من التطبيقات. سنبدأ بطريقة البحث بالعمق أولاً `DFS`. تبدأ تلك الطريقة من جذر الشجرة، ثم تختار الابن الأول للجذر. إذا كان لديه أبناء، فإنها ستختار الابن الأول، وتستمر في ذلك حتى تصل إلى عقدةٍ ليس لها أبناء، أين تبدأ بالتراجع عندها والتحرك لأعلى إلى عقدة الأب، لتختار منها الابن التالي إن كان موجودًا، وفي حالة عدم وجوده، فإنها تتراجع للوراء مجددًا. عندما تنتهي من البحث في الابن الأخير لعقدة الجذر، فإنها تكون قد انتهت.

هناك طريقتان شائعتان لتنفيذ `DFS`: إما بالتعاود `recursion`، أو بالتكرار. يُعَدُّ التنفيذ بالتعاود هو الطريقة

الأبسط:


```
private static void recursiveDFS(Node node) {
    if (node instanceof TextNode) {
        System.out.print(node);
    }
    for (Node child: node.childNodes()) {
        recursiveDFS(child);
    }
}
```

يُستدعى التابع السابق من أجل كل عقدة ضمن الشجرة بدايةً من الجذر. إذا كانت العقدة المُمرّرة من النوع `TextNode`، ويطبع التابع محتوياتها، ثم يفحص إذا كان للعقدة أي أبناء. فإذا كان لها أبناء، فإنه سيستدعي `recursiveDFS` -أي ذاته- لجميع عقد الأبناء على التوالي.

في هذا المثال، طَبَعْنَا محتويات العقد التي تنتمي إلى النوع `TextNode` قبل أن ننتقل إلى الأبناء، وهو ما يُعدّ مثالاً على التنقل ذي الترتيب السابق. يُمكنك القراءة عن التنقلات ذات الترتيب السابق `pre-order` والترتيب اللاحق `post-order` وفي الترتيب `in-order`. لا يُشكّل ترتيب التنقل في تطبيقنا هذا أي فارق.

نظرًا لأن التابع `recursiveDFS` يستدعي ذاته تعاوديًا، فقد كان بإمكانه استخدام مُكدّس الاستدعاءات للاحتفاظ بالعقد الأبناء، ومعالجتها بالترتيب المناسب، لكننا بدلًا من ذلك يُمكننا أن نستخدم مُكدّسًا صريحًا للاحتفاظ بالعقد، وفي تلك الحالة لن نحتاج إلى التعاود، حيث سَنُتِمَكِّن من التنقل في الشجرة عبر حلقة تكرارية.

6.6 المكدسات Stacks في جافا

قبل أن نشرح التنفيذ التكراري لتقنية DFS، سنناقش أولاً هيكَل بيانات يُعرَف باسم المُكدّس. سنبدأ بالفكرة العامة للمُكدّس، ثم سنتحدث عن واجهتين `interfaces` بلغة جافا تُعرِّفان توابيع المُكدّس، وهما `Stack` و `Deque`.

يُعدّ المُكدّس هيكَل بياناتٍ مشابهًا للقائمة، فهو عبارة عن مجموعة تتذكر ترتيب العناصر. ويتمثل الفرق بين المُكدّس والقائمة في أن المُكدّس يوفّر توابيع أقل، وأنه عادةً ما يوفّر المُكدّس التوابيع التالية:

- `push`: يضيف عنصرًا إلى أعلى المُكدّس.
- `pop`: يحذف العنصر الموجود أعلى المُكدّس ويعيده.
- `peek`: يعيد العنصر الموجود أعلى المُكدّس دون حذفه.
- `isEmpty`: يشير إلى ما إذا كان المُكدّس فارغًا.

نظرًا لأن التابع `pop` يسترجع العنصر الموجود في أعلى المكدس دائمًا، يُشار إلى المكدسات باستخدام كلمة "LIFO"، والتي تُعد اختصارًا لعبارة "الداخل آخرًا، يخرج أولًا". في المقابل، تُعدّ الأرتال `queue` بديلًا للمكدسات، ولكنها تُعيد العناصر بنفس ترتيب إضافتها، ولذلك، يُشار إليها عادةً باستخدام كلمة "FIFO" أي "الداخل أولًا، يخرج أولًا".

قد لا تكون أهمية المكدسات والأرتال واضحةً بالنسبة لك، فهما لا يوفران أي إمكانيات إضافية عن تلك الموجودة في القوائم `lists`. بل يوفران إمكانيات أقل، لذلك قد تتساءل لم لا نكتفي باستخدام القوائم؟ والإجابة هي أن هناك سببان:

1. إذا ألزمت نفسك بعدد أقل من التوابع، أي بواجهة تطوير تطبيقات API أصغر، فعادةً ما تصبح الشيفرة مقروءةً أكثر، كما تقل احتمالية احتوائها على أخطاء. على سبيل المثال، إذا استخدمت قائمةً لتمثيل مكدس، فقد تحذف -عن طريق الخطأ- عنصرًا بترتيب خاطئ. في المقابل، إذا استخدمت واجهة تطوير التطبيقات المخصصة للمكدس، فسيستحيل أن تقع في مثل هذا الخطأ، ولهذا فالطريقة الأفضل لتجنب الأخطاء هي بأن تجعلها مستحيلة.

2. إذا كانت واجهة تطوير التطبيقات التي يُوفرها هيكل البيانات صغيرةً، فسيكون تنفيذها بكفاءةً أسهل. على سبيل المثال، يُمكننا أن نستخدم قائمةً مترابطةً `linked list` أحادية الترابط لتنفيذ المكدس بسهولة، وعندما نضع عنصرًا في المكدس، فعلينا أن نضيفه إلى بداية القائمة؛ أما عندما نسحب عنصرًا منها، فعلينا أن نحذفه من بدايتها. ونظرًا لأن عمليتي إضافة العناصر وحذفها من بداية القوائم المترابطة تستغرق زمنًا ثابتًا، فإننا نكون قد حصلنا على تنفيذ ذي كفاءةٍ عالية. في المقابل، يصعب تنفيذ واجهات التطوير الكبيرة بكفاءة.

لديك ثلاثة خيارات لتنفيذ مكدس بلغة جافا:

1. استخدام الصنف `ArrayList` أو الصنف `LinkedList`. إذا استخدمت الصنف `ArrayList`، تأكد من إجراء عمليتي الإضافة والحذف من نهاية القائمة لأنهما بذلك سيستغرقان زمنًا ثابتًا، وانتبه من إضافة العناصر في مكان خاطئ أو حذفها بترتيب خاطئ.
2. توفّر جافا الصنف `Stack` الذي يحتوي على التوابع الأساسية للمكدسات، ولكنه يُعدّ جزءًا قديمًا من لغة جافا، فهو غير متوافق مع إطار عمل جافا للتجميعات `Java Collections Framework` الذي أُضيف لاحقًا.

3. ربما الخيار الأفضل هو استخدام إحدى تنفيذات الواجهة `Deque` مثل الصنف `ArrayDeque`.

إن كلمة `Deque` هي اختصار للتسمية رتل ذو نهايتين `double-ended queue`، والتي يُفترض أن تُلفظ `deck`، ولكنها تُلفظ أحيانًا `deek`. توفّر واجهة `Deque` بلغة جافا التوابع `push` و `pop` و `peek` و `isEmpty`، لذلك يُمكنك أن تستخدم كائنًا من النوع `Deque` كمكدس، كما أنها توفّر توابع أخرى ولكننا لن نستخدمها حاليًا.

6.7 التنفيذ التكراري لتقنية البحث بالعمق أولاً

انظر إلى التنفيذ التكراري لأسلوب "البحث بالعمق أولاً". يُستخدم ذلك التنفيذ كائنًا من النوع

ArrayDeque ليُمثل مُكدّسًا يحتوي على كائنات تنتمي إلى النوع Node:

```
private static void iterativeDFS(Node root) {
    Deque<Node> stack = new ArrayDeque<Node>();
    stack.push(root);

    while (!stack.isEmpty()) {
        Node node = stack.pop();
        if (node instanceof TextNode) {
            System.out.print(node);
        }

        List<Node> nodes = new ArrayList<Node>(node.childNodes());
        Collections.reverse(nodes);

        for (Node child: nodes) {
            stack.push(child);
        }
    }
}
```

يُمثل المعامل root جذر الشجرة التي نريد أن نجتازها، حيث سنُنشئ المُكدّس ونضيف الجذر إليه.

تستمر الحلقة loop بالعمل إلى أن يُصبح المُكدّس فارغًا. يَسحب كل تكرار ضمن الحلقة عقدةً من المُكدّس، فإذا كانت العقدة من النوع TextNode، فإنه يَطبَع محتوياتها ثم يضيف أبنائها إلى المُكدّس. ينبغي أن نضيف الأبناء إلى المُكدّس بترتيبٍ معاكسٍ لكي نتمكن من معالجتها بالترتيب الصحيح، ولذلك سننسخ الأبناء أولاً إلى قائمة من النوع ArrayList، ثم نعكس ترتيب العناصر فيها، وفي النهاية سنمرّ عبر القائمة المعكوسة.

من السهل كتابة التنفيذ التكراري لتقنية البحث بالعمق أولاً باستخدام كائن من النوع Iterator، وسترى ذلك في الفصل التالي.

في ملاحظة أخيرة عن الواجهة Deque، بالإضافة إلى الصنف ArrayDeque، تُوفّر جافا تنفيذًا آخرًا لتلك الواجهة، هو الصنف LinkedList الذي يُنفذ الواجهتين List و Deque، وتعتمد الواجهة التي تحصل عليها

على الطريقة التي تُستخدم بها. على سبيل المثال، إذا أسندت كائنًا من النوع `LinkedList` إلى متغيرٍ من النوع `Deque` كالتالي:

```
Deque<Node> deque = new LinkedList<Node>();
```

فسيكون في إمكانك استخدام التوابع المُعرَّفة بالواجهة `Deque`، لا توابع الواجهة `List`. وفي المقابل، إذا أسندته إلى متغيرٍ من النوع `List`، كالتالي:

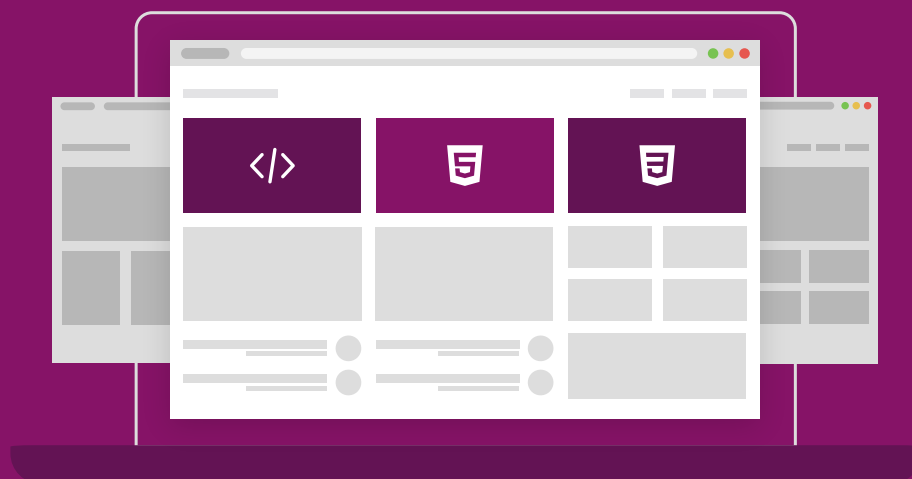
```
List<Node> deque = new LinkedList<Node>();
```

فسيكون في إمكانك استخدام التوابع المُعرَّفة بالواجهة `List` لا توابع الواجهة `Deque`؛ أما إذا أسندته على النحو التالي:

```
LinkedList<Node> deque = new LinkedList<Node>();
```

فسيكون بإمكانك استخدام جميع التوابع، ولكن الذي يحدث عند دمج توابع من واجهاتٍ مختلفة، هو أن الشيفرة ستصبح أصعب قراءةً وأكثر عرضةً لاحتواء الأخطاء.

دورة تطوير واجهات المستخدم



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حسوب
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



7. كل الطرق تؤدي إلى روما

سنبني في هذا الفصل زاحف إنترنت crawler يختبر صحة فرضية "الطريق إلى مقالة الفلسفة" Getting to Philosophy - التي تشبه المثل الشهير كل الطرق تؤدي إلى روما - في موقع ويكيبيديا التي شرحنا معناها في الفصل السابق.

7.1 البداية

ستجد في مستودع الكتاب ملفات الشيفرة التالية التي ستساعدك على بدء العمل:

1. `WikiNodeExample.java`: يحتوي على شيفرة التنفيذ التعاودي recursive والتكراري iterative لتقنية البحث بالعمق أولاً depth-first search.
2. `WikiNodeIterable.java`: يحتوي على صنف ممتد من النوع `Iterable` بإمكانه المرور عبر شجرة DOM.
3. `WikiFetcher.java`: يحتوي على صنف يُعرّف أداة تستخدم مكتبة `jsoup` لتحميل الصفحات من موقع ويكيبيديا. ويضع الصنف حدًا لسرعة تحميل الصفحات امتثالاً لشروط الخدمة في الموقع، فإذا طلبت أكثر من صفحة في الثانية الواحدة، فإنه ينتظر قليلاً قبل أن يُحمّل الصفحة التالية.
4. `WikiPhilosophy.java`: يحتوي على تصوّر مبدئي عن الشيفرة التي ينبغي أن تكملها في هذا التمرين، وسنناقشها في الأسفل.

ستجد أيضًا ملف البناء `build.xml`، حيث ستعمل الشيفرة المبدئية إذا نُفذت الأمر التالي:

```
ant WikiPhilosophy
```

7.2 الواجهتان Iterables و Iterators

تناولنا في الفصل السابق تنفيذًا تكراريًا له، وذكرنا وجه تفضيله على التنفيذ التعاودي من جهة سهولة تضمينه في كائنٍ من النوع `Iterator`. سنناقش في هذا الفصل طريقة القيام بذلك.

يُمكنك القراءة عن الواجهتين `Iterator` و `Iterable` إذا لم تكن على معرفة بهما.

ألقي نظرةً على محتويات الملف `WikiNodeIterable.java`. يُنفَّذ الصنف الخارجي `WikiNodeIterable` الواجهة `Iterable<Node>`، ولذا يُمكننا أن نستخدمه ضمن حلقة تكرار `loop` على النحو التالي:

```
Node root = ...
Iterable<Node> iter = new WikiNodeIterable(root);
for (Node node: iter) {
    visit(node);
}
```

يشير `root` إلى جذر الشجرة التي ننوي اجتيازها أو التنقل فيها، بينما يُمثل `visit` التابع الذي نرغب في تطبيقه عند مرورنا بعقدةٍ ما.

يتبع التنفيذ `WikiNodeIterable` المعادلة التقليدية:

1. يستقبل الباني `constructor` مرجعًا إلى عقدة الجذر.
2. يُنشئ التابع `iterator` كائنًا من النوع `Iterator` ويعيده.

انظر إلى شيفرة الصنف:

```
public class WikiNodeIterable implements Iterable<Node> {

    private Node root;

    public WikiNodeIterable(Node root) {
        this.root = root;
    }

    @Override
    public Iterator<Node> iterator() {
        return new WikiNodeIterator(root);
    }
}
```

```

    }
}

```

في المقابل، يُنجز الصنف الداخلي WikiNodeIterator العمل الفعلي:

```

private class WikiNodeIterator implements Iterator<Node> {

    Deque<Node> stack;

    public WikiNodeIterator(Node node) {
        stack = new ArrayDeque<Node>();
        stack.push(root);
    }

    @Override
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    @Override
    public Node next() {
        if (stack.isEmpty()) {
            throw new NoSuchElementException();
        }

        Node node = stack.pop();
        List<Node> nodes = new ArrayList<Node>(node.childNodes());
        Collections.reverse(nodes);
        for (Node child: nodes) {
            stack.push(child);
        }
        return node;
    }
}

```

تتطابق الشيفرة السابقة مع التنفيذ التكراري لأسلوب "البحث بالعمق أولاً" إلى حد كبير، ولكنها مُقسَّمة

الآن على ثلاثة توابع:

1. يُهيئ الباني المكس stack (الْمُنْفَذ باستخدام كائن من النوع ArrayDeque)، ويُضيف إليه عقدة الجذر.

2. isEmpty: يفحص ما إذا كان المكس فارغًا.

3. next: يَسَحِب العقدة التالية من المكس، ويضيف أبنائها بترتيب معاكس إلى المكس، ثم يعيد العقدة التي سحبها. وفي حال استدعاء التابع next في كائن Iterator فارغ، فإنه يُبلِّغ عن اعتراض exception.

ربما تعتقد أن إعادة كتابة تابع جيد فعليًا باستخدام صنفين، وأن خمسة توابع تُعد فكرةً غير جديرة بالاهتمام. ولكننا وقد فعلنا ذلك الآن، أصبح بإمكاننا أن نستخدم الصنف WikiNodeIterable في أي مكانٍ يُمكننا فيه استخدام النوع Iterable. يُسهِّل ذلك من الفصل بين منطق التنفيذ التكراري (البحث بالعمق أولًا) وبين المعالجة التي نريد إجراؤها على العقد.

7.3 الصنف WikiFetcher

يستطيع زاحف الويب أن يُحمِّل صفحاتٍ كثيرةً بسرعةٍ فائقةٍ، مما قد يؤدي إلى انتهاك شروط الخدمة لل خادم الذي يُحمِّل منه تلك الصفحات. ولكي نتجنَّب ذلك، وقَّرنَا الصنف WikiFetcher الذي يقوم بما يلي:

1. يُغَلِّف الشيفرة التي تناولناها في الفصل السابق، أي تلك التي تُحمِّل الصفحات من موقع ويكيبيديا، وتُحلِّل HTML، وتختار المحتوى النصي.

2. يقيس الزمن المُنقضي بين طلبات الاتصال، فإذا لم يكن كافيًا، فإنه ينتظر حتى تمرَّ فترةٌ معقولة. وقد ضبطنا تلك الفترة لتكون ثانيةً واحدةً بشكلٍ افتراضيٍّ.

انظر فيما يلي إلى تعريف الصنف WikiFetcher:

```
public class WikiFetcher {
    private long lastRequestTime = -1;
    private long minInterval = 1000;

    /**
     * حمِّل صفحة محدّد موارد موحد وحلّها
     * أعد قائمة تحتوي على عناصر تُمثِّل الفقرات
     *
     * @param url
     * @return
     * @throws IOException
```

```

    */

    public Elements fetchWikipedia(String url) throws IOException {
        sleepIfNeeded();

        Connection conn = Jsoup.connect(url);
        Document doc = conn.get();
        Element content = doc.getElementById("mw-content-text");
        Elements paragraphs = content.select("p");
        return paragraphs;
    }

    private void sleepIfNeeded() {
        if (lastRequestTime != -1) {
            long currentTime = System.currentTimeMillis();
            long nextRequestTime = lastRequestTime + minInterval;
            if (currentTime < nextRequestTime) {
                try {
                    Thread.sleep(nextRequestTime - currentTime);
                } catch (InterruptedException e) {
                    System.err.println(
                        "Warning: sleep interrupted in
fetchWikipedia.");
                }
            }
            lastRequestTime = System.currentTimeMillis();
        }
    }
}

```

يُعدّ `fetchWikipedia` هو التابع الوحيد المُعرّف باستخدام المُعدّل `public` ضمن ذلك الصنف. يُستقبل هذا التابع سلسلة نصيّة من النوع `String` وتُمثّل مُحدّد موارد موحّدًا URL، ويعيد تجميعيّة من النوع التي `Elements` تحتوي على عنصر DOM لكل فقرة ضمن المحتوى النصي. يُفترض أن تكون تلك الشيفرة مألوفة بالنسبة لك.

تقع الشيفرة الجديدة ضمن التابع `sleepIfNeeded` الذي يفحص الزمن المنقضي منذ آخر طلب، و ينتظر إذا كان الزمن أقلّ من القيمة الدنيا `minInterval` والمقدّرة بوحدة ميلي ثانية.

هذا هو كل ما يفعله الصنف `WikiFetcher`. وتوضّح الشيفرة التالية طريقة استخدامه:

```

WikiFetcher wf = new WikiFetcher();

for (String url: urlList) {
    Elements paragraphs = wf.fetchWikipedia(url);
    processParagraphs(paragraphs);
}

```

افترضنا في هذا المثال أن `urlList` عبارة عن تجميعية تحتوي على سلاسل نصية من النوع `String` وأن التابع `processParagraphs` يُعالج بطريقة ما كائن الصنف `Elements` الذي أعاده التابع `fetchWikipedia`.

يُوضّح هذا المثال شيئاً مهماً، حيث ينبغي أن تُنشئ كائناً واحداً فقط من النوع `WikiFetcher` وأن تُستخدمه لمعالجة جميع الطلبات؛ فلو كانت لديك عدة نسخ `instances` من الصنف `WikiFetcher`، فإنها لن تتمكن من فرض الزمن الأدنى اللازم بين كل طلب والطلب الذي يليه.

تفقدنا للصنف `WikiFetcher` بسيطاً للغاية، ولكن من السهل إساءة استخدامه بإنشاء عدة نسخ منه. يُمكنك أن تتجنب تلك المشكلة بجعل الصنف `WikiFetcher` يتبع نمط التصميم المفردة `singleton`.

7.4 تمرين 5

ستجد في الملف `WikiPhilosophy.java` تابع `main` بسيطاً يُوضّح طريقة استخدام أجزاء من تلك الشيفرة. وبدءاً منه، ستكون وظيفتك هي كتابة زاحف يقوم بما يلي:

1. يَسْتَقِيلُ مُحدّد موارد موحّداً URL لصفحةٍ من موقع ويكيبيديا، ويَحْمِلُها ويحلّلها.
2. يجتاز شجرة DOM الناتجة ويعثر على أول رابطٍ صالح. وسنشرح المقصود بكلمة "صالح" في الأسفل.
3. إذا لم تحتوِ الصفحة على أية روابط أو كنا قد زرنا أول رابط من قبل، فعندئذٍ ينبغي أن ينتهي البرنامج مشيراً إلى فشله.
4. إذا كان مُحدّد الموارد الموحد يشير إلى مقالة ويكيبيديا عن الفلسفة، فينبغي أن ينتهي البرنامج مشيراً إلى نجاحه.
5. وفيما عدا ذلك، يعود إلى الخطوة رقم 1.

ينبغي أن يُنشئ البرنامج قائمةً من النوع `List` تحتوي على جميع مُحدّدات الموارد التي زارها، ويَعرِض النتائج عند انتهائه، سواءً أكانت النتيجة الفشل أم النجاح.

والآن، ما الذي نعينه برابط "صالح"؟ في الحقيقة لدينا بعض الخيارات، إذ نستخدم النسخ المختلفة من نظرية "الوصول إلى مقالة ويكيبيديا عن الفلسفة" قواعد مختلفة نستعرض بعضاً منها هنا:

1. ينبغي أن يكون الرابط ضمن المحتوى النصي للصفحة وليس في شريط التنقل الجانبي أو خارج الصندوق.

2. لا ينبغي أن يكون الرابط مكتوباً بخط مائل أو بين أقواس.

3. ينبغي أن تتجاهل الروابط الخارجية والروابط التي تشير إلى الصفحة الحالية والروابط الحمراء.

4. ينبغي أن تتجاهل الرابط إذا كان بادئاً بحرف كبير.

ليس من الضروري أن تتقيد بكل تلك القواعد، ولكن يُمكنك على الأقل معالجة الأقواس والخطوط المائلة والروابط التي تشير إلى الصفحة الحالية.

إذا كنت تظن أن لديك المعلومات الكافية لتبدأ، فابدأ الآن، ولكن لا بأس قبل ذلك بقراءة التلميحات التالية:

1. ستحتاج إلى معالجة نوعين من العقد بينما تجتاز الشجرة، هما الصنفان `TextNode` و `Element`. إذا قابلت كائنًا من النوع `Element`، فلربما قد تضطر إلى تحويل نوعه `typecast` لكي تتمكن من استرجاع الوسم وغيره من المعلومات.

2. عندما تقابل كائنًا من النوع `Element` يحتوي على رابط، فعندها يُمكنك اختبار ما إذا كان مكتوباً بخط مائل باتباع روابط عقد الأب أعلى الشجرة، فإذا وجدت بينها الوسم `<i>` أو الوسم ``، فهذا يعني أن الرابط مكتوب بخط مائل.

3. لكي تفحص ما إذا كان الرابط مكتوباً بين أقواس، ستضطر إلى فحص النص أثناء التنقل في الشجرة لكي تتعقب أقواس الفتح والغلق (سيكون مثاليًا لو استطاع الحل الخاص بك معالجة الأقواس المتداخلة (مثل تلك)).

4. إذا بدأت من مقالة ويكيبيديا عن جافا، فينبغي أن تصل إلى مقالة الفلسفة بعد اتباع 7 روابط لو لم يحدث تغيير في صفحات ويكيبيديا منذ لحظة بدئنا بتشغيل الشيفرة.

الآن وقد حصلت على كل المساعدة الممكنة، يُمكنك أن تبدأ في العمل.

دورة تطوير تطبيقات الويب باستخدام لغة Ruby



دورة تدريبية متكاملة من الصفر وحتى الاحتراف
تمكنك من التخصص في هندسة الويب ودخول سوق العمل

التحق بالدورة الآن



8. المفهرس Indexer

انتهينا من بناء زاحف الإنترنت crawler في الفصل السابع السابق، وسننتقل الآن إلى الجزء التالي من تطبيق محرك البحث، وهو المفهرس. يُعدّ المفهرس -في سياق البحث عبر الإنترنت- هيكل بيانات data structure يُسهّل من العثور على الصفحات التي تحتوي على كلمة معينة، كما يساعدنا على معرفة عدد مرات ظهور الكلمة في كل صفحة، مما يُمكننا من تحديد الصفحات الأكثر صلة.

على سبيل المثال، إذا أدخل المُستخدم كلمتي البحث java و programming، فإننا نبحث عن كليهما ونسترجع عدة صفحات لكل كلمة. ستتضمّن الصفحات الناتجة عن البحث عن كلمة java مقالات عن جزيرة java، وعن الاسم المستعار للقهوة، وعن لغة البرمجة جافا. في المقابل، ستتضمّن الصفحات الناتجة عن البحث عن كلمة programming مقالات عن لغات البرمجة المختلفة، وعن استخدامات أخرى للكلمة.

باختيارنا لكلمات بحث تبحث عن الصفحات التي تحتوي على الكلمتين، سنتطلّع لاستبعاد المقالات التي ليس لها علاقة بكلمات البحث، وفي التركيز على الصفحات التي تتحدث عن البرمجة بلغة جافا.

والآن وقد فهمنا ما يعنيه المفهرس والعمليات التي يُنفذها، يُمكننا أن نُصمم هيكل بيانات يُمثّله.

8.1 اختيار هيكل البيانات

تتلخص العملية الرئيسية للمفهرس في إجراء البحث، فنحن نحتاج إلى إمكانية البحث عن كلمة معينة والعثور على جميع الصفحات التي تتضمنها. ربما يكون استخدام تجميعية من الصفحات هو الأسلوب الأبسط لتحقيق ذلك، فبتوقّر كلمة بحث معينة، يُمكننا المرور عبر محتويات الصفحات، وأن نختار من بينها تلك التي تحتوي على كلمة البحث، ولكن زمن التشغيل في تلك الطريقة سيتناسب مع عدد الكلمات الموجودة في جميع الصفحات، مما يَعني أن العملية ستكون بطيئة للغاية.

والطريقة البديلة عن تجميع الصفحات collection هي: الخريطة map، والتي هي عبارة عن هياكل بيانات يتكون من مجموعة من أزواج، حيث يتألف كل منها من مفتاح وقيمة key-value.

تُوفّر الخريطة طريقةً سريعةً للبحث عن مفتاح معين والعثور على قيمته المقابلة. سنُنشئ مثلًا الخريطة TermCounter، بحيث تربط كل كلمة بحث بعدد مرات ظهور تلك الكلمة في كل صفحة، وستُمثل المفاتيح كلمات البحث، بينما ستُمثل القيم عدد مرات الظهور (أو تكرار الظهور).

تُوفّر جافا الواجهة Map التي تُخصّص التوابع methods المُفترض توافرها في أيّ خريطة، ومن أهمها ما يلي:

- `get(key)`: يبحث هذا التابع عن مفتاح معين ويعيد قيمته المقابلة.
- `put(key, value)`: يضيف هذا التابع زوجًا جديدًا من أزواج مفتاح/قيمة إلى خريطة من النوع Map، أو يستبدل القيمة المرتبطة بالمفتاح في حالة وجوده بالفعل.
- تُوفّر جافا عدة تنفيذات للواجهة Map، ومن بينها التنفيذان HashMap و TreeMap اللذان سنناقشهما في فصول قادمة ونُحلّل أداء كُلّ منهما.

بالإضافة إلى الخريطة TermCounter التي تربط كلمات البحث بعدد مرات ظهورها، سنعرّف أيضًا الصنف Index الذي يربط كل كلمة بحث بتجميع الصفحات التي تظهر فيها الكلمة. يقودنا ذلك إلى السؤال التالي: كيف نُمثل تجميع الصفحات؟ سنتوصل إلى الإجابة المناسبة إذا فكرنا في العمليات التي ننوي تنفيذها على تلك التجميع.

سنحتاج في هذا المثال إلى دمج مجموعتين أو أكثر، وإلى العثور على الصفحات التي تظهر الكلمات فيها جميعًا. ويُمكن النظر إلى ذلك وكأنه عملية تقاطع مجموعتين sets. يتمثل تقاطع أي مجموعتين بمجموعة العناصر الموجودة في كليهما.

تُخصّص الواجهة Set بلغة جافا العمليات التي يُفترض لأي مجموعة أن تكون قادرةً على تنفيذها، ولكنها لا تُوفّر عملية تقاطع مجموعتين، وإن كانت تُوفّر توابع يُمكن باستخدامها تنفيذ عملية التقاطع وغيرها بكفاءة. وفيما يلي التوابع الأساسية للواجهة Set:

- `add(element)`: يضيف هذا التابع عنصرًا إلى مجموعة. وإذا كان العنصر موجودًا فعليًا ضمن المجموعة، فإنه لا يفعل شيئًا.
- `contains(element)`: يفحص هذا التابع ما إذا كان العنصر المُمرّر موجودًا في المجموعة.

تُوفّر جافا عدة تنفيذات للواجهة Set، ومن بينها الصنفان HashSet و TreeSet.

الآن وقد صممنا هياكل البيانات من أعلى لأسفل، فإننا سننفذها من الداخل إلى الخارج بدءًا من الصنف TermCounter.

8.2 الصنف TermCounter

يُمثّل الصنف `TermCounter` ربطًا بين كلمات البحث مع عدد مرات حدوثها في الصفحات، وتُعرض الشيفرة التالية الجزء الأول من تعريف الصنف:

```
public class TermCounter {

    private Map<String, Integer> map;
    private String label;

    public TermCounter(String label) {
        this.label = label;
        this.map = new HashMap<String, Integer>();
    }
}
```

يربط متغير النسخة `map` الكلمات بعدد مرات حدوثها، بينما يُحدّد المتغير `label` المستند الذي يحتوي على تلك الكلمات، وستُستخدمه لتخزين محددات الموارد الموحدة URLs.

يُعدّ الصنف `HashMap` أكثر تنفيذات الواجهة `Map` شيوعًا، وستُستخدمه لتنفيذ عملية الربط، كما سنتناول طريقة عمله ونفهم سبب شيوع استخدامه في الفصول القادمة.

يُوفّر الصنف `TermCounter` التابعين `put` و `get` المُعرّفين على النحو التالي:

```
public void put(String term, int count) {
    map.put(term, count);
}

public Integer get(String term) {
    Integer count = map.get(term);
    return count == null ? 0 : count;
}
```

يَعْمَل التابع `put` بمثابة تابع مُغلّف، فعندما تستدعيه، سيستدعي بدوره التابع `put` المُعرّف في الخريطة المُخزّنة داخله.

من الجهة الأخرى، يقوم التابع `get` بعمل حقيقي، فعندما تستدعيه سيستدعي التابع `get` المُعرّف في الخريطة، ثم يفحص النتيجة، فإذا لم تكن الكلمة موجودةً في الخريطة من قبل، فإن التابع `TermCounter.get` يعيد القيمة 0.

يُساعدنا تعريف التابع `get` بتلك الطريقة على تعريف التابع `incrementTermCount` بسهولة، ويُسْتَقْبَل ذلك التابع كلمةً ويزيد العدّاد الخاصّ بها بمقدار 1.

```
public void incrementTermCount(String term) {
    put(term, get(term) + 1);
}
```

إذا لم تكن الكلمة موجودةً ضمن الخريطة، فسيعيد `get` القيمة 0، ويزيد العداد بمقدار 1، ثم نستخدم التابع `put` لإضافة زوج مفتاح/قيمة `key-value` جديد إلى الخريطة. في المقابل، إذا كانت الكلمة موجودةً في الخريطة فعلاً، فإننا نسترجع قيمة العداد القديم، ونزيدها بمقدار 1، ثم نُخزّنها بحيث تُستبدل القيمة القديمة. يُعرّف الصنف `TermCounter` توابع أخرى للمساعدة على فهرسة صفحات الإنترنت:

```
public void processElements(Elements paragraphs) {
    for (Node node: paragraphs) {
        processTree(node);
    }
}

public void processTree(Node root) {
    for (Node node: new WikiNodeIterable(root)) {
        if (node instanceof TextNode) {
            processText(((TextNode) node).text());
        }
    }
}

public void processText(String text) {
    String[] array = text.replaceAll("\\pP", " ").
        toLowerCase().
        split("\\s+");

    for (int i=0; i<array.length; i++) {
        String term = array[i];
        incrementTermCount(term);
    }
}
```

- `processElements`: يستقبل هذا التابع كائنًا من النوع `Elements` الذي هو تجميعية من كائنات `Element`. يمرّ التابع عبر التجميعية ويستدعي لكل كائنٍ منها التابع `processTree`.
- `processTree`: يستقبل عقدةً تُمثل عقدة جذر شجرة DOM، ويمرّ التابع عبر الشجرة، ليعثر على العقد التي تحتوي على نص، ثم يستخرج منها النص ويمرّره إلى التابع `processText`.
- `processText`: يستقبل سلسلة نصيةً من النوع `String` تحتوي على كلمات وفراغات وعلامات ترقيم وغيرها. يحذف التابع علامات الترقيم باستبدالها بفراغات، ويحوّل الأحرف المتبقية إلى حالتها الصغرى، ثم يُقسّم النص إلى كلمات. يمرّ التابع عبر تلك الكلمات، ويستدعي التابع `incrementTermCount` لكلٍّ منها، ويستقبل التابعان `replaceAll` و `split` تعبيرات نمطية `regular expression` مثل معاملات.

وأخيرًا، انظر إلى المثال التالي الذي يوضح طريقة استخدام الصنف `TermCounter`:

```
String url =
"http://en.wikipedia.org/wiki/Java_(programming_language)";
WikiFetcher wf = new WikiFetcher();
Elements paragraphs = wf.fetchWikipedia(url);

TermCounter counter = new TermCounter(url);
counter.processElements(paragraphs);
counter.printCounts();
```

يستخدم هذا المثال كائنًا من النوع `WikiFetcher` لتحميل صفحةٍ من موقع ويكيبيديا، ثم يُحلّل النص الرئيسي الموجود بها، ويُنشئ كائنًا من النوع `TermCounter` ويستخدمه لعدّ الكلمات الموجودة في الصفحة. يُمكنك تشغيل الشيفرة في القسم التالي، واختبار فهمك لها بإكمال متن التابع غير المكتمل.

8.3 تمرين 6

ستجد ملفات شيفرة التمرين في مستودع الكتاب:

- `TermCounter.java`: يحتوي على شيفرة القسم السابق.
- `TermCounterTest.java`: يحتوي على شيفرة اختبار الملف `TermCounter.java`.
- `Index.java`: يحتوي على تعريف الصنف الخاص بالجزء التالي من التمرين.
- `WikiFetcher.java`: يحتوي على الصنف الذي استخدمناه في التمرين السابق لتحميل صفحة إنترنت وتحليلها.

- `WikiNodeIterable.java`: يحتوي على الصنف الذي استخدمناه للتنقل في عقد شجرة DOM.

ستجد أيضًا ملف البناء `build.xml`.

ننّذ الأمر `ant build` لتصريف ملفات الشيفرة، ثم ننّذ الأمر `ant TermCounter` لكي تُشغّل شيفرة القسم السابق. تطّبع تلك الشيفرة قائمة بالكلمات وعدد مرات ظهورها، وينبغي أن تحصل على خرجٍ مشابهٍ لما يلي:

```
genericServlet, 2
configurations, 1
claimed, 1
servletresponse, 2
occur, 2
Total of all counts = -1
```

قد تجد ترتيب ظهور الكلمات مختلفًا عندما تُشغّل الشيفرة، وينبغي أن يطّبع السطر الأخير المجموع الكليّ لعدد مرات ظهور جميع الكلمات، ولكنه يعيد القيمة -1 في هذا المثال لأن التابع `size` غير مكتمل. أكمل متن هذا التابع، ثم ننّذ الأمر `ant TermCounter` مرةً أخرى، حيث ينبغي أن تحصل على القيمة 4798.

ننّذ الأمر `ant TermCounterTest` لكي تتأكّد من أنك قد أكملت جزء التمرين ذاك بشكلٍ صحيح.

بالنسبة للجزء الثاني من التمرين، فسنوفّر تنفيذًا لكائني من النوع `Index`، وسيكون عليك إكمال متن التابع غير المكتمل. انظر إلى تعريف الصنف:

```
public class Index {

    private Map<String, Set<TermCounter>> index =
        new HashMap<String, Set<TermCounter>>();

    public void add(String term, TermCounter tc) {
        Set<TermCounter> set = get(term);

        // أنشئ مجموعةً جديدةً إذا كنت ترى الكلمة للمرة الأولى
        if (set == null) {
            set = new HashSet<TermCounter>();
            index.put(term, set);
        }

        // إذا كنت قد رأيت الكلمة من قبل، عدّل المجموعة الموجودة
```

```

        set.add(tc);
    }

    public Set<TermCounter> get(String term) {
        return index.get(term);
    }

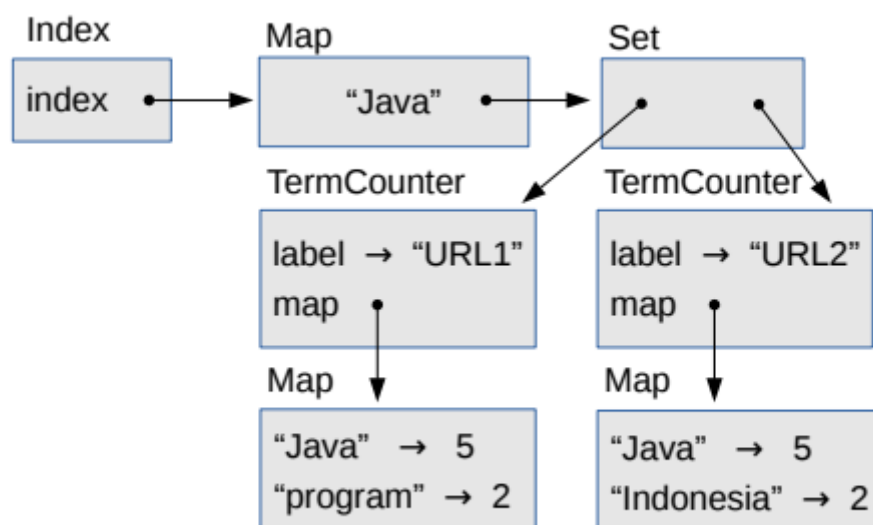
```

يُمثل متغير النسخة `index` خريطة `map` تربط كل كلمة بحثٍ بمجموعة كائناتٍ تنتمي إلى النوع `TermCounter`، ويُمثل كل كائنٍ منها صفحةً ظهرت فيها تلك الكلمة.

يضيف التابع `add` كائنًا جديدًا من النوع `TermCounter` إلى المجموعة الخاصة بكلمة معينة. وعندما نُفهرس كلمة لأول مرة، سيكون علينا أن نُنشئ لها مجموعةً جديدة، أما إذا كنا قد قابلنا الكلمة من قبل، فسنضيف فقط عنصرًا جديدًا إلى مجموعة تلك الكلمة، أي يُعدّل التابع `set.add` عندما تكون المجموعة موجودة بالفعل داخل `index` ولا يُعدّل `index` ذاته، حيث إننا سنضطر إلى تعديل `index` فقط عند إضافة كلمة جديدة.

وأخيرًا، يستقبل التابع `get` كلمة بحثٍ، ويعيد مجموعة كائنات الصنف `TermCounter` المقابلة للكلمة.

يُعَدّ هيكل البيانات هذا مُعقدًا بعض الشيء. ولاختصاره، يمكن القول أن كائن النوع `Index` يحتوي على خريطة من النوع `Map` تربط كل كلمة بحثٍ بمجموعة من النوع `Set`، المكوّنة من كائناتٍ تنتمي إلى النوع `TermCounter`، حيث يُمثل كل كائنٍ منها خريطة تربط كلمات البحث بعدد مرات ظهور تلك الكلمات.



تُعرض الصورة السابقة رسمًا توضيحيًا لتلك الكائنات، حيث يحتوي كائن الصنف `Index` على متغير نسخة اسمه `index` يشير إلى كائن الصنف `Map`، الذي يحتوي -في هذا المثال- على سلسلة نصية واحدة `Java` مرتبطة

بمجموعةٍ من النوع Set تحتوي على كائنين من النوع TermCounter؛ بحيث يكون واحدًا لكل صفحة قد ظهرت فيها كلمة Java.

يتضمّن كلّ كائنٍ من النوع TermCounter على متغيّر النسخة label الذي يُمثّل مُحدّد الموارد الموحد URL الخاص بالصفحة، كما يتضمّن المتغيّر map الذي يحتوي على الكلمات الموجودة في الصفحة، وعدد مرات حدوث كلّ كلمةٍ منها.

يُوضّح التابع printIndex طريقة قراءة هيكل البيانات ذاك:

```
public void printIndex() {
    // مرّ عبر كلمات البحث
    for (String term: keySet()) {
        System.out.println(term);

        // لكل كلمة، اطبع الصفحات التي ظهرت فيها الكلمة وعدد مرات ظهورها
        Set<TermCounter> tcs = get(term);
        for (TermCounter tc: tcs) {
            Integer count = tc.get(term);
            System.out.println("    " + tc.getLabel() + " " +
count);
        }
    }
}
```

تمرّ حلقة التكرار الخارجية عبر كلمات البحث، بينما تمرّ حلقة التكرار الداخلية عبر كائنات الصنف TermCounter.

نقّذ الأمر ant build لكي تتأكّد من تصريف ملفات الشيفرة، ثم نقّذ الأمر ant Index. سيُحمّل صفحتين من موقع ويكيبيديا ويُفهرسهما، ثم يطبع النتائج، ولكنك لن ترى أي خرج عند تشغيله لأننا تركنا أحد التوابع فارغًا.

دورك الآن هو إكمال التابع indexPage الذي يستقبل مُحدّد موارد موحّدًا URL (عبارة عن سلسلة نصية)، وكائنًا من النوع Elements، ويُحدّث المفهرس. تُوضّح التعليقات ما ينبغي أن تفعله:

```
public void indexPage(String url, Elements paragraphs) {
    // أنشئ كائنًا من النوع TermCounter وعدّ الكلمات بكل فقرة
    // لكل كلمة في كائن النوع TermCounter، أضفه إلى index
}
```

نفِّذ الأمر `Index ant` . وبعد الانتهاء، إذا كان كل شيء سليماً، فستحصل على الخرج التالي:

```
...
configurations
  http://en.wikipedia.org/wiki/Programming_language 1
  http://en.wikipedia.org/wiki/Java_(programming_language) 1
claimed
  http://en.wikipedia.org/wiki/Java_(programming_language) 1
servletresponse
  http://en.wikipedia.org/wiki/Java_(programming_language) 2
occur
  http://en.wikipedia.org/wiki/Java_(programming_language) 2
```

ضع في الحسبان أنه عند إجرائك للبحث قد يختلف ترتيب ظهور كلمات البحث.

وأخيراً، نفِّذ الأمر `TestIndex ant` لكي تتأكَّد من اكتمال هذا الجزء من التمرين على النحو المطلوب.

خُـ5سات

لبيع وشراء الخدمات المصغرة

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من \$5 فقط

تصفح الخدمات

9. الواجهة Map

سنتناول في التمارين التالية تنفيذاتٍ مختلفةً للواجهة Map، حيث يعتمدُ أحدها على الجدول hash table، والذي يُعدّ واحدًا من أفضل هياكل البيانات الموجودة، في حين يتشابه تنفيذٌ آخرُ مع الصنف TreeMap، ويُمكننا من المرور عبر العناصر بحسب ترتيبها، غير أنّه لا يتمتع بكفاءة الجداول.

ستكون لديك الفرصة لتنفيذ هياكل البيانات تلك وتحليل أدائها، وسنبدأ أولاً بتنفيذ بسيط للواجهة Map باستخدام قائمة من النوع List تتكوّن من أزواج مفاتيح/قيم key-value، ثم سننتقل إلى شرح الجداول.

9.1 تنفيذ الصنف MyLinearMap

سنُوفّر كالمعتاد شيفرةً مبدئيةً، ومهمتك إكمال التوابع غير المكتملة. انظر إلى بداية تعريف الصنف MyLinearMap:

```
public class MyLinearMap<K, V> implements Map<K, V> {  
  
    private List<Entry> entries = new ArrayList<Entry>();
```

يستخدم هذا الصنف معاملي نوع type parameters، حيث يشير المعامل الأول K إلى نوع المفاتيح، بينما يشير المعامل الثاني V إلى نوع القيم. ونظرًا لأن الصنف MyLinearMap يُنفّذ الواجهة Map، فإن عليه أن يُوفّر التوابع الموجودة في تلك الواجهة.

تحتوي كائنات النوع MyLinearMap على متغير نسخة instance variable وحيدٍ entries، وهو عبارة عن قائمة من النوع ArrayList مكوّنة من كائنات تنتمي إلى النوع Entry، حيث يحتوي كل كائن من النوع Entry على زوج مفتاح-قيمة. انظر فيما يلي إلى تعريف الصنف:


```

public class Entry implements Map.Entry<K, V> {
    private K key;
    private V value;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public K getKey() {
        return key;
    }

    @Override
    public V getValue() {
        return value;
    }
}

```

لا تتعدى كائنات الصنف `Entry` كونها أكثر من مجرد حاوٍ لزوج مفتاح/قيمة، ويقع تعريف ذلك الصنف ضمن الصنف `MyLinearList`، ويستخدم نفس معاملات النوع `K` و `V`. هذا هو كل ما ينبغي أن تعرفه لحل التمرين، ولذا سننتقل إليه الآن.

9.2 تمرين 7

ستجد ملفات شيفرة التمرين في مستودع الكتاب:

- `MyLinearMap.java`: يحتوي هذا الصنف على الشيفرة المبدئية للجزء الأول من التمرين.
- `MyLinearMapTest.java`: يحتوي على اختبارات الوحدة `unit tests` للصنف `MyLinearMap`.

ستجد أيضًا ملف البناء `build.xml` في المستودع.

ننفيذ الأمر `ant build` لكي تُصَرَّف ملفات الشيفرة، ثم ننفذ الأمر `ant MyLinearMapTest`. ستجد أن بعض الاختبارات لم تنجح؛ والسبب هو أنه ما يزال عليك القيام ببعض العمل.

أكمل متن التابع المساعد `findEntry` أولاً. لا يُعدّ هذا التابع جزءًا من الواجهة `Map`، ولكن بمجرد أن تكمله بشكلٍ صحيح، ستمكنك من استخدامه ضمن توابيع كثيرة. يبحث هذا التابع عن مفتاحٍ معينٍ ضمن المُدخَلات،

ثم يعيد إما المُدخَّل الذي يحتوي على ذلك المفتاح، أو القيمة الفارغة `null` إذا لم يكن موجودًا، كما يوازن التابع `equals` -الذي وفرناه لك- بين مفتاحين، ويعالج القيم الفارغة `null` بشكل مناسب.

نُفِّذ الأمر `MyLinearMapTest` مرةً أخرى. حتى لو كنت قد أكملت التابع `findEntry` بشكل صحيح، فلن تنجح الاختبارات لأن التابع `put` غير مكتملٍ بعد، لهذا أكمل التابع `put`. ينبغي أن تقرأ توثيق التابع `Map.put` باللغة الإنجليزية أولاً لكي تَعْرِف ما ينبغي أن تفعله. ويُمكنك البدء بكتابة نسخة بسيطةٍ من التابع `put`، تضيف دومًا مُدخَّلًا جديدًا ولا تُعَدِّل المدخلات الموجودة. سيساعدك ذلك على اختبار الحالة البسيطة من التابع، أما إذا كانت لديك الثقة الكافية، فبإمكانك كتابة التابع كاملاً من البداية.

ينبغي أن ينجح الاختبار `containsKey` بعدما تنتهي من كتابة التابع `put`. اقرأ توثيق التابع `Map.get` ثم نفِّذه، وشغِّل الاختبارات مرةً أخرى. وأخيرًا، اقرأ توثيق التابع `Map.remove`، ثم نفِّذه.

بوصولك إلى هذه النقطة، يُفترَض أن تكون جميع الاختبارات قد نجحت.

9.3 تحليل الصنف `MyLinearMap`

سنُقَدِّم حلاً للتمرين الوارد في الأعلى، ثم سنُحلِّل أداء التوابع الأساسية. انظر إلى تعريف التابعين

`findEntry` و `equals`:

```
private Entry findEntry(Object target) {
    for (Entry entry: entries) {
        if (equals(target, entry.getKey())) {
            return entry;
        }
    }
    return null;
}

private boolean equals(Object target, Object obj) {
    if (target == null) {
        return obj == null;
    }
    return target.equals(obj);
}
```

قد يعتمد زمن تشغيل التابع `equals` على حجم `target` والمفتاح، ولكنه لا يعتمد في العموم على عدد المُدخَّلات `n`، وعليه، يَستغرق التابع `equals` زمناً ثابتاً.

بالنسبة للتابع `findEntry`، ربما يحالفنا الحظ ونجد المفتاح الذي نبحث عنه في البداية، ولكن هذا ليس مضمونًا، ففي العموم، يتناسب عدد المُدخَلات التي سنبحث فيها مع `n`، وعليه، يَستغرقِ التابع `findEntry` زمنًا خطيًا.

تعتمد معظم التوابيع الأساسية المُعرَّفة في الصنف `MyLinearMap` على التابع `findEntry`، بما في ذلك التوابيع `put` و `get` و `remove`. انظر تعريف تلك التوابيع:

```
public V put(K key, V value) {
    Entry entry = findEntry(key);
    if (entry == null) {
        entries.add(new Entry(key, value));
        return null;
    } else {
        V oldValue = entry.getValue();
        entry.setValue(value);
        return oldValue;
    }
}

public V get(Object key) {
    Entry entry = findEntry(key);
    if (entry == null) {
        return null;
    }
    return entry.getValue();
}

public V remove(Object key) {
    Entry entry = findEntry(key);
    if (entry == null) {
        return null;
    } else {
        V value = entry.getValue();
        entries.remove(entry);
        return value;
    }
}
```

بعدما يستدعي التابع `put` التابع `findEntry`، فإن كل شيء آخر ضمته يستغرق زمنًا ثابتًا. لاحظ أن `entries` هي عبارة عن قائمة من النوع `ArrayList`، وأن إضافة عنصر إلى نهاية قائمة من ذلك النوع تستغرق زمنًا ثابتًا في المتوسط؛ فإذا كان المفتاح موجودًا بالفعل في الخريطة، فإننا لن نضطر إلى إضافة مُدخِل جديد، ولكننا في المقابل سنضطر لاستدعاء التابعين `entry.getValue` و `entry.setValue`، وكلاهما يستغرق زمنًا ثابتًا. بناءً على ما سبق، يُعدّ التابع `put` خطيًّا، ويستغرق التابع `get` زمنًا خطيًّا لنفس السبب.

يُعدّ التابع `remove` أعقد نوعًا ما؛ فقد يضطرّ التابع `entries.remove` إلى حذف العنصر من بداية أو وسط قائمة من النوع `ArrayList`، وهو ما يستغرق زمنًا خطيًّا. والواقع أنه ليس هناك مشكلة في ذلك، فما تزال محصلة عمليتين خطيتين عمليةً خطيَّةً أيضًا.

نستخلص مما سبق أن جميع التوابع الأساسية ضمن ذلك الصنف خطية، ولهذا السبب أطلقنا عليه اسم `MyLinearMap`.

قد يكون هذا التنفيذ مناسبًا إذا كان عدد المُدخَلات صغيرًا، ولكن ما يزال بإمكاننا أن نُحسّنه. في الحقيقة، يُمكننا أن نُنفِّذ جميع توابع الواجهة `Map`، بحيث تستغرق زمنًا ثابتًا. قد يبدو ذلك مستحيلًا عندما تسمعه لأول مرة، فهو أشبه بأن نقول أن بإمكاننا العثور على إبرٍ في كومة قشٍّ في زمنٍ ثابتٍ، وذلك بغض النظر عن حجم كومة القش.

سنشرح كيف لذلك أن يكون ممكنًا في خطوتين:

1. بدلًا من أن نُخزِّن المُدخَلات في قائمة واحدة كبيرة من النوع `List`، سنقسّمها على عدة قوائم قصيرة، وسنستخدم شيفرة تعمية `hash code` - وسنشرح معناها في الفصل التالي - لكل مفتاح؛ وذلك لتحديد القائمة التي سنستخدمها.

2. يُعدّ استخدام عدة قوائم قصيرة أسرع من استخدام قائمة واحدة كبيرة، ولكنه مع ذلك لا يُغيّر ترتيب النمو `order of growth` - كما سنناقش لاحقًا -، فما تزال العمليات الأساسية خطيَّةً، ولكن هنالك خدعة ستُمكننا من تجاوز ذلك، فإذا زدنا عدد القوائم بحيث نُقيّد عدد المُدخَلات الموجودة في كل قائمة، فسنحصل على خريطة ذات زمنٍ ثابتٍ. سنناقش تفاصيل ذلك في تمرين الفصل التالي، ولكن قبل أن نفعل ذلك سنشرح ما تعنيه التعمية `hashing`.

سنتناول حل هذا التمرين ونحلّ أداء التوابع الأساسية للواجهة `Map` في الفصل التالي، وسنقدّم أيضًا تنفيذًا أكثر كفاءة.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن



10. التعمية Hashing

سنُعرِّف في هذا الفصل الصنف `MyBetterMap` الذي يُنفَّذ الواجهة `Map` بشكلٍ أفضل من `MyLinearMap`. كما سنتناول تقنية التعمية hashing التي ساعدتنا على تنفيذ الصنف `MyBetterMap` بتلك الكفاءة.

10.1 التعمية Hashing

بهدف تحسين أداء الصنف `MyLinearMap`، سنُعرِّف صنفًا جديدًا هو `MyBetterMap` يحتوي على جميع كائنات تنتمي إلى الصنف `MyLinearMap`. يُقسَّم الصنف الجديد المفاتيح على الخرائط المُرفقة لكي يُقلَّل عدد المُدخَلات الموجودة في كل واحدةٍ منها، وبذلك يتمكَّن من زيادة سرعة التابع `findEntry` والتوابع التي تعتمد عليه.

انظر إلى تعريف الصنف:

```
public class MyBetterMap<K, V> implements Map<K, V> {  
  
    protected List<MyLinearMap<K, V>> maps;  
  
    public MyBetterMap(int k) {  
        makeMaps(k);  
    }  
  
    protected void makeMaps(int k) {  
        maps = new ArrayList<MyLinearMap<K, V>>(k);  
    }  
}
```

```

    for (int i=0; i<k; i++) {
        maps.add(new MyLinearMap<K, V>());
    }
}
}

```

يُمثل متغير النسخة maps تجميعاً كائناً تنتمي إلى الصنف MyLinearMap، حيث يستقبل الباني constructor المعامل k الذي يُحدّد عدد الخرائط المُستخدمة مبدئياً على الأقل، ثم يُنشئ التابع makeMaps تلك الخرائط ويُخزنها في قائمة من النوع ArrayList.

والآن، سنحتاج إلى طريقة تُمكننا من فحص مفتاح معين، وتقدير الخريطة التي ينبغي أن نستخدمها. وعندما نُسدي التابع put مع مفتاح جديد، سنختار إحدى الخرائط؛ أما عندما نُسدي التابع get مع نفس المفتاح، فعلينا أن نتذكر الخريطة التي وضعنا فيها المفتاح.

يُمكننا إجراء ذلك باختيار إحدى الخرائط الفرعية عشوائياً وتعقب المكان الذي وضعنا فيه كل مفتاح، ولكن كيف سنعمل ذلك؟ يُمكننا مثلاً أن نستخدم خريطة من النوع Map للبحث عن المفتاح والعثور على الخريطة الفرعية المُستخدمة، ولكن الهدف الأساسي من هذا التمرين هو كتابة تنفيذ ذي كفاءة عالية للواجهة Map، وعليه، لا يُمكننا أن نفترض وجود ذلك التنفيذ فعلياً.

بدلاً من ذلك، يُمكننا أن نستخدم دالة تعمية hash function تستقبل كائناً من النوع Object، وتعيد عدداً صحيحاً يُعرّف باسم شيفرة التعمية hash code. الأهم من ذلك هو أننا عندما نقابل نفس الكائن مرةً أخرى، فلا بُدّ أن تعيد الدالة نفس شيفرة التعمية دائماً. بتلك الطريقة، إذا استخدمنا شيفرة التعمية لتخزين مفتاح معين، فإننا سنحصل على نفس شيفرة التعمية إذا أردنا استرجاعه.

يُوفّر أيّ كائن من النوع Object بلغة جافا تابعاً اسمه hashCode، حيث يحسب هذا التابع شيفرة التعمية الخاصة بالكائن. يختلف تنفيذ هذا التابع باختلاف نوع الكائن، وسنرى مثلاً على ذلك لاحقاً.

يختار التابع المساعد التالي الخريطة الفرعية المناسبة لمفتاح معين:

```

protected MyLinearMap<K, V> chooseMap(Object key) {
    int index = 0;
    if (key != null) {
        index = Math.abs(key.hashCode()) % maps.size();
    }
    return maps.get(index);
}

```

إذا كان `key` يساوي `null`، فإننا سنختار الخريطة الفرعية الموجودة في الفهرس 0 عشوائيًا. وفيما عدا ذلك، سنستخدم التابع `hashCode` لكي نحصل على عدد صحيح، ثم نطبق عليه التابع `Math.abs` لكي نتأكد من أنه لا يحتوي على قيمة سالبة، ثم نستخدم عامل باقي القسمة `%` لكي نحصل على قيمة واقعة بين 0 و `maps.size() - 1`، وبذلك نضمن أن يكون `index` فهرسًا صالحًا للاستخدام مع التجميعية `maps` دائمًا. وفي الأخير، سيعيد `chooseMap` مرجعًا `reference` إلى الخريطة المختارة.

لاحظ أننا استدعينا `chooseMap` بالتابعين `put` و `get`، وبالتالي عندما نبحث عن مفتاح معين، يُفترض أن نحصل على نفس الخريطة التي حصلنا عليها عندما أضفنا ذلك المفتاح. نقول هنا أنه يُفترض وليس حتمًا، لأنه من المحتمل ألا يحدث، وهو ما سنشرح أسبابه لاحقًا.

انظر إلى تعريف التابعين `put` و `get`:

```
public V put(K key, V value) {
    MyLinearMap<K, V> map = chooseMap(key);
    return map.put(key, value);
}

public V get(Object key) {
    MyLinearMap<K, V> map = chooseMap(key);
    return map.get(key);
}
```

ربما لاحظت أن الشيفرة بسيطة للغاية، حيث يستدعي التابعان التابع `chooseMap` للعثور على الخريطة الفرعية الصحيحة، ثم يستدعيان تابعًا في تلك الخريطة الفرعية، وهذا كل ما في الأمر. والآن، لنفحص أداء التابعين.

إذا كان لدينا عدد مقداره `n` من المُدخَلات مُقسَّمًا على عدد مقداره `k` من الخرائط الفرعية، فسيصبح لدينا في المتوسط عدد `n/k` من المُدخَلات في كل خريطة. وعندما نبحث عن مفتاح معين، سنضطر إلى حساب شيفرة تعميته، والتي تستغرق بعض الوقت، ثم سنبحث في الخريطة الفرعية المقابلة.

لما كان حجم قوائم المُدخَلات في الصنف `MyBetterMap` أقل بمقدار `k` مرة من حجمها في الصنف `MyLinearMap`، فمن المُتوقع أن يكون البحث أسرع بمقدار `k` مرة، ومع ذلك، ما يزال زمن التشغيل متناسبًا مع `n`، وبالتالي ما يزال الصنف `MyBetterMap` خطيًّا. سنعالج تلك المشكلة في التمرين التالي.

10.2 كيف تعمل التعمية؟

إذا طَبَّقنا دالة تعمية على نفس الكائن، فلا بُدَّ لها أن تنتج نفس شيفرة التعمية في كل مرّة، وهو أمرٌ سهلٌ نوعًا ما إذا كان الكائن غير قابلٍ للتعديل `immutable`؛ أما إذا كان قابلاً للتعديل، فالأمر يحتاج إلى بعض التفكير. كمثال على الكائنات غير القابلة للتعديل، سنعرّف الصنف `SillyString`، حيث يُغلف ذلك الصنف سلسلة نصيّة من النوع `String`:

```
public class SillyString {
    private final String innerString;

    public SillyString(String innerString) {
        this.innerString = innerString;
    }

    public String toString() {
        return innerString;
    }
}
```

في الواقع، هذا الصنف ليس ذا فائدةٍ كبيرةٍ، ولهذا السبب سميناه `SillyString`، ولكنه مع ذلك يُوضّح كيف يُمكن لصنفٍ أن يُعرّف دالة التعمية الخاصّة به:

```
@Override
public boolean equals(Object other) {
    return this.toString().equals(other.toString());
}

@Override
public int hashCode() {
    int total = 0;
    for (int i=0; i<innerString.length(); i++) {
        total += innerString.charAt(i);
    }
    return total;
}
```

أعاد الصنف `SillyString` تعريفَ التابعين `equals` و `hashCode`، وهذا الأمر ضروري لأننا لو أردنا له أن يعمل بشكل مناسب، فلا بُدَّ أن يكون التابع `equals` متوافقًا مع التابع `hashCode`. يعني هذا أنه لو كان

لدينا كائنان متساويين -يُعيد التابع `equals` القيمة `true` عند تطبيقه عليهما-، فلا بُدَّ أن تكون لهما نفس شيفرة التعمية، ولكن هذا صحيحٌ من اتجاهٍ واحدٍ فقط، فمن المحتمل أن يملك كائنان نفس شيفرة التعمية، ومع ذلك لا يكونان متساويين.

يُستدعي `equals` التابع `toString` الذي يعيد قيمةً متغير النسخة `innerString`، ولذلك يتساوى كائن من النوع `SillyString` إذا تساوى متغير النسخة `innerString` المُعرَّف فيهما.

يمرّ التابع `hashCode` عبر محارف السلسلة النصية -من النوع `String`- ويحسب حاصل مجموعها. وعندما نضيف محرّفًا إلى عددٍ صحيحٍ من النوع `int`، سُنحوّل جافا المحرف إلى عددٍ صحيحٍ باستخدام رقم محرف يونيكود `Unicode code point` الخاص به. يُمكنك قراءة المزيد عن أرقام محارف اليونيكود (باللغة الإنجليزية) إذا أردت، ولكنه غير ضروري لفهم هذا المثال.

تُحقّق دالّة التعمية السابقة الشرط التالي:

إذا احتوى كائن من النوع `SillyString` على سلاسل نصيّة متساوية، فإنهما يحصلان على نفس شيفرة التعمية.

تعمل الشيفرة السابقة بشكل صحيح، ولكنها ليست بالكفاءة المطلوبة؛ فهي تعيد شيفرة التعمية نفسها لعدد كبير من السلاسل النصية المختلفة؛ فمثلاً لو تكوّنت سلسلتان من نفس الأحرف مهما كان ترتيبها، فإنهما ستحصلان على نفس شيفرة التعمية، بل حتى لو لم تتكونا من نفس الأحرف، فقد يكون حاصل المجموع متساوياً مثل `ac` و `bb`.

إذا حصلت كائنات كثيرة على نفس شيفرة التعمية، فإنها سُنحزّن في نفس الخريطة الفرعية، وإذا احتوت خرائط فرعيّة معيّنة على مُدخلاتٍ أكثر من غيرها، فإن السرعة التي نُحقّقها باستخدام عدد مقداره `k` من الخرائط تكون أقلّ بكثيرٍ من `k`، ولذلك ينبغي أن تكون دوال التعمية منتظمةً، أي لا بُدَّ أن تكون احتمالية الحصول على أي قيمة ضمن النطاق المسموح به متساوية. يُمكنك قراءة المزيد عن التصميم الجيد لدوال التعمية لو أردت.

10.3 التعمية والقابلية للتغيير mutation

يُعدّ الصنف `String` غير قابلٍ للتعديل، وكذلك الصنف `SillyString`؛ وذلك لأننا صرحنا عنه باستخدام `final`. بمجرد أن تُنشئ كائنًا من النوع `SillyString`، فإنك لا تستطيع أن تُعدّل متغير النسخة `innerString` المُعرّف فيه لتجعله يشير إلى سلسلة نصيّة مختلفة من النوع `String`، كما أنك لا تستطيع أن تُعدّل السلسلة النصيّة التي يشير إليها، وبالتالي ستكون للكائن نفس شيفرة التعمية دائماً.

ولكن، ماذا يحدث لو كان الكائن قابلاً للتعديل؟ انظر إلى تعريف الصنف `SillyArray` المطابق للصنف `SillyString` باستثناء أنه يستخدم مصفوفة محارف بدلاً من الصنف `String`:

```

public class SillyArray {
    private final char[] array;

    public SillyArray(char[] array) {
        this.array = array;
    }

    public String toString() {
        return Arrays.toString(array);
    }

    @Override
    public boolean equals(Object other) {
        return this.toString().equals(other.toString());
    }

    @Override
    public int hashCode() {
        int total = 0;
        for (int i=0; i<array.length; i++) {
            total += array[i];
        }
        System.out.println(total);
        return total;
    }
}

```

يُوفّر الصنف SillyArray التابع setChar الذي يَسْمَح بتعديل المحارف الموجودة في المصفوفة:

```

public void setChar(int i, char c) {
    this.array[i] = c;
}

```

والآن، لنفترض أننا أنشأنا كائنًا من النوع SillyArray، ثم أضفناه إلى خريطةٍ كالتالي:

```

SillyArray array1 = new SillyArray("Word1".toCharArray());
map.put(array1, 1);

```

شيفرة التعمية لتلك المصفوفة هي 461. والآن إذا عدّنا محتويات المصفوفة، وحاولنا أن نسترجعها كالتالي:

```
array1.setChar(0, 'C');
Integer value = map.get(array1);
```

ستكون شيفرة التعمية بعد التعديل قد أصبحت 441. بحصولنا على شيفرة تعمية مختلفة، فإننا غالبًا سنبحث في الخريطة الفرعية الخاطئة، وبالتالي لن نعثر على المفتاح على الرغم من أنه موجود في الخريطة، وهذا أمر سيء.

لا يُعد استخدام الكائنات القابلة للتعديل مفاتيحًا لهياكل البيانات المبنية على التعمية -مثل `MyBetterMap` و `HashMap`- حلًا آمنًا، فإذا كنت متأكدًا من أن قيم المفاتيح لن تتعدّل بينما هي مُستخدمة في الخريطة، أو أن أي تعديل يُجرى عليها لن يؤثر على شيفرة التعمية؛ فلربما يكون استخدامها مناسبًا، ولكن من الأفضل دائمًا أن تتجنّب ذلك.

10.4 تمرين 8

سنُنهى في هذا التمرين تنفيذ الصنف `MyBetterMap`، حيث ستجد ملفات شيفرة التمرين في مستودع الكتاب:

- `MyLinearMap.java`: يحتوي على حل تمرين الفصل السابق الواجهة `Map` الذي سنبني عليه هذا التمرين.
- `MyBetterMap.java`: يحتوي على شيفرة من نفس الفصل مع إضافة بعض التوابع التي يُفترض أن تُكملها.
- `MyHashMap.java`: يحتوي على تصوّر مبدئيّ -عليك إكماله- لجدول ينمو عند الضرورة.
- `MyLinearMapTest.java`: يحتوي على اختبارات وحدة للصنف `MyLinearMap`.
- `MyBetterMapTest.java`: يحتوي على اختبارات وحدة للصنف `MyBetterMap`.
- `MyHashMapTest.java`: يحتوي على اختبارات وحدة للصنف `MyHashMap`.
- `Profiler.java`: يحتوي على شيفرة لقياس الأداء ورسم تأثير حجم المشكلة على زمن التشغيل.
- `ProfileMapPut.java`: يحتوي على شيفرة تقيس أداء التابع `Map.put`.

كالعادة، عليك أن تُنفذ الأمر `ant build` لكي تُصرّف ملفات الشيفرة، ثم الأمر `ant MyBetterMapTest`. ستفشل العديد من الاختبارات؛ وذلك لأنه ما يزال عليك إكمال بعض التوابع.

راجع تنفيذ التابعين `put` و `get` من ذات الفصول المشار إليها في الأعلى، ثم أكمل متن التابع `containsKey`. سيكون عليك استخدام التابع `chooseMap`، وبعدها تنتهي نَقْذ الأمر `MyBetterMapTest` مرةً أخرى، وتأكد من نجاح `testContainsKey`.

أكمل متن التابع `containsValue`، ولا تستخدم لذلك التابع `chooseMap`. نَقْذ الأمر `MyBetterMapTest` مرةً أخرى، وتأكد من نجاح `testContainsValue`. لاحظ أن العثور على القيمة يتطلب عملاً أكبر من العثور على المفتاح.

يُعد تنفيذ التابع `containsKey` تنفيذاً خاطئاً مثل التابعين `put` و `get`؛ لأنه عليه أن يبحث في إحدى الخرائط الفرعية. سنشرح في الفصل التالي كيف يُمكننا تحسين هذا التنفيذ أكثر.

دورة إدارة تطوير المنتجات



تعلم تحويل أفكارك لمنتجات ومشاريع حقيقية بدءًا من دراسة السوق وتحليل المنافسين وحتى إطلاق منتج مميز وناجح

التحق بالدورة الآن



11. الواجهة HashMap

كتبنا تنفيذًا للواجهة Map باستخدام التعمية hashing في الفصل السابق، وتوقعنا أن يكون ذلك التنفيذ أسرع لأن القوائم التي يبحث فيها أقصر، ولكن ما يزال ترتيب نمو order of growth ذلك التنفيذ خطيًّا.

إذا كان هناك عدد مقداره n من المُدخَلات وعدد مقداره k من الخرائط الفرعية sub-maps، فإن حجم تلك الخرائط يُساوي n/k في المتوسط، أي ما يزال متناسبًا مع n ، ولكننا لو زدنا k مع n ، سَنتمكّن من الحدّ من حجم n/k . لنفترض على سبيل المثال أننا سنضاعف قيمة k في كلّ مرّة تتجاوز فيها n قيمة k . في تلك الحالة، سيكون عدد المُدخَلات في كلّ خريطة أقلّ من 1 في المتوسط، وأقلّ من 10 على الأغلب بشرط أن تُوزّع دالّة التعمية المفاتيح بشكلٍ معقول.

إذا كان عدد المُدخَلات في كلّ خريطة فرعيّة ثابتًا، سَنتمكّن من البحث فيها بزمنٍ ثابت. علاوة على ذلك، يَستغرق حساب دالة التعمية في العموم زمنًا ثابتًا (قد يعتمد على حجم المفتاح، ولكنه لا يعتمد على عدد المفاتيح). بناءً على ما سبق، سَتستغرق توابع Map الأساسية أي put و get زمنًا ثابتًا.

سنفحص تفاصيل ذلك في التمرين التالي.

11.1 تمرين 9

وقرنا التصور المبدئي لجدول تعمية hash table ينمو عند الضرورة في الملف MyHashMap.java. انظر إلى بداية تعريفه:

```
public class MyHashMap<K, V> extends MyBetterMap<K, V> implements
    Map<K, V> {
```

// متوسط عدد المُدخَلات المسموح بها في كل خريطة فرعية قبل إعادة حساب شيفرات التعمية

```

private static final double FACTOR = 1.0;

@Override
public V put(K key, V value) {
    V oldValue = super.put(key, value);

    // تأكد مما إذا كان عدد العناصر في الخريطة الفرعية قد تجاوز الحد الأقصى
    if (size() > maps.size() * FACTOR) {
        rehash();
    }
    return oldValue;
}
}

```

يمتدّ الصنف MyHashMap من الصنف MyBetterMap، وبالتالي، فإنه يرث التوابع المُعرّفة فيه. يعيد الصنف MyHashMap تعريف التابع put، فيستدعي أولاً التابع put في الصنف الأعلى superclass أي يستدعي النسخة المُعرّفة في الصنف MyBetterMap، ثم يفحص ما إذا كان عليه أن يُعيد حساب شيفرة التعمية. يعيد التابع size عدد المُدخّلات الكلية n بينما يعيد التابع maps.size عدد الخرائط k.

يُحدّد الثابت FACTOR -الذي يُطلق عليه اسم عامل الحمولة load factor- العدد الأقصى للمُدخّلات وسيطاً في كل خريطة فرعية. إذا تحقّق الشرط $n > k * FACTOR$ ، فهذا يعني أن الشرط $n/k > FACTOR$ مُتحقّق أيضاً، مما يعني أن عدد المُدخّلات في كل خريطة فرعية قد تجاوز الحد الأقصى، ولذلك، يكون علينا استدعاء التابع rehash.

ننّفذ الأمر `ant build` لتصريف ملفات الشيفرة، ثم ننّفذ الأمر `MyHashMapTest ant`. ستفشل الاختبارات لأن تنفيذ التابع rehash يُبلّغ عن اعتراض exception، ودورك هو أن تكمل متن هذا التابع.

إذا أكمل متن التابع rehash بحيث يُجمّع المُدخّلات الموجودة في الجدول. بعد ذلك، عليه أن يضبط حجم الجدول، ويضيف المُدخّلات إليه مرةً أخرى. وقّرنا تابعين مساعدين هما `MyBetterMap.makeMaps` و `MyLinearMap.getEntries`. ينبغي أن يُضاعف حلك عدد الخرائط k في كل مرة يُستدعى فيها التابع.

11.2 تحليل الصنف MyHashMap

إذا كان عدد المُدخّلات في أكبر خريطة فرعية متناسباً مع n/k ، وكانت الزيادة بقيمة k متناسبةً مع n، فإن العديد من التوابع الأساسية في الصنف MyBetterMap تُصبح ثابتة الزمن:


```

public boolean containsKey(Object target) {
    MyLinearMap<K, V> map = chooseMap(target);
    return map.containsKey(target);
}

public V get(Object key) {
    MyLinearMap<K, V> map = chooseMap(key);
    return map.get(key);
}

public V remove(Object key) {
    MyLinearMap<K, V> map = chooseMap(key);
    return map.remove(key);
}

```

يَحْسِب كلّ تابع شيفرة التعمية للمفتاح، وهو ما يَسْتَغْرِقُ زَمَنًا ثَابِتًا، ثم يَسْتَدْعِي تَابِعًا على خريطةٍ فرعيةٍ، وهو ما يَسْتَغْرِقُ أيضًا زَمَنًا ثَابِتًا.

ربما الأمورُ جيدةٌ حتى الآن، ولكن ما يزال من الصعب تحليل أداء التابع الأساسي الآخر put، فهو يَسْتَغْرِقُ زَمَنًا ثَابِتًا إذا لم يضطرّ لاستدعاء التابع rehash، ويَسْتَغْرِقُ زَمَنًا خَطِيئًا إذا اضطرّ لذلك. بتلك الطريقة، يكون هذا التابع مشابهًا للتابع ArrayList.add الذي حللناه أدناه في الفصل الثالث قائمة المصفوفة ArrayList.

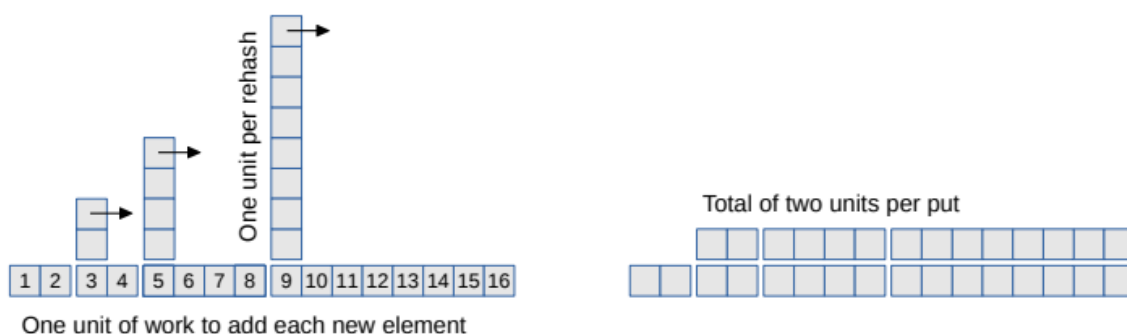
ولنفس السبب، يتّضح أن التابع MyHashMap.put يَسْتَغْرِقُ زَمَنًا ثَابِتًا إذا حسبنا متوسط زمنٍ متتاليةٍ من الاستدعاءات. يعتمد هذا التفسير على التحليل بالتسديد amortized analysis الذي شرحناه في نفس الفصل.

لنفترض أن العدد المبدئيّ للخرائط الفرعية k يساوي 2، وأن عامل التحميل يساوي 1، والآن، لنفحص الزمن الذي يَسْتَغْرِقُهُ التابع put لإضافة متتاليةٍ من المفاتيح. سنُعَدُّ عدد المرات التي سنضطرّ خلالها لحساب شيفرة التعمية لمفتاحٍ وإضافته لخريطةٍ فرعيةٍ، وسيكون ذلك بمنزلةٍ وحدةٍ عملٍ واحدة.

سَيُنْفَذُ التابع put عند استدعائه لأوّل مرةٍ وحدةً عملٍ واحدةً من وحدات العمل، وسَيُنْفَذُ أيضًا عند استدعائه في المرة الثانية وحدةً عملٍ واحدةً. أمّا في المرة الثالثة، فسيضطرّ لإعادة حساب شيفرات التعمية، وبالتالي، سَيُنْفَذُ عدد 2 من وحدات العمل لكي يَحْسِب شيفرات تعمية المفاتيح الموجودة بالفعل بالإضافة إلى وحدة عملٍ أخرى لحساب شيفرة تعمية المفتاح الجديد.

والآن، أصبح حجم الجدول 4، وبالتالي، سيُنقذ التابع put عند استدعائه في المرة التالية وحدة عملٍ واحدة، ولكن، في المرة التالية التي سيضطرّ خلالها لاستدعاء rehash، فإنه سيُنقذ 4 وحدات عملٍ لحساب شيفرات تعمية المفاتيح الموجودة ووحدة عملٍ إضافية للمفتاح الجديد.

تُوضّح الصورة التالية هذا النمط، حيث يظهر العمل اللازم لحساب شيفرة تعمية مفتاحٍ جديدٍ في الأسفل بينما يَظهر العمل الإضافي كبرج.



إذا أنزلنا الأبراج كما تقترح الأسهم، سيملاً كلّ واحدٍ منها الفراغ الموجود قبل البرج التالي، وسنحصل على ارتفاعٍ منتظمٍ يساوي 2 وحدة عمل. يُوضّح ذلك أن متوسط العمل لكل استدعاءٍ للتابع put هو 2 وحدة عمل، مما يعني أنه يستغرق زمنًا ثابتًا في المتوسط.

يُوضّح الرسم البياني مدى أهمية مضاعفة عدد الخرائط الفرعية k عندما نعيد حساب شيفرات التعمية؛ فلو أضفنا قيمةً ثابتةً إلى k بدلاً من مضاعفتها، ستكون الأبراج قريبة جداً من بعضها، وستتراكم فوق بعضها، وعندها، لن نحصل على زمن ثابت.

11.3 مقايضات ما بين الزمن والأداء

رأينا أن التوابع containsKey و get و remove تستغرق زمنًا ثابتًا، وأن التابع put يستغرق زمنًا ثابتًا في المتوسط، وهذا أمرٌ رائعٌ بحق، فإداء تلك العمليات هو نفسه تقريبًا بغض النظر عن حجم الجدول.

يَعتمد تحليلنا لأداء تلك العمليات على نموذج معالجةٍ بسيطٍ تستغرق كلّ وحدة عمل فيه نفس مقدار الزمن، ولكن الحواسيب الحقيقية أعقدُ من ذلك بكثير، فتبلغ أقصى سرعتها عندما تتعامل مع هياكل بيانات صغيرة بما يكفي لتُوضع في الذاكرة المخبئية cache، وتكون أبطأ قليلاً عندما لا يتناسب حجم هياكل البيانات مع الذاكرة المخبئية ولكن مع إمكانية وضعها في الذاكرة، وتكون أبطأ بكثيرٍ إذا لم يتناسب حجم الهياكل حتى مع الذاكرة.

هنالك مشكلة أخرى، وهي أن التعمية لا تكون ذات فائدة في هذا التنفيذ إذا كان المُدخَل قيمةً وليس مفتاحًا، فالتابع `containsValue` خطيٌّ لأنه مضطّر للبحث في كل الخرائط الفرعية، فليس هناك طريقة فعالة للبحث عن قيمة ما والعثور على مفتاحها المقابل (أو مفاتيحها).

بالإضافة إلى ما سبق، فإن بعض التوابع التي كانت تستغرق زمنًا ثابتًا في الصنف `MyLinearMap` قد أصبحت خطيَّة. انظر إلى التابع التالي على سبيل المثال:

```
public void clear() {
    for (int i=0; i<maps.size(); i++) {
        maps.get(i).clear();
    }
}
```

يضطرّ التابع `clear` لتفريغ جميع الخرائط الفرعية التي يتناسب عددها مع `n`، وبالتالي، هذا التابع خطيٌّ. لحسن الحظ، لا يُستخدم هذا التابع كثيرًا، ولذا فما يزال هذا التنفيذ مقبولا في غالبية التطبيقات.

11.4 تشخيص الصنف MyHashMap

سنفحص أولاً ما إذا كان التابع `MyHashMap.put` يستغرق زمنًا خطيًّا.

نقِّد الأمر `ant build` لتصريف ملفات الشيفرة، ثم نقِّد الأمر `ant ProfileMapPut`. يقيس الأمر زمن تشغيل التابع `HashMap.put` (الذي توقَّره جافا) مع أحجام مختلفة للمشكلة، ويعرِّض زمن التشغيل مع حجم المشكلة بمقياس لوغاريتمي-لوغاريتمي. إذا كانت العملية تستغرق زمنًا ثابتًا، ينبغي أن يكون الزمن الكلي لعدد `n` من العمليات خطيًّا، ونحصل عندها على خطّ مستقيم ميله يساوي 1. عندما شغلنا تلك الشيفرة، كان الميل المُقدَّر قريبًا من 1، وهو ما يتوافق مع تحليلنا للتابع. ينبغي أن تحصل على نتيجة مشابهة.

عدّل الصنف `ProfileMapPut.java` لكي يُشخِّص التنفيذ `MyHashMap` الخاص بك وليس تنفيذ الجافا `HashMap`. شغل شيفرة التشخيص مرةً أخرى، وافحص ما إذا كان الميل قريبًا من 1. قد تضطرّ إلى تعديل قيم `startN` و `endMillis` لكي تعثر على نطاقٍ مناسبٍ من أحجام المشكلة يبلغ زمن تشغيلها أجزاءً صغيرةً من الثانية، وفي نفس الوقت لا يتعدى بضعة آلاف.

عندما شغلنا تلك الشيفرة، وجدنا أن الميل يساوي 1.7 تقريبًا، مما يشير إلى أن ذلك التنفيذ لا يستغرق زمنًا ثابتًا. في الحقيقة، هو يحتوي على خطأ برمجيّ متعلّق بالأداء.

عليك أن تعثر على ذلك الخطأ وتصلحه وتتأكد من أن التابع `put` يستغرق زمنًا ثابتًا كما كنا نتوقَّع قبل أن تنتقل إلى القسم التالي.

11.5 إصلاح الصنف MyHashMap

تتمثل مشكلة الصنف MyHashMap بالتابع size الموروث من الصنف MyBetterMap. انظر إلى شيفرته

فيما يلي:

```
public int size() {
    int total = 0;
    for (MyLinearMap<K, V> map: maps) {
        total += map.size();
    }
    return total;
}
```

كما ترى يضطرّ التابع للمرور عبر جميع الخرائط الفرعية لكي يحسب الحجم الكليّ. نظرًا لأننا نزيد عدد الخرائط الفرعية k بزيادة عدد المُدخّلات n، فإن k يتناسب مع n، ولذلك، يستغرق تنفيذ التابع size زمناً خطّيًا.

يجعل ذلك التابع put خطّيًا أيضًا لأنه يستخدم التابع size كما هو مُبيّن في الشيفرة التالية:

```
public V put(K key, V value) {
    V oldValue = super.put(key, value);

    if (size() > maps.size() * FACTOR) {
        rehash();
    }
    return oldValue;
}
```

إذا تركنا التابع size خطّيًا، فإننا نهدر كل ما فعلناه لجعل التابع put ثابت الزمن.

لحسن الحظ، هناك حلّ بسيط رأيناه من قبل، وهو أننا سنحتفظ بعدد المُدخّلات ضمن متغير نسخة instance variable، وسنحدّثه كلما استدعينا تابعًا يُجري تعديلًا عليه.

ستجد الحل في مستودع الكتاب في الملف MyFixedHashMap.java. انظر إلى بداية تعريف الصنف:

```
public class MyFixedHashMap<K, V> extends MyHashMap<K, V> implements
    Map<K, V> {

    private int size = 0;
```

```
public void clear() {
    super.clear();
    size = 0;
}
```

بدلاً من تعديل الصنف `MyHashMap`، عرّفنا صنفاً جديداً يمتدّ منه، وأضفنا إليه متغير النسخة `size`، وضبطنا قيمته المبدئية إلى صفر.

أجرينا أيضاً تعديلاً بسيطاً على التابع `clear`. استدعينا أولاً نسخة `clear` المُعرّفة في الصنف الأعلى (لتفريغ الخرائط الفرعية)، ثم حدثنا قيمة `size`.

كانت التعديلات على التابعين `remove` و `put` أعقد قليلاً؛ لأننا عندما نستدعي نسخها في الصنف الأعلى، فإننا لا نستطيع معرفة ما إذا كان حجم الخرائط الفرعية قد تغيّر أم لا. تُوضّح الشيفرة التالية الطريقة التي حاولنا بها معالجة تلك المشكلة:

```
public V remove(Object key) {
    MyLinearMap<K, V> map = chooseMap(key);
    size -= map.size();
    V oldValue = map.remove(key);
    size += map.size();
    return oldValue;
}
```

يستخدم التابع `remove` التابع `chooseMap` لكي يعثر على الخريطة المناسبة، ثم يطرح حجمها. بعد ذلك، يستدعي تابع الخريطة الفرعية `remove` الذي قد يُغيّر حجم الخريطة، حيث يعتمد ذلك على ما إذا كان قد وجد المفتاح فيها أم لا، ثم يضيف الحجم الجديد للخريطة الفرعية إلى `size`، وبالتالي تصبح القيمة النهائية صحيحة.

أعدنا كتابة التابع `put` باتباع نفس الأسلوب:

```
public V put(K key, V value) {
    MyLinearMap<K, V> map = chooseMap(key);
    size -= map.size();
    V oldValue = map.put(key, value);
    size += map.size();

    if (size() > maps.size() * FACTOR) {
        size = 0;
    }
}
```

```

        rehash();
    }
    return oldValue;
}

```

واجهنا نفس المشكلة هنا: عندما استدعينا تابع الخريطة الفرعية put، فإننا لا نعرف ما إذا كان قد أضاف مدخلًا جديدًا أم لا، ولذلك استخدمنا نفس الحل، أي بطرح الحجم القديم، ثم إضافة الحجم الجديد. والآن، أصبح تنفيذ التابع size بسيطًا:

```

public int size() {
    return size;
}

```

ويستغرق زمنًا ثابتًا بوضوح.

عندما شخّصنا أداء هذا الحل، وجدنا أن الزمن الكليّ لإضافة عدد n من المفاتيح يتناسب مع n، ويعني ذلك أن كل استدعاء للتابع put يستغرق زمنًا ثابتًا كما هو متوقع.

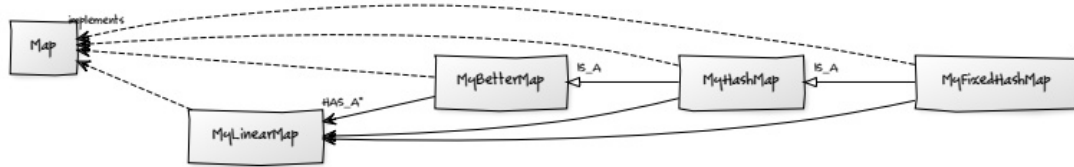
11.6 مخططات أصناف UML

كان أحد التحديات التي واجهناها عند العمل مع شيفرة هذا الفصل هو وجود عددٍ كبيرٍ من الأصناف التي يعتمد بعضها على بعض. انظر إلى العلاقات بين تلك الأصناف:

- MyLinearMap يحتوي على LinkedList ويُنفَّذ Map.
- MyBetterMap يحتوي على الكثير من كائنات الصنف MyLinearMap ويُنفَّذ Map.
- MyHashMap يمتد من الصنف MyBetterMap، ولذلك يحتوي على كائناتٍ تنتمي إلى الصنف MyLinearMap ويُنفَّذ Map.
- MyFixedHashMap يمتد من الصنف MyHashMap ويُنفَّذ Map.

لتسهيل فهم هذا النوع من العلاقات، يلجأ مهندسو البرمجيات إلى استخدام مخططات أصناف UML -اختصارًا إلى لغة النمذجة الموحدة Unified Modeling Language. تُعدّ مخططات الأصناف class diagram واحدةً من المعايير الرسومية التي تُعرّفها UML.

يُمثّل كل صنفٍ في تلك المخططات بصندوق، بينما تُمثّل العلاقات بين الأصناف بأسهم. تُعرض الصورة التالية مخطط أصناف UML للأصناف المُستخدمة في التمرين السابق، وهي مُولّدة تلقائيًا باستخدام أداة yUML المتاحة عبر الإنترنت.



تُمثّل العلاقات المختلفة بأنواع مختلفة من الأسهم:

- تشير الأسهم ذات الرؤوس الصلبة إلى علاقات من نوع HAS-A. على سبيل المثال، تحتوي كل نسخة من الصنف MyBetterMap على نسخ متعددة من الصنف MyLinearMap، ولذلك هي متصلة بأسهم صلبة.
- تشير الأسهم ذات الرؤوس المجوفة والخطوط الصلبة إلى علاقات من نوع IS-A. على سبيل المثال، يمتد الصنف MyHashMap من الصنف MyBetterMap، ولذلك ستجدهما موصولين بسهم IS-A.
- تشير الأسهم ذات الرؤوس المجوفة والخطوط المتقطعة إلى أن الصنف يُنفَّذ واجهة. تُنفَّذ جميع الأصناف في هذا المخطط الواجهة Map.

تُوفّر مخططات أصناف UML طريقةً موجزةً لتوضيح الكثير من المعلومات عن مجموعة من الأصناف، وتُستخدم عادةً في مراحل التصميم للإشارة إلى تصاميم بديلة، وفي مراحل التنفيذ لمشاركة التصور العام عن المشروع، وفي مراحل النشر لتوثيق التصميم.



هل تريد كتابة سيرة ذاتية احترافية؟

نساعذك في إنشاء سيرة ذاتية احترافية عبر خبراء توظيف
مختصين في أكبر منصة توظيف عربية عن بعد

أنشئ سيرتك الذاتية الآن

12. الواجهة TreeMap

سنناقش في هذا الفصل تنفيذًا جديدًا للواجهة Map يُعرّف باسم شجرة البحث الثنائية binary search tree. يشيع استخدام هذا التنفيذ عند الحاجة إلى الاحتفاظ بترتيب العناصر.

12.1 ما هي مشكلة التعمية hashing؟

يُفترض أن تكون على معرفة بالواجهة Map، وبالصنف المُنفذ لها HashMap الذي تُوقّره جافا. إذا كنت قد قرأت الفصل السابق الذي نُقدنا فيه نفس الواجهة باستخدام جدول hash table، فيُفترض أنك تُعرف الكيفية التي يَعْمَل بها الصنف HashMap، والسبب الذي لأجله تَستغرقُ توابع ذلك التنفيذ زمنًا ثابتًا.

يشيع استخدام الصنف HashMap بفضل كفاءته العالية، ولكنه مع ذلك ليس التنفيذ الوحيد للواجهة Map. فهناك أسبابٌ عديدةٌ قد تدفعك لاختيار تنفيذٍ آخر، منها:

1. قد تستغرق عملية حساب شيفرة التعمية زمنًا طويلًا. فعلى الرغم من أن عمليات الصنف HashMap تستغرق زمنًا ثابتًا، فقد يكون ذلك الزمن كبيرًا.

2. تَعْمَل التعمية بشكلٍ جيّدٍ فقط عندما تُوزَّع دالةُ التعمية hash function المفاتيح بالتساوي على الخرائط الفرعية، ولكنّ تصميم دوالّ التعمية لا يُعدّ أمرًا سهلاً، فإذا احتوت خريطةُ فرعيةٌ معيّنةً على مفاتيحٍ كثيرة، تقل كفاءة الصنف HashMap.

3. لا تُخزّن المفاتيح في الجدول وفقًا لترتيبٍ معيّن، بل قد يتغير ترتيبها عند إعادة ضبط حجم الجدول وإعادة حساب شيفرات التعمية للمفاتيح. بالنسبة لبعض التطبيقات، قد يكون الحفاظ على ترتيب المفاتيح ضروريًا أو مفيدًا على الأقل.

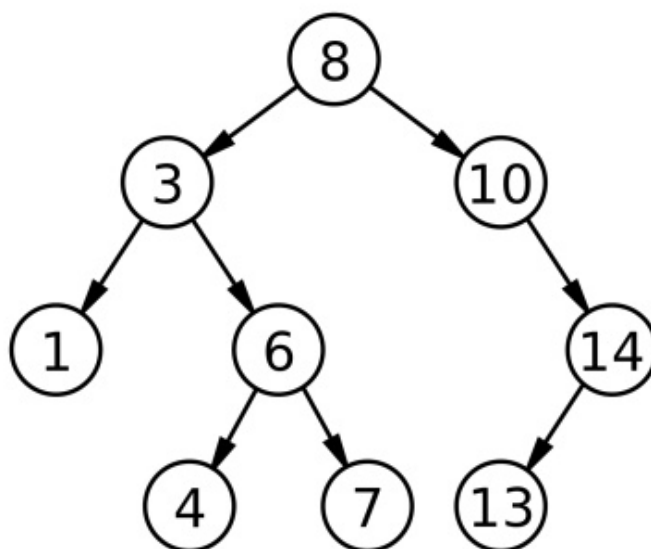
من الصعب حل كل تلك المشكلات في الوقت نفسه، ومع ذلك، تُوفّر جافا التنفيذ TreeMap الذي يُعالج بعضًا منها:

1. لا يُستخدم ذلك الصنف دالةً تعميةً، وبالتالي، يتجنّب الزمن الإضافي اللازم لحساب شيفرات التعمية، كما يُجنّبنا صعوبات اختيار دالةً تعميةً مناسبة.
 2. تُخزّن المفاتيح في الصنف TreeMap بهيئة شجرة بحث ثنائية، مما يُسهّل من التنقل في المفاتيح وفقًا لترتيبٍ معيّن وبزمنٍ خطّي.
 3. يتناسب زمن تنفيذ غالبية توابع الصنف TreeMap مع $\log(n)$ ، والتي رغم أنها ليست بكفاءة الزمن الثابت، ولكنها ما تزال جيدة جدًا.
- سنشرح طريقة عمل أشجار البحث الثنائية في القسم التالي ثم سنستخدمها لتنفيذ الواجهة Map، وأخيرًا، سنحلّل أداء التوابع الأساسية في الخرائط المُنفّذة باستخدام شجرة.

12.2 أشجار البحث الثنائية

شجرة البحث الثنائية عبارة عن شجرة تحتوي كل عقدة فيها على مفتاح، كما تتوفّر فيها "خاصية BST" التي تنص على التالي:

1. إذا كان لأي عقدة أب عقدة ابنة يسرى، فلا بُدّ أن تكون قيم جميع المفاتيح الموجودة في الشجرة الفرعية اليسرى أصغر من قيمة مفتاح تلك العقدة.
2. إذا كان لأي عقدة أب عقدة ابنة يمينى، فلا بُدّ أن تكون قيم جميع المفاتيح الموجودة في الشجرة الفرعية اليمينية أكبر من قيمة مفتاح تلك العقدة.



تعرض الصورة السابقة شجرة أعدادٍ صحيحةٍ تُحقّق الشروط السابقة. هذه الصورة مأخوذة من مقالة ويكيبيديا موضوعها **أشجار البحث الثنائية**، والتي قد تفيدك لحل هذا التمرين.

لاحظ أن مفتاح عقدة الجذر يساوي 8. يُمكنك التأكد من أن مفاتيح العقد الموجودة على يسار عقدة الجذر أقل من 8 بينما مفاتيح العقد الموجودة على يمينها أكبر من 8. تأكد من تحقق نفس الشرط للعقد الأخرى.

لا يستغرق البحث عن مفتاح ما ضمن شجرة بحثٍ ثنائيةٍ زمنًا طويلاً، لأنك غير مُضطّر للبحث في كامل الشجرة، وإنما عليك أن تبدأ من جذر الشجرة، ومن ثم، تُطبّق الخوارزمية التالية:

1. افحص قيمة المفتاح الهدف `target` الذي تبحث عنه وطابقه مع قيمة مفتاح العقدة الحالية. فإذا كانا متساويين، فقد انتهيت بالفعل.

2. أمّا إذا كان المفتاح `target` أصغر من المفتاح الحالي، ابحث في الشجرة الموجودة على اليسار، فإذا لم تكن موجودة، فهذا يعني أن المفتاح `target` غير موجود في الشجرة.

3. وأمّا إذا كان المفتاح `target` أكبر من المفتاح الحالي، ابحث في الشجرة الموجودة على اليمين. فإذا لم تكن موجودة، فهذا يعني أن المفتاح `target` غير موجود في الشجرة.

يعني ما سبق أنك مضطّر للبحث في عقدة ابنة واحدة فقط لكل مستوى ضمن الشجرة. فعلى سبيل المثال، إذا كنت تبحث عن مفتاح `target` قيمته تساوي 4 في الرسم السابقة، فعليك أن تبدأ من عقدة الجذر التي تحتوي على المفتاح 8، ولأن المفتاح المطلوب أقل من 8، فستذهب إلى اليسار، ولأنه أكبر من 3، فستذهب إلى اليمين، ولأنه أقل من 6، فستذهب إلى اليسار، ثم ستعثر على المفتاح الذي تبحث عنه.

تطلب البحث عن المفتاح في المثال السابق 4 عملياتٍ موازنةٍ رغم أن الشجرة تحتوي على 9 مفاتيح. يتناسب عدد الموازنات المطلوبة في العموم مع ارتفاع الشجرة وليس مع عدد المفاتيح الموجودة فيها.

ما الذي نستنتجه من ذلك بخصوص العلاقة بين ارتفاع الشجرة h وعدد العقد n ؟ إذا بدأنا بارتفاعٍ قصيرٍ وزدناه تدريجياً، فسنحصل على التالي:

- إذا كان ارتفاع الشجرة h يساوي 1، فإن عدد العقد n ضمن تلك الشجرة يساوي 1.
- وإذا كان ارتفاع الشجرة h يساوي 2، فيمكننا أن نضيف عقدتين أخريين، وبالتالي، يصبح عدد العقد n في الشجرة مساوياً للقيمة 3.
- وإذا كان ارتفاع الشجرة h يساوي 3، فيمكننا أن نضيف ما يصل إلى أربع عقدٍ أخرى، وبالتالي، يصبح عدد العقد n مساوياً للقيمة 7.
- وإذا كان ارتفاع الشجرة h يساوي 4، فيمكننا أن نضيف ما يصل إلى ثماني عقدٍ أخرى، وبالتالي، يصبح عدد العقد n مساوياً للقيمة 15.

ربما لاحظت النمط المشترك بين تلك الأمثلة. إذا رَقَمْنَا مستويات الشجرة تدريجيًّا من 1 إلى h ، فإن عدد العقد في أيِّ مستوى i يَصِلُ إلى 2^{i-1} كحدٍّ أقصى، وبالتالي، يكون عددُ العقدِ الإجماليِّ في عدد h من المستويات هو $2^h - 1$. إذا كان:

$$n = 2^h - 1$$

بتطبيق لوغاريتم الأساس 2 على طرفي المعادلة السابقة، نحصل على التالي:

$$\log_2 n \approx h$$

إذًا، يتناسب ارتفاع الشجرة مع $\log(n)$ إذا كانت الشجرة ممتلئة؛ أي إذا كان كل مستوى فيها يحتوي على العدد الأقصى المسموح به من العقد.

وبالتالي، يتناسب زمنُ البحثِ عن مفتاحٍ ضمن شجرة بحثٍ ثنائيةٍ مع $\log(n)$. يُعدُّ ذلك صحيحًا سواءً أكانت الشجرة ممتلئة كليًّا أم جزئيًّا، ولكنه ليس صحيحًا في المطلق، وهو ما سنراه لاحقًا.

يُطلَق على الخوارزميات التي تَسْتَغْرِقُ زمنًا يتناسب مع $\log(n)$ اسم "خوارزمية لوغاريتمية"، وتنتمي إلى ترتيب النمو $O(\log(n))$.

12.3 تمرين 10

ستكتب في هذا التمرين تنفيذًا للواجهة Map باستخدام شجرة بحثٍ ثنائيةٍ.

انظر إلى التعريف المبدئيِّ للصنف MyTreeMap:

```
public class MyTreeMap<K, V> implements Map<K, V> {

    private int size = 0;
    private Node root = null;
```

يحتفظ متغيِّرُ النسخة size بعدد المفاتيح بينما يحتوي root على مرجع reference يشير إلى عقدة الجذر الخاصّة بالشجرة. إذا كانت الشجرة فارغةً، يحتوي root على القيمة null وتكون قيمة size مساويةً للصفر.

انظر إلى الصنف Node المُعرَّف داخل الصنف MyTreeMap:

```
protected class Node {
    public K key;
    public V value;
    public Node left = null;
```

```

    public Node right = null;

    public Node(K key, V value) {
        this.key = key;
        this.value = value;
    }
}

```

تحتوي كل عقدة على زوج مفتاح/قيمة وعلى مراجع تشير إلى العقد الأبناء left و right. قد تكون إحداها أو كليهما فارغة أي تحتوي على القيمة null.

من السهل تنفيذ بعض توابع الواجهة Map مثل size و clear:

```

public int size() {
    return size;
}

public void clear() {
    size = 0;
    root = null;
}

```

من الواضح أن التابع size يستغرق زمنًا ثابتًا.

قد تظن للوهلة الأولى أن التابع clear يستغرق زمنًا ثابتًا، ولكن فكر بالتالي: عندما تُضبط قيمة root إلى القيمة null، يستعيد كانس المهملات garbage collector العقد الموجودة في الشجرة ويستغرق لإنجاز ذلك زمنًا خطيًا. هل ينبغي أن يُحسب العمل الذي يقوم به كانس المهملات؟ ربما.

ستكتب في القسم التالي تنفيذًا لبعض التوابع الأخرى لا سيّما التابعين الأهمّ get و put.

12.4 تنفيذ الصنف TreeMap

ستجد ملفات الشيفرة التالية في مستودع الكتاب:

- MyTreeMap.java: يحتوي على الشيفرة الموضّحة في الأعلى مع تصورٍ مبدئيٍّ للتوابع غير المكتملة.
- MyTreeMapTest.java: يحتوي على اختبارات وحدةٍ للصنف MyTreeMap.

ننّفذ الأمر ant build لتصريف ملفات الشيفرة، ثم ننّفذ الأمر ant MyTreeMapTest. قد تفشل بعض الاختبارات لأنّ هناك بعض التوابع التي ينبغي عليك إكمالها أولاً.

وَقَرْنَا تصوّرًا مبدئيًا للتابعين `get` و `containsKey`. يَستخدِم كلاهما التابع `findNode` المُعرَّف باستخدام المُعدِّل `private`، لأنه ليس جزءًا من الواجهة `Map`. انظر إلى بداية تعريفه:

```
private Node findNode(Object target) {
    if (target == null) {
        throw new IllegalArgumentException();
    }

    @SuppressWarnings("unchecked")
    Comparable<? super K> k = (Comparable<? super K>) target;

    // TODO: FILL THIS IN!
    return null;
}
```

يشير المعامل `target` إلى المفتاح الذي نبحث عنه. إذا كانت قيمة `target` تساوي `null`، يُبلِّغ التابع `findNode` عن اعتراض `exception`. في الواقع، بإمكان بعض تنفيذات الواجهة `Map` معالجة الحالات التي تكون فيها قيمة المفتاح فارغة، ولكن لأننا في هذا التنفيذ نستخدم شجرة بحث ثنائية، فلا بُدَّ أن نتمكّن من موازنة المفاتيح، ولذلك، يُشكّل التعامل مع القيمة `null` مشكلةً، ولذا ولكي نُبسّط الأمور، لن نسمَح لهذا التنفيذ باستخدام القيمة `null` كمفتاح.

تُوضِّح الأسطر التالية كيف يمكننا أن نوازن قيمة المفتاح `target` مع قيمة مفتاح ضمن الشجرة. تشير نسخة التابعين `get` و `containsKey` إلى أن المُصَرِّف يتعامل مع `target` كما لو أنه ينتمي إلى النوع `Object`، ولأننا نريد موازنته مع المفاتيح، فإننا نحوّل نوع `target` إلى النوع `Comparable<? super K>` لكي يُصبح قابلاً للموازنة مع كائنٍ من النوع `K` أو أيٍّ من أصنافه الأعلى `superclass`. يُمكنك قراءة المزيد عن أنواع محارف البدل (باللغة الإنجليزية).

ليس المقصودُ من هذا التمرين احتراف التعامل مع نظام الأنواع في لغة جافا، فدورك فقط هو أن تُكْمِل التابع `findNode`. إذا وجد ذلك التابع عقدةً تحتوي على قيمة `target` كمفتاح، فعليه أن يعيدها، أما إذا لم يجدها، فعليه أن يعيد القيمة `null`. ينبغي أن تنجح اختبارات التابعين `get` و `containsKey` بعد أن تنتهي من إكمال هذا التابع.

ينبغي للحل الخاص بك أن يبحث في مسارٍ واحدٍ فقط ضمن الشجرة لا أن يبحث في كامل الشجرة، أي سيستغرق زمنًا يتناسب مع ارتفاع الشجرة.

والآن، عليك أن تُكْمِل التابع `containsValue`، ولمساعدتك على ذلك، وَقَرْنَا التابع المساعد `equals` الذي يُوازن بين قيمة `target` وقيمة مفتاحٍ معيّن. على العكس من المفاتيح، قد لا تكون القيم المُخزّنة في

الشجرة قابلةً للموازنة، وبالتالي، لا يُمكننا أن نستخدم معها التابع `compareTo`، وإنما علينا أن نستخدم التابع `equals` بالمتغير `target`.

بخلاف التابع `findNode`، سيضطرّ التابع `containsValue` للبحث في كامل الشجرة، أي يتناسب زمن تشغيله مع عدد المفاتيح n وليس مع ارتفاع الشجرة h .

والآن، أكمل متنّ التابع `put`. وقرّنا له شيفرةً مبدئيةً تعالج الحالات البسيطة فقط:

```
public V put(K key, V value) {
    if (key == null) {
        throw new IllegalArgumentException();
    }
    if (root == null) {
        root = new Node(key, value);
        size++;
        return null;
    }
    return putHelper(root, key, value);
}

private V putHelper(Node node, K key, V value) {
    // TODO: Fill this in.
}
```

إذا حاولت استخدام القيمة الفارغة `null` كمفتاح، سيُبلّغ `put` عن اعتراض. إذا كانت الشجرة فارغةً، سيُنشئ التابع `put` عقدةً جديدةً، ويهيئ المتغير `root` المُعرّف فيها.

أما إذا لم تكن فارغةً، فإنه يُستدعى التابع `putHelper` المُعرّف باستخدام المُعدّل `private` لأنه ليس جزءاً من الواجهة `Map`.

أكمل متنّ التابع `putHelper` واجعله يبحث ضمن الشجرة وفقاً لما يلي:

1. إذا كان المفتاح `key` موجوداً بالفعل ضمن الشجرة، عليه أن يُستبدل القيمة الجديدة بالقيمة القديمة، ثم يعيدها.
2. إذا لم يكن المفتاح `key` موجوداً في الشجرة، فعليه أن يُنشئ عقدةً جديدةً، ثم يضيفها إلى المكان الصحيح، وأخيراً، يعيد القيمة `null`.

ينبغي أن يستغرق التابع put زمنًا يتناسب مع ارتفاع الشجرة h وليس مع عدد العناصر n . سيكون من الأفضل لو بحثت في الشجرة مرةً واحدةً فقط، ولكن إذا كان البحث فيها مرتين أسهل بالنسبة لك، فلا بأس. سيكون التنفيذ أبطأ، ولكنه لن يؤثر على ترتيب نموه.

وأخيرًا، عليك أن تكمل متن التابع keySet. يعيد ذلك التابع -وفقًا للتوثيق (باللغة الإنجليزية)- قيمة من النوع Set بإمكانها المرور عبر جميع مفاتيح الشجرة بترتيب تصاعديٍّ وفقًا للتابع compareTo. كنا قد استخدمنا الصنف HashSet في الفصل الثامن المفهرس Indexer. يُعدّ ذلك الصنف تنفيذًا للواجهة Set ولكنه لا يحافظ على ترتيب المفاتيح. في المقابل، يتوفّر التنفيذ LinkedHashMap الذي يحافظ على ترتيب المفاتيح.

يُنشئ التابع keySet قيمةً من النوع LinkedHashMap ويعيدها كما يلي:

```
public Set<K> keySet() {
    Set<K> set = new LinkedHashMap<K>();
    return set;
}
```

عليك أن تكمل هذا التابع بحيث تجعله يضيف المفاتيح من الشجرة إلى المجموعة set بترتيب تصاعديّ.

قد تحتاج إلى كتابة تابع مساعدٍ، وقد ترغب بجعله تعاوديًا recursive، كما قد يساعدك على الحلّ قراءة بعض المعلومات عن أسلوب "في الترتيب" التنقل في الشجرة (باللغة الإنجليزية).

ينبغي أن تنجح جميع الاختبارات بعد أن تنتهي من إكمال هذا التابع. سنعرض حل هذا التمرين ونفحص أداء التوابع الأساسية في الصنف في فصل لاحق من هذا الكتاب.

دورة إدارة تطوير المنتجات



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



13. شجرة البحث الثنائي Binary Search Tree

سنناقش في هذا الفصل حل تمرين الفصل السابق، ونختبر أداء الخرائط المُنفَّذة باستخدام شجرة، وبعدها سنناقش إحدى مشاكل ذلك التنفيذ والحلّ الذي يقدمه الصنف `TreeMap` لتلك المشكلة.

13.1 الصنف `MyTreeMap`

وقرنا في الفصل المشار إليها تصوّرًا مبدئيًا للصنف `MyTreeMap`، وتركنا للقارئ مهمة إكمال توابعه. وسنُكملها الآن معًا، ولنبدأ بالتابع `findNode`:

```
private Node findNode(Object target) {
    // بعض التنفيذات تعد null مفتاحًا ولكن ليس في هذه الحالة
    if (target == null) {
        throw new IllegalArgumentException();
    }

    // هذا ما يُسعد المُصرّف
    @SuppressWarnings("unchecked")
    Comparable<? super K> k = (Comparable<? super K>) target;

    // البحث الحقيقي
    Node node = root;
    while (node != null) {
        int cmp = k.compareTo(node.key);
        if (cmp < 0)
```

```

        node = node.left;
    else if (cmp > 0)
        node = node.right;
    else
        return node;
}
return null;
}

```

يستخدم التابعان `containsKey` و `getNode` التابع `findNode`، ولأنه ليس جزءاً من الواجهة `Map`، عرّفناه باستخدام المُعدّل `private`. يُمثل المعامل `target` المفتاح الذي نبحث عنه. كنا قد شرحنا الجزء الأول من هذا التابع في الفصل المشار إليه:

- لا تُعدّ `null` قيمةً صالحةً كمفتاحٍ في هذا التنفيذ.
- ينبغي أن نحوّل نوعَ المعامل `target` إلى `Comparable` قبل أن نُسدعيّ تابعه `compareTo`. استخدمنا أكثر أنواع محارف البدل عمومية، حيث يعمل مع أي نوع يُنفذ الواجهة `Comparable`، كما أن تابعه `compareTo` يستقبل النوع `K` أو أيّاً من أنواعه الأعلى `supertype`.

يُجرى البحث على النحو التالي: نضبط متغير الحلقة `node` إلى عقدة الجذر، وفي كلّ تكرارٍ، نوازن بين المفتاح `target` وقيمة `node.key`. إذا كان `target` أصغرَ من مفتاح العقدة الحالية، سننتقل إلى عقدة الابن اليسرى، أما إذا كان أكبرَ منه، سننتقل إلى عقدة الابن اليمنى، وإذا كانا متساويين، سنعيد العقدة الحالية. إذا وصلنا إلى قاع الشجرة دون أن نعثر على المفتاح المطلوب، فهذا يعني أنه غير موجود فيها، وسنعيد في تلك الحالة القيمة الفارغة `null`.

13.2 البحث عن القيم values

كما أوضحنا في نفس الفصل المشار إليها في الأعلى، يتناسب زمن تنفيذ التابع `findNode` مع ارتفاع الشجرة وليس مع عدد العقد الموجودة فيها؛ وذلك لأننا غير مضطّرين للبحث في كامل الشجرة، ولكن بالنسبة للتابع `containsValue`، فإننا سنضطرّ للبحث بالقيم وليس المفاتيح، ولأن خاصية BST لا تُطبّق على القيم، فإننا سنضطرّ إلى البحث في كامل الشجرة.

يستخدم الحلّ التالي التعاود `recursion`:

```

public boolean containsValue(Object target) {
    return containsValueHelper(root, target);
}

```

```
private boolean containsValueHelper(Node node, Object target) {
    if (node == null) {
        return false;
    }
    if (equals(target, node.value)) {
        return true;
    }
    if (containsValueHelper(node.left, target)) {
        return true;
    }
    if (containsValueHelper(node.right, target)) {
        return true;
    }
    return false;
}
```

يُستقبل التابع `containsValue` المعامل `target`، ويُمرّره مع معامل إضافي يحتوي على عقدة الجذر إلى التابع `containsValueHelper`.

يُعمل التابع `containsValueHelper` وفقاً لما يلي:

- تفحص تعليمة `if` الأولى الحالة الأساسية للتعاود: إذا كانت قيمة `node` مساويةً للقيمة الفارغة `null`، فإن التابع وصل إلى قاع الشجرة دون إيجاد القيمة المطلوبة `target`، ويعيد عندها القيمة `false`. انتبه، يعني ذلك أن القيمة `target` غير موجودة في واحدٍ فقط من مسارات الشجرة لا في مسارات الشجرة كلّها، ولذا ما يزال من الممكن العثور عليها في مسارٍ آخر.
- تفحص تعليمة `if` الثانية ما إذا كان التابع قد وجد القيمة المطلوبة، وفي تلك الحالة، يعيد التابع القيمة `true`، أما إذا لم يجدها، فإنه يستمر في البحث.
- تستدعي الحالة الشرطية الثالثة التابع تعاودياً لكي يبحث عن نفس القيمة، أي `target`، في الشجرة الفرعية اليسرى. إذا وجدها فيها، فإنه يعيد القيمة `true` مباشرةً دون أن يحاول البحث في الشجرة الفرعية اليمنى، أما إذا لم يجدها فيها، فإنه يستمر في البحث.
- تبحث الحالة الشرطية الرابعة عن القيمة المطلوبة في الشجرة الفرعية اليمنى. إذا وجدها فيها، فإنه يعيد القيمة `true`، أما إذا لم يجدها، فإنه يعيد القيمة `false`.

يمرّ التابع السابق عبر كل عقدةٍ من الشجرة، ولهذا، يستغرق زمناً يتناسب مع عدد العقد.

13.3 تنفيذ التابع put

يُعدّ التابع put أعقد قليلاً من التابع get؛ لأن عليه أن يتعامل مع حالتين: الأولى عندما يكون المفتاح المُعطى موجوداً في الشجرة بالفعل، وينبغي عندها أن يَستبدله ويعيد القيمة القديمة، والثانية عندما لا يكون موجوداً، وعندها ينبغي أن يُنشئ عقدةً جديدةً ثم يضعها في المكان الصحيح.

كنا قد وقّنا الشيفرة المبدئية التالية لذلك التابع في الفصل المذكور:

```
public V put(K key, V value) {
    if (key == null) {
        throw new IllegalArgumentException();
    }
    if (root == null) {
        root = new Node(key, value);
        size++;
        return null;
    }
    return putHelper(root, key, value);
}
```

وكان المطلوب هو إكمال متن التابع putHelper. انظر إلى شيفرته فيما يلي:

```
private V putHelper(Node node, K key, V value) {
    Comparable<? super K> k = (Comparable<? super K>) key;
    int cmp = k.compareTo(node.key);

    if (cmp < 0) {
        if (node.left == null) {
            node.left = new Node(key, value);
            size++;
            return null;
        } else {
            return putHelper(node.left, key, value);
        }
    }
    if (cmp > 0) {
        if (node.right == null) {
            node.right = new Node(key, value);
            size++;
            return null;
        } else {
            return putHelper(node.right, key, value);
        }
    }
    return value;
}
```

```

        size++;
        return null;
    } else {
        return putHelper(node.right, key, value);
    }
}
V oldValue = node.value;
node.value = value;
return oldValue;
}

```

يُضَبِّطُ المعاملُ الأولُ `node` مبدئيًا إلى عقدة الجذر `root`، وفي كل مرة نَسْتَدْعِي فيها التابع تعاوديًا، يشير المعامل إلى شجرة فرعية مختلفة. مثل التابع `get`، إستخدَمنا التابع `compareTo` لتحديد المسار الذي سنتبعه في الشجرة. إذا تحقَّق الشرط `cmp < 0`، يكون المفتاح المطلوب إضافته أقل من `node.key`، وعندها يكون علينا فحص الشجرة الفرعية اليسرى. هنالك حالتان:

- إذا كانت الشجرة الفرعية فارغة، فإن `node.left` تحتوي على `null`، وعندها نكون قد وصلنا إلى قاع الشجرة دون أن نعثر على المفتاح `key`. في تلك الحالة، نكون قد تأكدنا من أن المفتاح `key` غير موجود في الشجرة، وعرفنا المكان الذي ينبغي أن نضيف المفتاح إليه، ولذلك، نُنشئ عقدةً جديدةً، ونضيفها كعقدة ابنة يسرى للعقدة `node`.
- إن لم تكن الشجرة فارغة، نَسْتَدْعِي التابع تعاوديًا للبحث في الشجرة الفرعية اليسرى.

في المقابل، إذا تحقَّق الشرط `cmp > 0`، يكون المفتاح المطلوب إضافته أكبر من `node.key`، وعندها يكون علينا فحص الشجرة الفرعية اليمنى، وسيكون علينا معالجة نفس الحالتين السابقتين. أخيرًا، إذا تحقَّق الشرط `cmp == 0`، نكون قد عثرنا على المفتاح داخل الشجرة، وعندها، نستطيع أن نستبدله ونعيد القيمة القديمة.

كتبنا هذا التابع تعاوديًا لكي نُسهِّل من قراءته، ولكن يُمكن كتابته أيضًا بأسلوبٍ تكراريٍّ. يُمكنك القيام بذلك كتمرين.

13.4 التنقل بالترتيب In-order

كنا قد طلبنا منك كتابة التابع `keySet` لكي يعيد مجموعةً من النوع `Set` تحتوي على مفاتيح الشجرة مُرتَّبة تصاعديًا. لا يعيد هذا التابع في التنفيذات الأخرى من الواجهة `Map` المفاتيح وفقًا لأيِّ ترتيب، ولكن لأن هذا التنفيذ يتمتع بالبساطة والكفاءة، فإنه يَسْمَح لنا بترتيب المفاتيح، وعلينا أن نَسْتفيد من ذلك.

انظر إلى شيفرة التابع فيما يلي:

```

public Set<K> keySet() {
    Set<K> set = new LinkedHashSet<K>();
    addInOrder(root, set);
    return set;
}

private void addInOrder(Node node, Set<K> set) {
    if (node == null) return;
    addInOrder(node.left, set);
    set.add(node.key);
    addInOrder(node.right, set);
}

```

كما ترى فقد أنشأنا قيمةً من النوع `LinkedHashSet` في التابع `keySet`. يُنفَّذ ذلك النوع الواجهة `Set` ويحافظ على ترتيب العناصر (بخلاف معظم تنفيذات تلك الواجهة). نُسْتَدْعِي بعد ذلك التابع `addInOrder` للتنقل في الشجرة.

يشير المعامل الأول `node` مبدئيًا إلى جذر الشجرة، ونُسْتَحْدِمُه -كما يُفْتَرَض أن تتوقَّع- للتنقل في الشجرة تعاوديًا. يجتاز التابع `addInOrder` الشجرة بأسلوب "في الترتيب" المعروف.

إذا كانت العقدة `node` فارغةً، يَعْنِي ذلك أن الشجرة الفرعية فارغةً، وعندها يعود التابع دون إضافة أي شيء إلى المجموعة `set`، أما إذا لم تكن فارغةً، نقوم بما يلي:

1. نجتاز الشجرة الفرعية اليسرى بالترتيب.

2. نضيف `node.key`.

3. نجتاز الشجرة الفرعية اليمنى بالترتيب.

تذكّر أن خاصية BST تضمن أن تكون جميع العقد الموجودة في الشجرة الفرعية اليسرى أقلّ من `node.key` وأن تكون جميع العقد الموجودة في الشجرة الفرعية اليمنى أكبر منه، أي أننا نضيف `node.key` إلى الترتيب الصحيح.

بتطبيق نفس المبدأ تعاوديًا، نستنتج أن عناصر الشجرة الفرعية اليسرى واليمنى مُرتَّبة، كما أن الحالة الأساسية صحيحة: إذا كانت الشجرة الفرعية فارغةً، فإننا لا نضيف أيّة مفاتيح. يَعْنِي ما سبق أن هذا التابع يضيف جميع المفاتيح وفقًا لترتيبها الصحيح.

ولأن هذا التابع يمرّ عبر كل عقدة ضمن الشجرة مثله مثل التابع `containsValue`، فإنه يَسْتَغْرِقُ زمنًا يتناسب مع n .

13.5 التوابع اللوغاريتمية

يَستغْرِقُ التابَعانِ `get` و `put` في الصنف `MyTreeMap` زَمَنًا يَتَنَاسَبُ مَعَ ارتفاعِ الشجرة `h`. أوضحنا في الفصل المشار إليه أنه إذا كانت الشجرة ممثلةً أي كان كل مستوى منها يحتوي على الحد الأقصى من عدد العقد المسموح به، فإن ارتفاع تلك الشجرة يكون متناسبًا مع $\log(n)$.

نفترض الآن أن التابعين `get` و `set` يستغرقان زَمَنًا لوغاريتميًا، أي زَمَنًا يَتَنَاسَبُ مَعَ $\log(n)$ ، مع أننا لا نَظْمَنُ أن تكون الشجرة ممثلةً دائماً. يعتمد شكل الشجرة في العموم على المفاتيح وعلى الترتيب الذي تُضاف به.

سنختبر التنفيذ الذي كتبناه بمجموعتي بيانات لكي نرى كيف يعمل. المجموعة الأولى عبارة عن قائمةٍ تحتوي على سلاسلٍ نصيةٍ عشوائيةٍ، والثانية عبارة عن قائمةٍ تحتوي على علاماتٍ زمنيةٍ `timestamp` مُرتبةٍ تصاعديًا.

تُولدُ الشيفرةُ التاليةُ السلاسلَ النصيةَ العشوائية:

```
Map<String, Integer> map = new MyTreeMap<String, Integer>();

for (int i=0; i<n; i++) {
    String uuid = UUID.randomUUID().toString();
    map.put(uuid, 0);
}
```

يقع تعريف الصنف `UUID` ضمن حزمة `java.util`، ويُمكنه أن يُولّد مُعرّفَ هويةٍ فريدًا عمومياً `universally unique identifier` بأسلوبٍ عشوائي. تُعدّ تلك المُعرّفات ذات فائدةٍ كبيرةٍ في مختلف أنواع التطبيقات، ولكننا سنستخدمها في هذا المثال كطريقةٍ سهلةٍ لتوليد سلاسلٍ نصيةٍ عشوائيةٍ.

شغلنا الشيفرة التالية مع $n=16384$ وحسبنا زمن التنفيذ وارتفاع الشجرة النهائي، وحصلنا على الخرج التالي:

```
Time in milliseconds = 151
Final size of MyTreeMap = 16384
log base 2 of size of MyTreeMap = 14.0
Final height of MyTreeMap = 33
```

أضفنا أيضًا قيمة اللوغاريتم للأساس 2 إلى الخريطة لكي نرى طول الشجرة إذا كانت ممثلة. تشير النتيجة إلى أن شجرةً ممثلةً بارتفاعٍ يساوي 14 تحتوي على 16,384 عقدة.

في الواقع، ارتفاع شجرة السلاسل النصية العشوائية الفعلي هو 33، وهو أكبر من الحد الأدنى النظري ولكن ليس بشكل كبير. لكي نعثر على مفتاح ضمن مجموعة مكونة من 16,384 عقدة، سنضطرّ لإجراء 33 موازنة، أي أسرع بـ 500 مرة تقريبًا من البحث الخطي linear search.

يُعدّ هذا الأداء نموذجيًا للسلاسل النصية العشوائية والمفاتيح الأخرى التي لا تضاف وفقًا لأي ترتيب. رغم أن ارتفاع الشجرة النهائي يصل إلى ضعف الحد النظري الأدنى أو ثلاثة أضعافه، فهو ما يزال متناسبًا مع $\log(n)$ ، أي أقل بكثير من n ، حيث تنمو قيمة $\log(n)$ ببطء مع زيادة قيمة n لدرجة يصعب معها التمييز بين الزمن الثابت والزمن اللوغاريتمي عمليًا.

في المقابل، لا تعمل أشجار البحث الثنائية بهذه الكفاءة دائمًا. لنرى ما قد يحدث عند إضافة المفاتيح بترتيب تصاعدي. يستخدم المثال التالي علامات زمنية -بوحدات النانو ثانية- كمفاتيح:

```
MyTreeMap<String, Integer> map = new MyTreeMap<String, Integer>();

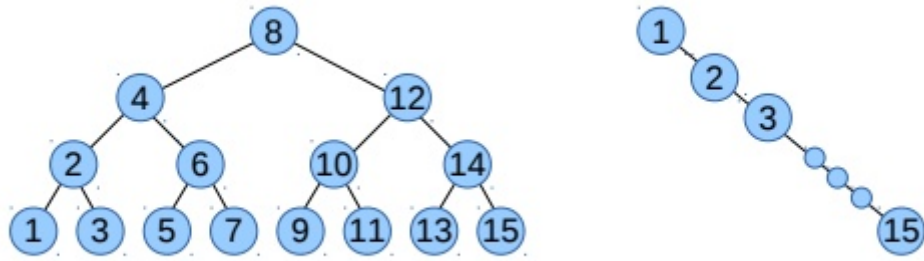
for (int i=0; i<n; i++) {
    String timestamp = Long.toString(System.nanoTime());
    map.put(timestamp, 0);
}
```

يعيد `System.nanoTime` عددًا صحيحًا من النوع `long` يشير إلى الزمن المُنقضي بوحدات النانو ثانية. نحصل على عدد أكبر قليلًا في كل مرة نستخدمه فيها. عندما نُحوّل تلك العلامات الزمنية إلى سلاسل نصية، فإنها تكون مُرتبة أبجديًا.

لنرى ما نحصل عليه عند التشغيل:

```
Time in milliseconds = 1158
Final size of MyTreeMap = 16384
log base 2 of size of MyTreeMap = 14.0
Final height of MyTreeMap = 16384
```

يتجاوز زمن التشغيل في هذه الحالة سبعة أضعاف زمن التشغيل في الحالة السابقة. إذا كنت تتساءل عن السبب، فألق نظرة على ارتفاع الشجرة النهائي 16384.



إذا أمعنت النظر في الطريقة التي يَعْمَلُ بها التابع `put`، فقد تفهم ما يحدث: ففي كل مرة نضيف فيها مفتاحًا جديدًا، فإنه يكون أكبر من جميع المفاتيح الموجودة في الشجرة، وبالتالي، نضطرّ دائمًا لاختيار الشجرة الفرعية اليمنى، ونضيف دائمًا العقدة الجديدة كعقدة ابن اليمنى للعقدة الواقعة على أقصى اليمين. نحصل بذلك على شجرة غير متزنة `unbalanced` تحتوي على عقدٍ أبناء يمنى فقط.

يتناسب ارتفاع تلك الشجرة مع n وليس $\log(n)$ ، ولذلك يصبح أداء التابعين `get` و `set` خطيًا لا لوغاريتميًا.

تعرض الصورة السابقة مثالًا عن شجرتين إحداها متزنة والأخرى غير متزنة. يُمكننا أن نرى أن ارتفاع الشجرة المتزنة يساوي 4 وعدد العقد الكلية يساوي 1–24 أي 15. تحتوي الشجرة غير المتزنة على نفس عدد العقد، ولكن ارتفاعها يساوي 15.

13.6 الأشجار المتزنة ذاتيًا Self-balancing trees

هناك حلّان محتملان لتلك المشكلة:

- يُمكننا أن نتجنّب إضافة المفاتيح إلى الخريطة بالترتيب، ولكن هذا الحل ليس ممكنًا دائمًا.
- يُمكننا أن نُنشئ شجرةً قادرةً على التعامل مع المفاتيح المرتبة تعاملًا أفضل.

يبدو الحل الثاني أفضل، وتتوقّر طرائق عديدة لتنفيذه. يُمكننا مثلًا أن نُعدّل التابع `put` لكي نجعله يفحص ما إذا كانت الشجرة قد أصبحت غير متزنة، وعندئذ، يعيد ترتيب العقد. يُطلَق على الأشجار التي تتميز بتلك المقدرة اسم "الأشجار المتزنة ذاتيًا"، ومن أشهرها شجرة AVL (اختصار `Adelson-Velskii Tree` حيث إن `Adelson` و `Velskii` هما مبتكرا هذه الشجرة)، وشجرة `red-black` التي يَستخدِمها صنف الجافا `TreeMap`.

إذا استخدمنا الصنف `TreeMap` بدلًا من الصنف `MyTreeMap` في الشيفرة السابقة، سيصبح زمن تشغيل مثال السلاسل النصية ومثال العلامات الزمنية هو نفسه، بل في الحقيقة، سيكون مثال العلامات الزمنية أسرع على الرغم من أن المفاتيح مُرتبة؛ لأنه يَستغْرِق وقتًا أقل لحساب شيفرة التعمية `hash`.

نستخلص مما سبق أن أشجار البحث الثنائية قادرةٌ على تنفيذ التابعين `get` و `put` بزمان لوغاريتمي بشرط إضافة المفاتيح إليها وفقًا لترتيبٍ يحافظ على اتزانها بشكلٍ كافٍ. في المقابل، تتجنب الأشجار المتزنة ذاتيًا تلك المشكلة بإنجاز بعض العمل الإضافي في كل مرة يُضاف فيها مفتاح جديد.

يُمكنك قراءة المزيد عن الأشجار المتزنة ذاتيًا.

13.7 تمرين إضافي

لم نُنفذ التابع `remove` في ذاك التمرين، ولكن يُمكنك أن تُجرب كتابته الآن. إذا حذفت عقدةً من وسط الشجرة، ستضطر إلى إعادة ترتيب العقد المتبقية لكي تحافظ على خاصية BST. ربما بإمكانك التفكير في طريقة إنجاز ذلك أو الاطلاع على الرابط (باللغة الإنجليزية).

تعدّ عمليتا حذف عقدة وإعادة الشجرة إلى الاتزان عمليتين متشابهتين، لذا إذا أتممت هذا التمرين، ستفهم طريقة عمل الأشجار المتزنة ذاتيًا فهماً أعمق.

دورة تطوير التطبيقات باستخدام لغة بايثون



احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن



14. حفظ البيانات عبر Redis

سنُكمل في التمارين القليلة القادمة بناء محرك البحث الذي تحدثنا عنه في الفصل السادس التنقل في الشجرة. يتكوّن أيّ محرك بحثٍ من الوظائف التالية:

- الزحف crawling: ويُنفَّذ من خلال برنامجٍ بإمكانه تحميلُ صفحة إنترنت وتحليلها واستخراج النص وأيّ روابط إلى صفحاتٍ أخرى.
- الفهرسة indexing: وتنفَّذ من خلال هيكل بيانات data structure بإمكانه البحث عن كلمة والعثور على الصفحات التي تحتوي على تلك الكلمة.
- الاسترجاع retrieval: وهي طريقةٌ لتجميع نتائج المُفهرس واختيار الصفحات الأكثر صلة بكلمات البحث.

إذا كنت قد أتممت تمرين الفصل الثامن المُفهرس Indexer، فقد نَقَذت مُفهرسًا بالفعل باستخدام خرائط جافا. سنناقش هذا التمرين هنا وسُنشئُ نسخةً جديدةً تُخزّن النتائج في قاعدة بيانات.

وإذا كنت قد أكملت تمرين الفصل السابع كل الطرق تؤدي إلى روما، فقد نَقَذت بالفعل زاحفًا يتبع أول رابطٍ يعثر عليه. سنُنشئُ في التمرين التالي نسخةً أعمّ تُخزّن كل رابطٍ تجده في رتل queue، وتتبع تلك الروابط بالترتيب.

في النهاية، ستُكلّف بالعمل على برنامج الاسترجاع.

سنُوفّر شيفرةً مبدئيةً أقصر في هذه التمارين، وسنعطيك فرصةً أكبر لاتخاذ القرارات المتعلقة بالتصميم. وتجدر الإشارة إلى أن هذه التمارين ذات نهايات مفتوحة، أي سنطرح عليك فقط بعض الأهداف البسيطة التي يتعين عليك الوصول إليها، ولكنك تستطيع بالطبع المُضي قدمًا إذا أردت المزيد من التحدي.

والآن، سنبدأ بالنسخة الجديدة من المُفهرِس.

14.1 قاعدة بيانات Redis

تُخزّن النسخة السابقة من المُفهرِس البيانات في هيكليّ بياناتٍ: الأول هو كائنٌ من النوع TermCounter يربط كل كلمة بحثٍ بعدد المرات التي ظهرت فيها الكلمة في صفحة إنترنت معينة، والثاني كائنٌ من النوع Index يربط كلمة البحث بمجموعة الصفحات التي ظهرت فيها.

يُخزّن هيكلا البيانات في ذاكرة التطبيق، ولذا يتلاشيان بمجرد انتهاء البرنامج. توصف البيانات التي تُخزّن فقط في ذاكرة التطبيق بأنها "متطايرة volatile"؛ لأنها تزول بمجرد انتهاء البرنامج.

في المقابل، تُوصف البيانات التي تظل موجودةً بعد انتهاء البرنامج الذي أنشأها بأنها "مستمرة persistent". مثال ذلك الملفات المُخزّنة في نظام الملفات فهي مُستمرة في العموم، وكذلك البيانات المُخزّنة في قاعدة بيانات أيضًا مستمرة.

يُعدّ تخزين البيانات في ملف واحدة من أبسط طرق حفظ البيانات، فيمكننا ببساطة أن نحوّل هياكل البيانات المُتضمّنة للبيانات إلى صيغة JSON ثم نكتبها إلى ملف قبل انتهاء البرنامج. عندما تُشغّل البرنامج مرة أخرى، سنتمكّن من قراءة الملف وإعادة بناء هياكل البيانات.

ولكن هناك مشكلتان في هذا الحل:

1. عادةً ما تكون عمليتا قراءة هياكل البيانات الضخمة (مثل مفهرس الويب) وكتابتها عمليتين بطيئتين.
 2. قد لا تستوعب مساحة ذاكرة برنامج واحد هيكلا البيانات بأكمله.
 3. إذا انتهى برنامجٌ معينٌ على نحوٍ غير متوقع (نتيجة لانقطاع الكهرباء مثلاً)، سنفقد جميع التغييرات التي أجريناها على البيانات منذ آخر مرة فتحنا فيها البرنامج.
- وهناك طريقة أخرى لحفظ البيانات وهي قواعد البيانات. إذ تُعدّ قواعد البيانات البديل الأفضل، فهي تُوفّر مساحةً تخزينٍ مستمرةً، كما أنها قادرةٌ على قراءة جزءٍ من قاعدة البيانات أو كتابته دون الحاجة إلى قراءة قاعدة البيانات أو كتابتها بالكامل.

تتوفّر الكثير من أنواع نظم إدارة قواعد البيانات DBMS، ويتمتع كلّ منها بإمكانيات مختلفة. ويُمكنك قراءة مقارنة بين أنظمة إدارة قواعد البيانات العلاقية والاطلاع على سلسلة تصميم قواعد البيانات.

تُوفّر قاعدة بيانات Redis -التي ستُستخدمها في هذا التمرين- هياكل بيانات مستمرة مشابهةً لهياكل البيانات التي تُوفّرها جافا، فهي تُوفّر:

- قائمة سلاسل نصية مشابهة للنوع List.
- جداول مشابهة للنوع Map.

- مجموعات من السلاسل النصية مشابهة للنوع Set.

تُعدّ Redis قاعدة بيانات من نوع زوج مفتاح/قيمة، ويعني ذلك أن هياكل البيانات (القيم) التي تُخزّن لها تكون مُعرّفة بواسطة سلاسل نصية فريدة (مفاتيح). تلعب المفاتيح في قاعدة بيانات Redis نفس الدور الذي تلعبه المراجع references في لغة جافا، أي أنّها تُعرّف هوية الكائنات. سنرى أمثلة على ذلك لاحقاً.

14.2 خادوم وعملاء Redis

عادةً ما تعمل قاعدة بيانات Redis كخدمة عن بعد، فكلمة Redis هي في الحقيقة اختصار لعبارة "خادم قاموس عن بعد REMote DIctionary Server"، ولنتمكن من استخدامها، علينا أن نُشغّل خادم Redis في مكان ما ثم نتصل به عبر عميل Redis. من الممكن إعداد ذلك الخادم بأكثر من طريقة، كما يُمكننا الاختيار من بين العديد من برامج العملاء، وسنستخدم في هذا التمرين ما يلي:

1. بدلاً من أن نُثبّت الخادم ونُشغّله بأنفسنا، سنستخدم خدمة مثل RedisToGo. تُشغّل تلك الخدمة قاعدة بيانات Redis على السحابة cloud، وتُقدّم خطة مجانية بموارد تتناسب مع متطلبات هذا التمرين.

2. بالنسبة للعميل، سنستخدم Jedis، وهو عبارة عن مكتبة جافا تحتوي على أصناف وتوابع يُمكنها العمل مع قاعدة بيانات Redis.

انظر إلى التعليمات المُفصّلة التالية لمساعدتك على البدء:

- أنشئ حساباً على موقع RedisToGo، واختر الخطة التي تريدها (ربما الخطة المجانية).
- أنشئ نسخة آلة افتراضية يعمل عليها خادم Redis. إذا نقرت على تبويب "Instances"، ستجد أن النسخة الجديدة مُعرّفة باسم استضافة ورقم منفذ. كان اسم النسخة الخاصة بنا مثلاً هو dory-10534.
- انقر على اسم النسخة لكي تفتح صفحة الإعدادات، وسجّل اسم مُحدّد الموارد الموحد URL الموجود أعلى الصفحة. سيكون مشابهاً لما يلي:

```
redis://redistogo:1234567feedfacebeefa1e1234567@dory.redistogo.com:10534
```

يحتوي محدد الموارد الموحد السابق ذكره على اسم الاستضافة الخاص بالخادم dory.redistogo.com، ورقم المنفذ 10534، وكلمة المرور التي سنحتاج إليها للاتصال بالخادم، وهي السلسلة النصية الطويلة المُكوّنة من أحرف وأعداد في المنتصف. ستحتاج تلك المعلومات في الخطوة التالية.

14.3 إنشاء مفهرس يعتمد على Redis

ستجد الملفات التالية الخاصة بالتمرين في مستودع الكتاب:

- `JedisMaker.java`: يحتوي على بعض الأمثلة على الاتصال بخادم Redis وتشغيل بعض توابع `Jedis`.

- `JedisIndex.java`: يحتوي على شيفرة مبدئية لهذا التمرين.

- `JedisIndexTest.java`: يحتوي على اختبارات للصنف `JedisIndex`.

- `WikiFetcher.java`: يحتوي على شيفرة تقرأ صفحات إنترنت وتُحلّلها باستخدام مكتبة `jsoup`. كتبنا تلك الشيفرة في تمارين الفصول المشار إليها بالأعلى.

ستجد أيضًا الملفات التالية التي كتبناها في نفس تلك التمارين:

- `Index.java`: يُنفذ فهرسًا باستخدام هياكل بيانات تُوقّرها جافا.

- `TermCounter.java`: يُمثل خريطة تربط كلمات البحث بعدد مرات حدوثها.

- `WikiNodeIterable.java`: يمرّ عبر عقد شجرة DOM الناتجة من مكتبة `jsoup`.

إذا تمكّنت من كتابة نسخك الخاصة من تلك الملفات، يُمكنك استخدامها لهذا التمرين. إذا لم تكن قد أكملت تلك التمارين أو أكملتها ولكنك غير متأكد مما إذا كانت تعمل على النحو الصحيح، يُمكنك أن تَنسخ الحلول من مجلد `solutions`.

والآن، ستكون خطوتك الأولى هي استخدام عميل `Jedis` للاتصال بخادم `Redis` الخاص بك. يُوضّح الصنف `RedisMaker.java` طريقة القيام بذلك: عليه أولاً أن يقرأ معلومات الخادم من ملف، ثم يتصل به، ويُسجل دخوله باستخدام كلمة المرور، وأخيرًا، يُعيد كائنًا من النوع `Jedis` الذي يُمكن استخدامه لتنفيذ عمليات `Redis`.

ستجد الصنف `JedisMaker` مُعرّفًا في الملف `JedisMaker.java`. يعمل ذلك الصنف كصنف مساعد حيث يحتوي على التابع الساكن `make` الذي يُنشئ كائنًا من النوع `Jedis`. يُمكنك أن تستخدم ذلك الكائن بعد التصديق عليه للاتصال بقاعدة بيانات `Redis` الخاصة بك.

يقرأ الصنف `JedisMaker` بيانات خادم `Redis` من ملف اسمه `redis_url.txt` موجود في المجلد `:src/resources`

- استخدم مُحرّر نصوص لإنشاء وتعديل الملف التالي:

`ThinkDataStructures/code/src/resources/redis_url.txt`.

- ضع فيه مُحدّد موارد الخادم الخاص بك. إذا كنت تستخدم خدمة `RedisToGo`، سيكون محدد الموارد مشابهًا لما يلي:


```
redis://
redistogo:1234567feedfacebeefa1e1234567@dory.redistogo.com:10534
```

لا تضع هذا الملف في مجلدٍ عامٍّ لأنه يحتوي على كلمة مرور خادم Redis. يُمكنك تجنُّب وقوع ذلك عن طريق الخطأ باستخدام الملف gitignore. الموجود في مستودع الكتاب لمنع وضع الملف فيه.

نقِّذ الأمر `ant build` لتصريف ملفات الشيفرة والأمر `ant JedisMaker` لتشغيل المثال التوضيحي

بالتابع `:main`

```
public static void main(String[] args) {
    Jedis jedis = make();

    // String
    jedis.set("mykey", "myvalue");
    String value = jedis.get("mykey");
    System.out.println("Got value: " + value);

    // Set
    jedis.sadd("myset", "element1", "element2", "element3");
    System.out.println("element2 is member: " +
        jedis.sismember("myset", "element2"));

    // List
    jedis.rpush("mylist", "element1", "element2", "element3");
    System.out.println("element at index 1: " +
        jedis.lindex("mylist", 1));

    // Hash
    jedis.hset("myhash", "word1", Integer.toString(2));
    jedis.hincrBy("myhash", "word2", 1);
    System.out.println("frequency of word1: " +
        jedis.hget("myhash", "word1"));
    System.out.println("frequency of word1: " +
        jedis.hget("myhash", "word2"));

    jedis.close();
}
```

يَعْرِضُ المِثَالُ أنواعَ البيانات والتوابع التي ستحتاج إليها غالبًا في هذا التمرين. ينبغي أن تحصل على الخرج التالي عند تشغيله:

```
Got value: myvalue
element2 is member: true
element at index 1: element2
frequency of word1: 2
frequency of word2: 1
```

سنشرح طريقة عمل تلك الشيفرة في القسم التالي.

14.4 أنواع البيانات في قاعدة بيانات Redis

تَعْمَلُ Redis كخريطة تربط مفاتيح (سلاسل نصية) بقيم. قد تنتمي تلك القيم إلى مجموعة أنواع مختلفة من البيانات. يُعَدُّ النوع string واحدًا من أبسط الأنواع التي تُوفِّرها قاعدة بيانات Redis. لاحظ أننا سنكتب أنواع بيانات Redis بخطوط مائلة لتمييزها عن أنواع جافا.

سَنَسْتَخِدمُ التابع `jedis.set` لإضافة سلسلة نصية من النوع string إلى قاعدة البيانات. قد تجد ذلك مألوفاً، فهو يشبه التابع `Map.put`، حيث تُمَثِّلُ المعاملات المُمرَّرة المفتاح الجديد وقيمه المقابلة. في المقابل، يُسْتَخْدَمُ التابع `jedis.get` للبحث عن مفتاح معين والعثور على قيمته. انظر الشيفرة إلى التالية:

```
jedis.set("mykey", "myvalue");
String value = jedis.get("mykey");
```

كان المفتاح هو "mykey" والقيمة هي "myvalue" في هذا المثال.

تُوفِّرُ Redis هيكل البيانات `set` الذي يَعْمَلُ بشكلٍ مشابهٍ للنوع `Set<String>` في جافا. إذا أردت أن تضيف عنصراً جديداً إلى مجموعة من النوع `set`، اختر مفتاحاً يُحدِّد هوية تلك المجموعة، ثم اِسْتَخْدِمِ التابع `jedis.sadd` كما يلي:

```
jedis.sadd("myset", "element1", "element2", "element3");
boolean flag = jedis.sismember("myset", "element2");
```

لاحظ أنه ليس من الضروري إنشاء المجموعة بخطوة منفصلة، حيث تُنشأها Redis إن لم تكن موجودة. تُنشَأُ Redis في هذا المثال مجموعة من النوع `set` اسمها `myset` تحتوي على ثلاثة عناصر.

يفحص التابع `jedis.sismember` ما إذا كان عنصر معين موجوداً في مجموعة من النوع `set`. تستغرق عمليتا إضافة العناصر وفحص عضويتها زمناً ثابتاً.

تُوفّر Redis أيضًا هيكل البيانات `list` الذي يشبه النوع `List<String>` في جافا. يضيف التابع `jedis.rpush` العناصر إلى النهاية اليمنى من القائمة، كما يلي:

```
jedis.rpush("mylist", "element1", "element2", "element3");
String element = jedis.lindex("mylist", 1);
```

مثلما سبق، لا يُعدّ إنشاء هيكل البيانات قبل إضافة العناصر إليه أمرًا ضروريًا. يُنشئ هذا المثال قائمةً من النوع `list` اسمها `mylist` تحتوي على ثلاثة عناصر.

يُستقبل التابع `jedis.lindex` فهرسًا هو عبارة عن عددٍ صحيح، ويعيد عنصرَ القائمة المشار إليه. تستغرق عمليتا إضافة العناصر واسترجاعها زمنًا ثابتًا.

أخيرًا، تُوفّر Redis الهيكل `hash` الذي يشبه النوع `Map<String, String>` في جافا. يضيف التابع `jedis.hset` مُدخلاً جديدًا إلى الجدول على النحو التالي:

```
jedis.hset("myhash", "word1", Integer.toString(2));
String value = jedis.hget("myhash", "word1");
```

يُنشئ هذا المثال جدولًا جديدًا اسمه `myhash` يحتوي على مدخلٍ واحدٍ يربط المفتاح `word1` بالقيمة "2". تنتمي المفاتيح والقيم إلى النوع `string`، ولذلك، إذا أردنا أن نُخزّن عددًا صحيحًا من النوع `Integer`، علينا أن نُحوّله أولًا إلى النوع `String` قبل أن نُسديّ التابع `hset`. وبالمثل، عندما نبحث عن قيمة باستخدام التابع `hget`، ستكون النتيجة من النوع `String`، ولذلك، قد نضطرّ إلى تحويلها مرةً أخرى إلى النوع `Integer`.

قد يكون العمل مع النوع `hash` في قاعدة بيانات Redis مربكًا نوعًا ما؛ لأننا نستخدم مفاتيحين، الأول لتحديد الجدول الذي نريده، والثاني لتحديد القيمة التي نريدها من الجدول. في سياق قاعدة بيانات Redis، يُطلق على المفتاح الثاني اسم "الحقل `field`"، أي يشير مفتاح مثل `myhash` إلى جدولٍ معينٍ من النوع `hash` بينما يشير حقلٌ مثل `word1` إلى قيمةٍ ضمن ذلك الجدول.

عادةً ما تكون القيم المُخزّنة في جداول من النوع `hash` أعدادًا صحيحة، ولذلك، يوفّر نظام إدارة قاعدة بيانات Redis بعض التوابع الخاصة التي تعامل القيم وكأنها أعداد مثل التابع `hincrby`. انظر إلى المثال التالي:

```
jedis.hincrBy("myhash", "word2", 1);
```

يسترجع هذا التابع المفتاح `myhash`، ويحصل على القيمة الحالية المرتبطة بالحقل `word2` (أو على الصفر إذا لم يكن الحقل موجودًا)، ثم يزيدها بمقدار 1، ويكتبها مرةً أخرى في الجدول.

تستغرق عمليات إضافة المُدخّلات إلى جدول من النوع `hash` واسترجاعها وزيادتها زمنًا ثابتًا.

14.5 تمرين 11

بوصولك إلى هنا، أصبح لديك كل المعلومات الضرورية لإنشاء مُفهرسٍ قادرٍ على تخزين النتائج في قاعدة بيانات Redis.

والآن، نَقِّذ الأمر `JedisIndexTest` ستفشل بعض الاختبارات لأنه ما يزال أماننا بعض العمل.

يختبر الصنف `JedisIndexTest` التوابع التالية:

- `JedisIndex`: يستقبل هذا الباني كائنًا من النوع `Jedis` كمعامل.
- `indexPath`: يضيف صفحة إنترنت إلى المفهرس. يَسْتَقْبِل سلسلة نصيَّة من النوع `String` تُمثِّل مُحدَّد موارد URL بالإضافة إلى كائن من النوع `Elements` يحتوي على عناصر الصفحة المطلوب فهرستها.
- `getCounts`: يستقبل كلمة بحثٍ ويعيد خريطةً من النوع `Map<String, Integer>` تربُّط كل محدد مواردٍ يحتوي على تلك الكلمة بعدد مرات ظهورها في تلك الصفحة.

يُوضِّح المثال التالي طريقة استخدام تلك التوابع:

```
WikiFetcher wf = new WikiFetcher();
String url1 =
    "http://en.wikipedia.org/wiki/Java_(programming_language)";
Elements paragraphs = wf.readWikipedia(url1);

Jedis jedis = JedisMaker.make();
JedisIndex index = new JedisIndex(jedis);
index(indexPath(url1, paragraphs);
Map<String, Integer> map = index.getCounts("the");
```

إذا بحثنا عن `url1` في الخريطة الناتجة `map`، ينبغي أن نحصل على 339، وهو عدد مرات ظهور كلمة "the" في مقالة ويكيبيديا عن لغة جافا (نسخة المقالة التي خزناها).

إذا حاولنا فهرسة الصفحة مرَّةً أخرى، ستحلَّ النتائج الجديدة محل النتائج القديمة.

إذا أردت تحويل هياكل البيانات من جافا إلى Redis، فتذكَّر أن كل كائنٍ مُخزَّنٍ في قاعدة بيانات Redis مُعرَّفٌ بواسطة مفتاحٍ فريدٍ من النوع `string`. إذا كان لديك نوعان من الكائنات في نفس قاعدة البيانات، فقد ترغب في إضافة كلمةٍ إلى بداية المفاتيح لتمييزها عن بعضها. على سبيل المثال، لدينا النوعان التاليان من الكائنات:

- يُمثل النوع `URLSet` مجموعةً من النوع `set` في قاعدة بيانات Redis. تحتوي تلك المجموعة على محددات الموارد الموحدة URL التي تحتوي على كلمة بحثٍ معينة. يبدأ المفتاح الخاص بكل قيمةٍ من النوع `URLSet` بكلمة "URLSet:"، وبالتالي، لكي نحصل على محددات الموارد الموحدة التي تحتوي على كلمة "the"، علينا أن نسترجع المجموعة التي مفتاحها هو "URLSet: the".

- يُمثل النوع `TermCounter` جدولاً من النوع `hash` في قاعدة بيانات Redis. يربط هذا الجدول كل كلمة بحثٍ ظهرت في صفحةٍ معيّنة بعدد مرات ظهورها. يبدأ المفتاح الخاص بكل قيمةٍ من النوع `TermCounter` بكلمة "TermCounter:" وينتهي بمحدد الموارد الموحّد الخاص بالصفحة التي نبحث فيها.

يحتوي التنفيذ الخاص بنا على قيمةٍ من النوع `URLSet` لكل كلمة بحثٍ، وعلى قيمةٍ من النوع `TermCounter` لكل صفحةٍ مُفهرسة. وقرنا أيضاً التابعين المساعدين `urlSetKey` و `termCounterKey` لإنشاء تلك المفاتيح.

14.6 المزيد من الاقتراحات

أصبح لديك الآن كل المعلومات الضرورية لحل التمرين، لذا يُمكنك أن تبدأ الآن إذا أردت، ولكن ما يزال هناك بعض الاقتراحات القليلة التي ننصحك بقراءتها:

- سنوفّر لك مساعدةً أقلّ في هذا التمرين، ونترك لك حرية أكبر في اتخاذ بعض القرارات المتعلقة بالتصميم. سيكون عليك تحديداً أن تُفكّر بالطريقة التي ستُقسّم بها المشكلة إلى أجزاءٍ صغيرةٍ يُمكن اختبار كلٍّ منها على حدة. بعد ذلك، ستُجمّع تلك الأجزاء إلى حلٍّ كاملٍ. إذا حاولت كتابة الحل بالكامل على خطوةٍ واحدةٍ بدون اختبار الأجزاء الأصغر، فقد تستغرق وقتاً طويلاً جداً لتنقيح الأخطاء.
- تُمثّل الاستمرارية واحدةً من تحديات العمل مع البيانات المستمرة، لأن الهياكل المُخزّنة في قواعد البيانات قد تتغير في كل مرة تُشغّل فيها البرنامج. فإذا تسبّبت بخطأ في قاعدة البيانات، سيكون عليك إصلاحه أو البدء من جديد. ولكي نُبقي الأمور تحت السيطرة، وقرنا لك التوابع `deleteURLSets` و `deleteTermCounters` التي تستطيع أن تستخدمها لتنظيف قاعدة البيانات والبدء من جديد. يُمكنك أيضاً استخدام التابع `printIndex` لطباعة محتويات المُفهرس.
- في كلّ مرةٍ تستدعي فيها أيّاً من توابع `Jedis`، فإنه يُرسل رسالةً إلى الخادم الذي يُنفّذ بدوره الأمر المطلوب، ثم يردّ برسالة. إذا نفّذت الكثير من العمليات الصغيرة، فستحتاج إلى وقت طويل لمعالجتها، ولهذا، من الأفضل تجميع متتالية من العمليات ضمن معاملة واحدة من النوع `Transaction` لتحسين الأداء.

على سبيل المثال، انظر إلى تلك النسخة البسيطة من التابع `deleteAllKeys`:

```
public void deleteAllKeys() {
    Set<String> keys = jedis.keys("*");
    for (String key: keys) {
        jedis.del(key);
    }
}
```

في كل مرة تُستدعى خلالها التابع `del`، يضطرّ العميل إلى إجراء اتصالٍ مع الخادم وانتظار الرد. إذا كان المُفهرِس يحتوي على بضع صفحاتٍ، فقد يستغرق تنفيذ ذلك التابع وقتًا طويلًا. بدلًا من ذلك، يُمكنك أن تُسرّع تلك العملية باستخدام كائنٍ من النوع `Transaction` على النحو التالي:

```
public void deleteAllKeys() {
    Set<String> keys = jedis.keys("*");
    Transaction t = jedis.multi();
    for (String key: keys) {
        t.del(key);
    }
    t.exec();
}
```

يعيد التابع `jedis.multi` كائنًا من النوع `Transaction`. يُوفّر هذا الكائن جميع التوابع المتاحة في كائنات النوع `Jedis`. عندما تستدعي أيًا من تلك التوابع بكائنٍ من النوع `Transaction`، فإن العميل لا يُنفّذها تلقائيًا، ولا يتصل مع الخادم، وإنما يُخزّن تلك العمليات إلى أن تستدعي التابع `exec`، وعندها، يُرسل جميع العمليات المُخزّنة إلى الخادم في نفس الوقت، وهو ما يكون أسرع عادةً.

14.7 تلميحات بسيطة بشأن التصميم

الآن وقد أصبح لديك جميع المعلومات المطلوبة، يمكنك البدء في حل التمرين. إذا لم يكن لديك فكرة عن طريقة البدء، يُمكنك العودة لقراءة المزيد من التلميحات البسيطة.

لا تتابع القراءة قبل أن تُشغّل شيفرة الاختبار، وتُجرّب بعض أوامر Redis البسيطة، وتكتب بعض التوابع الموجودة في الملف `JedisIndex.java`.

إذا لم تتمكن من متابعة الحل فعلًا، إليك بعض التوابع التي قد ترغب في العمل عليها:

```
/**
 * أضف محدد موارد موحّدًا إلى المجموعة الخاصة بكلمة *
 */
```

```

public void add(String term, TermCounter tc) {}

/**
 * ابحث عن كلمة وأعد مجموعة مُحدّدت الموارد الموحدة التي تحتوي عليها
 */

public Set<String> getURLs(String term) {}

/**
 * أعد عدد مرات ظهور كلمة معينة بمحدد موارد موحد
 */

public Integer getCount(String url, String term) {}

/**
 * أضف محتويات كائن من النوع `TermCounter` إلى قاعدة بيانات Redis
 */

public List<Object> pushTermCounterToRedis(TermCounter tc) {}

```

تلك هي التوابع التي استخدمناها في الحل، ولكنها ليست بالتأكيد الطريقة الوحيدة لتقسيم المشكلة. لذلك، استعن بتلك الاقتراحات فقط إذا وجدتتها مفيدة، وتجاهلها تمامًا إذا لم تجدها كذلك.

اكتب بعض الاختبارات لكل تابعٍ منها أولاً، فبمعرفة الطريقة التي ينبغي بها أن تختبر تابعًا معيّنًا، عادةً ما تتكوّن لديك فكرة عن طريقة كتابته.

وفقك الله!



ادخل سوق العمل ونفذ المشاريع باحترافية
عبر أكبر منصة عمل حر بالعالم العربي

ابدأ الآن كمستقل

15. الزحف على ويكيبيديا

سنقدّم في هذا الفصل حل تمرين الفصل السابق. بعد ذلك، سنحلّل أداء خوارزمية فهرسة صفحات الإنترنت، ثم سنبنّي زاحف ويب Web crawler بسيطًا.

15.1 الم فهرس المبني على قاعدة بيانات Redis

سنخزّن هيكلي البيانات التاليين في قاعدة بيانات Redis:

- سيُقابل كل كلمة بحث كائن من النوع `URLSet` هو عبارة عن مجموعة `Set` في قاعدة بيانات Redis تحتوي على مُحدّدات الموارد الموحدة `URLs` التي تحتوي على تلك الكلمة.
 - سيُقابل كل مُحدّد موارد كائنًا من النوع `TermCounter` يُمثّل جدول `Hash` في قاعدة بيانات Redis يربط كل كلمة بحث بعدد مرات ظهورها.
- يُمكنك مراجعة أنواع البيانات التي ناقشناها في الفصل المشار إليه، كما يُمكنك قراءة المزيد عن المجموعات والجداول بقاعدة بيانات Redis من توثيقها الرسمي.
- يُعرّف الصنف `JedisIndex` تابعًا يستقبل كلمة بحث ويعيد مفتاح كائن الصنف `URLSet` المقابل في قاعدة بيانات Redis:

```
private String urlSetKey(String term) {  
    return "URLSet:" + term;  
}
```

كما يُعرّف الصنف المذكور التابع `termCounterKey`، والذي يستقبل مُحدّد موارد موحدًا ويعيد مفتاح كائن الصنف `TermCounter` المقابل في قاعدة بيانات Redis:

```
private String termCounterKey(String url) {
    return "TermCounter:" + url;
}
```

يُستقبل تابع الفهرسة `indexPath` محدّدَ مواردٍ موحّدًا وكائنًا من النوع `Elements` يحتوي على شجرة DOM للفقرات التي نريد فهرستها:

```
public void indexPath(String url, Elements paragraphs) {
    System.out.println("Indexing " + url);

    // أنشئ كائنًا من النوع TermCounter وأحصِ كلمات الفقرات النصية

    TermCounter tc = new TermCounter(url);
    tc.processElements(paragraphs);

    // أضف محتويات الكائن إلى قاعدة بيانات Redis
    pushTermCounterToRedis(tc);
}
```

سنقوم بالتالي لكي نُفهرّس الصفحة:

1. نُنشئ كائنًا من النوع `TermCounter` يُمثّل محتويات الصفحة باستخدام شيفرة تمرين الفصل المشار إليه بالأعلى.

2. نُضيف محتويات ذلك الكائن في قاعدة بيانات `Redis`.

توضّح الشيفرة التالية طريقة إضافة كائنات النوع `TermCounter` إلى قاعدة بيانات `Redis`:

```
public List<Object> pushTermCounterToRedis(TermCounter tc) {
    Transaction t = jedis.multi();

    String url = tc.getLabel();
    String hashname = termCounterKey(url);

    // إذا كانت الصفحة مفهرسة مسبقًا، احذف الجدول القديم
    t.del(hashname);

    // أضف مدخلًا جديدًا في كائن الصنف TermCounter وعنصرًا جديدًا إلى الفهرس
```

```
// لكل كلمة بحث
for (String term: tc.keySet()) {
    Integer count = tc.get(term);
    t.hset(hashname, term, count.toString());
    t.sadd(urlSetKey(term), url);
}
List<Object> res = t.exec();
return res;
}
```

يستخدم هذا التابع معاملةً من النوع `Transaction` لتجميع العمليات، ثم يرسلها جميعًا إلى الخادم على خطوة واحدة. تُعدّ تلك الطريقة أسرع بكثير من إرسال متتاليةٍ من العمليات الصغيرة.

يمرّ التابع عبر العناصر الموجودة في كائن الصنف `TermCounter`، ويُنفَّذ التالي من أجل كل عنصرٍ منها:

1. يبحث عن كائنٍ من النوع `TermCounter` -أو ينشئه إن لم يجده- في قاعدة بيانات `Redis`، ثم يضيف حقلاً فيه يُمثل العنصر الجديد.

2. يبحث عن كائنٍ من النوع `URLSet` -أو ينشئه إن لم يجده- في قاعدة بيانات `Redis`، ثم يضيف إليه محدّد الموارد الموحد الحالي.

إذا كنا قد فهرّسنا تلك الصفحة من قبل، علينا أن نحذف كائن الصنف `TermCounter` القديم الذي يمثلها قبل أن نضيف المحتويات الجديدة.

هذا هو كل ما نحتاج إليه لفهرسة الصفحات الجديدة.

طلبَ الجزء الثاني من التمرين كتابةً التابع `getCounts` الذي يَستقبل كلمة بحثٍ ويعيد خريطةً تربط محدّدات الموارد الموحدة التي ظهرت فيها تلك الكلمة بعدد مرات ظهورها فيها. انظر إلى تنفيذ التابع:

```
public Map<String, Integer> getCounts(String term) {
    Map<String, Integer> map = new HashMap<String, Integer>();
    Set<String> urls = getURLs(term);
    for (String url: urls) {
        Integer count = getCount(url, term);
        map.put(url, count);
    }
    return map;
}
```

يستخدم هذا التابعُ تابعين مساعدين:

- `getURLs`: يَستَقبِل كلمة بحثٍ ويعيد مجموعةً من النوع `Set` تحتوي على محدّدات الموارد الموحدة التي ظهرت فيها الكلمة.
- `getCount`: يَستَقبِل محدّد موارد موحّدًا `URI` وكلمة بحث، ويعيد عدد مرات ظهور الكلمة بمحدّد الموارد المُمرَّر.

انظر تنفيذات تلك التوابع:

```
public Set<String> getURLs(String term) {
    Set<String> set = jedis.smembers(urlSetKey(term));
    return set;
}

public Integer getCount(String url, String term) {
    String redisKey = termCounterKey(url);
    String count = jedis.hget(redisKey, term);
    return new Integer(count);
}
```

تَعمَل تلك التوابع بكفاءةٍ نتيجةً للطريقة التي صمّمنا بها المُفهرِس.

15.2 تحليل أداء عملية البحث

لنفترض أننا فهرسنا عددًا مقداره N من الصفحات، وتوصلنا إلى عددٍ مقداره M من كلمات البحث. كم الوقت الذي سيستغرقه البحث عن كلمةٍ معينة؟ فكر قبل أن تكمل القراءة. سنُنقِّذ التابع `getCounts` للبحث عن كلمةٍ، يُنقِّذ ذلك التابع ما يلي:

1. يُنشئ خريطةً من النوع `HashMap`.
2. يُنقِّذ التابع `getURLs` ليسترجع مجموعة مُحدّدات الموارد الموحدة.
3. يَستدعي التابع `getCount` لكل مُحدّد موارد، ويضيف مُدخلاً إلى الخريطة.

يستغرق التابع `getURLs` زمناً يتناسب مع عدد محدّدات الموارد الموحدة التي تحتوي على كلمة البحث. قد يكون عددًا صغيرًا بالنسبة للكلمات النادرة، ولكنه قد يكون كبيرًا -قد يصل إلى N - في حالة الكلمات الشائعة. سنُنقِّذ داخل الحلقة التابع `getCount` الذي يبحث عن كائني من النوع `TermCounter` في قاعدة بيانات `Redis`، ثم يبحث عن كلمةٍ، ويضيف مُدخلاً إلى خريطةٍ من النوع `HashMap`. تستغرق جميع تلك العمليات زمناً

ثابتًا، وبالتالي، ينتمي التابع `getCounts` في المجلد إلى المجموعة $O(N)$ في أسوأ الحالات، ولكن عمليًا، يتناسب زمن تنفيذه مع عدد الصفحات التي تحتوي على تلك الكلمة، وهو عادةً عدد أصغر بكثير من N .

وأما فيما يتعلق بتحليل الخوارزميات، فإن تلك الخوارزمية تتميز بأقصى قدر من الكفاءة، ولكنها مع ذلك بطيئة لأنها ترسل الكثير من العمليات الصغيرة إلى قاعدة بيانات Redis. من الممكن تحسين أدائها باستخدام معاملة من النوع `Transaction`. يُمكنك محاولة تنفيذ ذلك أو الاطلاع على الحل في الملف `RedisIndex.java` (انتبه إلى أن اسمه في المستودع `JedisIndex.java` والله أعلم).

15.3 تحليل أداء عملية الفهرسة

ما الزمن الذي تستغرقه فهرسة صفحة عند استخدام هياكل البيانات التي صممناها؟ فكر قبل أن تكمل القراءة.

لكي نُفهرّس صفحةً، فإننا نمرّ عبر شجرة DOM، ونعثر على الكائنات التي تنتمي إلى النوع `TextNode`، ونُقسّم السلاسل النصية إلى كلمات بحث. تستغرق كل تلك العمليات زمنًا يتناسب مع عدد الكلمات الموجودة في الصفحة.

نزيد العداد ضمن خريطة النوع `HashMap` لكل كلمة بحث ضمن الصفحة، وهو ما يستغرق زمنًا ثابتًا، ما يجعل الصنف `TermCounter` يستغرق زمنًا يتناسب مع عدد الكلمات الموجودة في الصفحة.

تتطلب إضافة كائن الصنف `TermCounter` إلى قاعدة بيانات Redis حذف كائن آخر منها. يستغرق ذلك زمنًا يتناسب خطيًا مع عدد كلمات البحث. بعد ذلك، علينا أن نُنفذ التالي من أجل كل كلمة:

1. نضيف عنصرًا إلى كائن من النوع `URLSet`.

2. نضيف عنصرًا إلى كائن من النوع `TermCounter`.

تستغرق العمليتان السابقتان زمنًا ثابتًا، وبالتالي، يكون الزمن الكلي المطلوب لإضافة كائن من النوع `TermCounter` خطيًا مع عدد كلمات البحث الفريدة.

نستخلص مما سبق أنّ زمن تنفيذ الصنف `TermCounter` يتناسب مع عدد الكلمات الموجودة في الصفحة، وأن إضافة كائن ينتمي إلى النوع `TermCounter` إلى قاعدة بيانات Redis تتطلب زمنًا يتناسب مع عدد الكلمات الفريدة.

ولما كان عدد الكلمات الموجودة في الصفحة يتجاوز عادةً عدد كلمات البحث الفريدة، فإن التعقيد يتناسب طرديًا مع عدد الكلمات الموجودة في الصفحة، ومع ذلك، قد تحتوي صفحة ما نظريًا على جميع كلمات البحث الموجودة في الفهرس، وعليه، ينتمي الأداء في أسوأ الحالات إلى المجموعة $O(M)$ ، ولكننا لا نتوقع حدوث تلك الحالة عمليًا.

يتّضح من التحليل السابق أنه ما يزال من الممكن تحسين أداء الخوارزمية، فمثلاً يُمكننا أن نتجنّب فهرسة الكلمات الشائعة جدًّا، لأنها أولاً تحتل مساحةً كبيرةً من الذاكرة كما أنها تستغرق وقتًا طويلًا؛ فهي تظهر تقريبًا في جميع كائنات النوعين URLSet و TermCounter، كما أنها لا تحمل أهميةً كبيرةً فهي لا تساعد على تحديد الصفحات التي يُحتمل أن تكون ذات صلةٍ بكلمات البحث.

تجنّب غالبية محركات البحث فهرسةً الكلمات الشائعة التي يطلق عليها اسم الكلمات المهملة stop words ضمن هذا السياق.

15.4 التنقل في مخطط graph

إذا أكملت تمرين الفصل السابع كل الطرق تؤدي إلى روما، فلديك بالفعل برنامجٌ يقرأ صفحةً من موقع ويكيبيديا، ويبحث عن أول رابطٍ فيها، ويستخدمه لتحميل الصفحة التالية، ثم يكرر العملية. يُعدّ هذا البرنامج بمنزلةٍ زاحفٍ من نوع خاص، فعادةً عندما يُذكر مصطلح "زاحف إنترنت" Web crawler، فالمقصود برنامج يُنفَّذ ما يلي:

- يُحمّل صفحة بداية معينة، ويُفهرس محتوياتها.
 - يبحث عن جميع الروابط الموجودة في تلك الصفحة ويضيفها إلى تجميعية collection.
 - يمرّ عبر تلك الروابط، ويُحمّل كلّ منها، ويُفهرسه، ويضيف أثناء ذلك روابط جديدة.
 - إذا عثر على مُحدّدٍ مواردٍ مُوحّدٍ فهرسه من قبل، فإنه يتخطاه.
- يُمكننا أن نتصوّر شبكة الإنترنت كما لو كانت شعبة أو مخطط graph. تُمثّل كل صفحة إنترنت عقدةً node في تلك الشعبة، ويُمثّل كل رابط ضلعًا مُوجّهًا directed edge من عقدةٍ إلى عقدةٍ أخرى. يُمكنك قراءة المزيد عن الشعب إذا لم تكن على معرفةٍ بها.
- بناءً على هذا التصور، يستطيع الزاحف أن يبدأ من عقدةٍ معيّنة، ويتنقل عبر أو يجتاز الشعبة، وذلك بزيارة العقد التي يُمكنه الوصول إليها مرةً واحدةً فقط.
- تُحدّد التجميعية التي سنستخدمها لتخزين محددات الموارد المحددة نوع الاجتياز أو التنقل الذي يُنفّذه الزاحف:

- إذا كانت التجميعية رتلاً queue، أي تتبع أسلوب "الداخل أولاً، يخرج أولاً" FIFO، فإن الزاحف يُنفّذ اجتياز السّعة أولاً breadth-first.
- إذا كانت التجميعية مكدّسا stack، أي تتبع أسلوب "الداخل آخرًا، يخرج أولاً" LIFO، فإن الزاحف يُنفّذ اجتياز العمق أولاً depth-first.

- من الممكن تحديد أولويات لعناصر التجميع. على سبيل المثال، قد نرغب في إعطاء أولوية أعلى للصفحات التي لم تُفهرَس منذ فترة طويلة.

15.5 تمرين 12

والآن، عليك كتابة الزاحف، ستجد ملفات الشيفرة التالية الخاصة بالتمرين في مستودع الكتاب:

- `WikiCrawler.java`: يحتوي على شيفرة مبدئية للزاحف.
 - `WikiCrawlerTest.java`: يحتوي على اختبارات وحدة للصنف `WikiCrawler`.
 - `JedisIndex.java`: يحتوي على حل تمرين الفصل المشار إليه بالأعلى.
- ستحتاج أيضًا إلى الأصناف المساعدة التالية التي استخدمناها في تمارين الفصول السابقة:

- `JedisMaker.java`

- `WikiFetcher.java`

- `TermCounter.java`

- `WikiNodeIterable.java`

سيُتعيّن عليك توفير ملفّ يحتوي على بيانات خادم Redis قبل تنفيذ الصنف `JedisMaker`. إذا أكملت تمرين الفصل المشار إليه بالأعلى، فقد جُهّزت كل شيء بالفعل، أما إذا لم تكمله، فستجد التعليمات الضرورية لإتمام ذلك في نفس الفصل.

ننّذ الأمر `ant build` لتصريف ملفات الشيفرة، ثم ننّذ الأمر `ant JedisMaker` لكي تتأكد من أنه مهياً للاتصال مع خادم Redis الخاص بك.

والآن، ننّذ الأمر `ant WikiCrawlerTest`. ستجد أن الاختبارات قد فشلت؛ لأن ما يزال عليك إتمام بعض العمل أولاً.

انظر إلى بداية تعريف الصنف `WikiCrawler`:

```
public class WikiCrawler {

    public final String source;
    private JedisIndex index;
    private Queue<String> queue = new LinkedList<String>();
    final static WikiFetcher wf = new WikiFetcher();
```

```

public WikiCrawler(String source, JedisIndex index) {
    this.source = source;
    this.index = index;
    queue.offer(source);
}

public int queueSize() {
    return queue.size();
}

```

يحتوي هذا الصنف على متغيرات النسخ instance variables التالية:

- **source**: مُحدّد الموارد الموحد الذي ستبدأ منه.
 - **index**: فهرس -من النوع JedisIndex- ينبغي أن تُخزّن النتائج فيه.
 - **queue**: عبارة عن قائمة من النوع LinkedList. يُفترض أن تُخزّن فيها كلُّ محدّدات الموارد التي عثرت عليها، ولكن لم تُفهرسها بعد.
 - **wf**: عبارة عن كائن من النوع WikiFetcher. عليك أن تستخدمه لقراءة صفحات الإنترنت وتحليلها.
- عليك الآن أن تكمل التابع `crawl`. انظر إلى بصمته:

```

public String crawl(boolean testing) throws IOException {}

```

ينبغي أن تكون قيمة المعامل `testing` مساويةً للقيمة `true` إذا كان مستدعيه هو الصنف `WikiCrawlerTest`، وأن تكون مساويةً للقيمة `false` في الحالات الأخرى.

عندما تكون قيمة المعامل `testing` مساويةً للقيمة `true`، يُفترض أن يُنفّذ التابع `crawl` ما يلي:

- يختار مُحدّدَ مواردٍ موحّدًا من الرتل وفقًا لأسلوب "الداخل أولاً، يخرج أولاً" ثم يحذفه منه.
- يقرأ محتويات تلك الصفحة باستخدام التابع `WikiFetcher.readWikipedia` الذي يعتمد في قراءته للصفحات على نسخٍ مُخزّنةٍ مؤقتًا في المستودع بهدف الاختبار (لكي نتجنّب أيّ مشكلات ممكنة في حال تغيّرت النسخُ الموجودةُ في موقع ويكيبيديا).
- يُفهرس الصفحة بغض النظر عما إذا كانت قد فُهرست من قبل.
- ينبغي أن يجد كل الروابط الداخلية الموجودة في الصفحة ويضيفها إلى الرتل بنفس ترتيب ظهورها. يُقصد بالروابط الداخلية تلك الروابط التي تشير إلى صفحات أخرى ضمن موقع ويكيبيديا.
- ينبغي أن يعيد مُحدّد الموارد الخاص بالصفحة التي فهرسها.

في المقابل، عندما تكون قيمة المعامل `testing` مساوية للقيمة `false`، يُفترض له أن يُنفَّذ ما يلي:

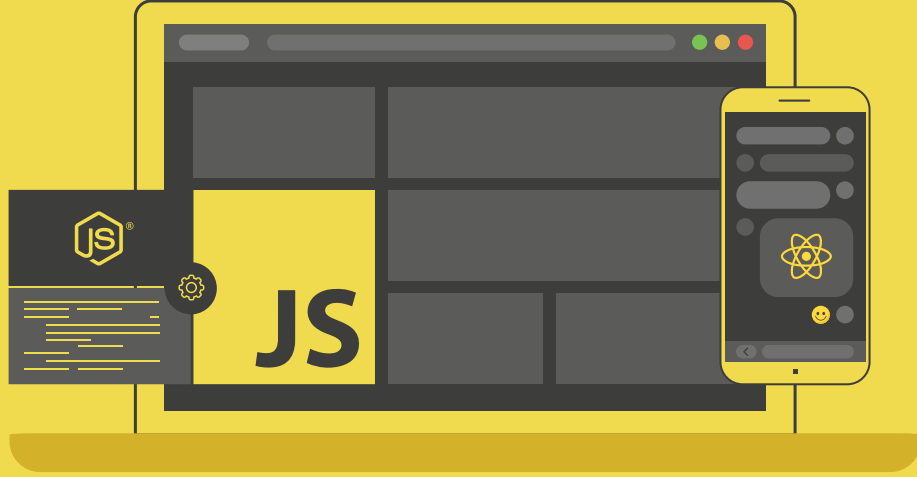
- يختار مُحدّد موارد موحّدًا من الرتل وفقًا لأسلوب "الداخل أولاً، يخرج أولاً" ثم يحذفه منه.
- إذا كان محدّد الموارد المختار مفهرسًا بالفعل، لا ينبغي أن يعيد فهرسته، وعليه أن يعيد القيمة `null`.
- إذا لم يُفهرس من قبل، فينبغي أن يقرأ محتويات الصفحة باستخدام التابع `WikiFetcher.fetchWikipedia` الذي يعتمد في قراءته للصفحات على شبكة الإنترنت.
- ينبغي أن يُفهرس الصفحة، ويضيف أيّ روابط موجودة فيها إلى الرتل، ويعيد مُحدّد الموارد الخاصّ بالصفحة التي فهرسها.

يُحمّل الصنف `WikiCrawlerTest` رتلًا يحتوي على 200 رابط، ثم يستدعي التابع `crawl` ثلاث مرّات، ويفحص في كلّ استدعاء القيمة المعادة والطول الجديد للرتل.

إذا أكملت زاحف الإنترنت الخاص بك بشكل صحيح، فينبغي أن تنجح الاختبارات.

وفقك الله!

دورة تطوير التطبيقات باستخدام لغة JavaScript



مميزات الدورة

- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ شهادة معتمدة من أكاديمية حسوب
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ تحديثات مستمرة على الدورة مجاناً
- ✓ من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن



16. البحث المنطقي Boolean Search

سنشرح في هذا الفصل حل التمرين التالي من الفصل السابق، ثم ننفذ شيفرة تدمج مجموعة من نتائج البحث وترتيبها بحسب مدى ارتباطها بكلمات البحث.

16.1 الزاحف crawler

لنمرّ أولاً على حل تمرين الفصل المشار إليه. كنا قد وقّرنا الشيفرة المبدئية للصنف WikiCrawler وكان المطلوب منك هو إكمال التابع crawl. انظر الحقول المُعرّفة في ذلك الصنف:

```
public class WikiCrawler {  
    // يشير إلى المكان الذي بدأنا منه  
    private final String source;  
  
    // المفهرس الذي سنخزن فيه النتائج  
    private JedisIndex index;  
  
    // رتل محددات الموارد الموحدة المطلوب فهرستها  
    private Queue<String> queue = new LinkedList<String>();  
  
    // يُستخدم لقراءة الصفحات من موقع ويكيبيديا  
    final static WikiFetcher wf = new WikiFetcher();  
}
```

عندما نُنشئ كائناً من النوع WikiCrawler، علينا أن نُمرّر قيمتي source و index. يحتوي المتغير queue مبدئياً على عنصر واحد فقط هو source.

لاحظ أن الرتل `queue` مُنقذ باستخدام قائمة من النوع `LinkedList`، وبالتالي، تستغرق عملية إضافة العناصر إلى نهايته وحذفها من بدايته - زمنًا ثابتًا، ولأننا أسندنا قائمةً من النوع `LinkedList` إلى متغير من النوع `Queue`، أصبح استخدامنا له مقتصرًا على التوابيع المُعرَّفة بالواجهة `Queue`، أي سنستخدم التابع `offer` لإضافة العناصر و `poll` لحذفها منه.

انظر تنفيذنا للتابع `WikiCrawler.crawl`:

```
public String crawl(boolean testing) throws IOException {
    if (queue.isEmpty()) {
        return null;
    }
    String url = queue.poll();
    System.out.println("Crawling " + url);

    if (testing==false && index.isIndexed(url)) {
        System.out.println("Already indexed.");
        return null;
    }

    Elements paragraphs;
    if (testing) {
        paragraphs = wf.readWikipedia(url);
    } else {
        paragraphs = wf.fetchWikipedia(url);
    }
    index.indexPage(url, paragraphs);
    queueInternalLinks(paragraphs);
    return url;
}
```

السبب وراء التعقيد الموجود في التابع السابق هو تسهيل عملية اختبارهِ. تُوضّح النقاط التالية المنطق المبني عليه التابع:

- إذا كان الرتل فارغًا، يعيد التابع القيمة الفارغة `null` لكي يشير إلى أنه لم يُفهرس أي صفحة.
- إذا لم يكن فارغًا، فإنه يقرأ محدد الموارد الموحد URL التالي ويحذفه من الرتل.

- إذا كان محدد الموارد قيد الاختيار مُفهرَّسًا بالفعل، لا يُفهرَّسه التابع مرة أخرى إلا إذا كان في وضع الاختبار.
 - يقرأ التابع بعد ذلك محتويات الصفحة: إذا كان التابع في وضع الاختبار، فإنه يقرأها من ملف، وإن لم يكن كذلك، فإنه يقرأها من شبكة الإنترنت.
 - يُفهرِّس الصفحة.
 - يُحلِّل الصفحة ويضيف الروابط الداخلية الموجودة فيها إلى الرتل.
 - يعيد في النهاية مُحدَّد موارد الصفحة التي فهرَّسها للتو.
- كنا قد عرضنا تنفيذًا للتابع `Index.indexPage` في نفس الفصل المشار إليه في الأعلى، أي أن التابع الوحيد الجديد هو `WikiCrawler.queueInternalLinks`.
- كتبنا نسختين من ذلك التابع بمعاملات `parameters` مختلفة: تستقبل الأولى كائنًا من النوع `Elements` يتضمَّن شجرة DOM واحدة لكل فقرة، بينما تستقبل الثانية كائنًا من النوع `Element` يُمثل فقرة واحدة. تمرّ النسخة الأولى عبر الفقرات، في حين تُنفَّذ النسخة الثانية العمل الفعلي.

```
void queueInternalLinks(Elements paragraphs) {
    for (Element paragraph: paragraphs) {
        queueInternalLinks(paragraph);
    }
}

private void queueInternalLinks(Element paragraph) {
    Elements elts = paragraph.select("a[href]");
    for (Element elt: elts) {
        String relURL = elt.attr("href");

        if (relURL.startsWith("/wiki/")) {
            String absURL = elt.attr("abs:href");
            queue.offer(absURL);
        }
    }
}
```

لكي نُحدِّد ما إذا كان مُحدَّد موارد موحد معين هو مُحدَّد داخلي، فإننا نفحص ما إذا كان يبدأ بكلمة `"/wiki/"`. قد يتضمَّن ذلك بعض الصفحات التي لا نرغب في فهرستها مثل بعض الصفحات الوصفية لموقع

ويكيبيديا، كما قد يستثني ذلك بعض الصفحات التي نريدها مثل روابط الصفحات المكتوبة بلغات أخرى غير الإنجليزية، ومع ذلك، تُعدّ تلك الطريقة جيدة بالقدر الكافي كبداية.

لا يتضمّن هذا التمرين الكثير، فهو فرصة فقط لتجميع الأجزاء الصغيرة معًا.

16.2 استرجاع البيانات

سننتقل الآن إلى المرحلة التالية من المشروع، وهي تنفيذ أداة بحث تتكوّن مما يلي:

1. واجهة تُمكن المُستخدمين من إدخال كلمات البحث ومشاهدة النتائج.
2. طريقة لاستقبال كلمات البحث وإعادة الصفحات التي تتضمنها.
3. طريقة لدمج نتائج البحث العائدة من عدة كلمات بحث.
4. خوارزمية تُصنّف نتائج البحث وترتّبها.

يُطلَق على تلك العمليات وما يشابهها اسم استرجاع المعلومات Information retrieval.

أنشأنا بالفعل نسخة بسيطة من الخطوة رقم 2، وسُرِّكز في هذا التمرين على الخطوتين 3 و 4. قد ترغب بالعمل أيضًا على الخطوة رقم 1 إذا كنت مهتمًا ببناء تطبيقات الويب.

16.3 البحث المنطقي/الثنائي Boolean search

تستطيع معظم محركات البحث أن تُنفّذ بحثًا منطقيًا، بمعنى أن بإمكانها دمج نتائج البحث الخاصة بعدة كلمات باستخدام المنطق الثنائي، ولنأخذ أمثلةً على ذلك:

- تعيد عملية البحث عن "java AND programming" الصفحات التي تحتوي على الكلمتين "java" و "programming" فقط.
- تعيد عملية البحث عن "java OR programming" الصفحات التي تحتوي على إحدى الكلمتين وليس بالضرورة كليهما.
- تعيد عملية البحث عن "java -indonesia" الصفحات التي تحتوي على كلمة "java" ولا تحتوي على كلمة "indonesia".

يُطلَق على تلك التعبيرات، أي تلك التي تحتوي على كلمات بحث وعمليات، اسم "استعلامات queries". عندما تُطبّق تلك العمليات على نتائج البحث، فإن الكلمات "AND" و "OR" و "-" تقابل في الرياضيات عمليات "التقاطع" و "الاتحاد" و "الفرق" على الترتيب. لنفترض مثالًا أن:

- s1 يمثل مجموعة الصفحات التي تحتوي على كلمة "java"،

- s2 يمثل مجموعة الصفحات التي تحتوي على كلمة "programming"،
 - s3 يمثل مجموعة الصفحات التي تحتوي على كلمة "indonesia"،
- في تلك الحالة:
- يُمثل التقاطع بين s1 و s2 مجموعة الصفحات التي تحتوي على الكلمتين "java" و "programming" معًا.
 - يُمثل الاتحاد بين s1 و s2 مجموعة الصفحات التي تحتوي على كلمة "java" أو كلمة "programming".
 - يُمثل الفرق بين s1 و s3 مجموعة الصفحات التي تحتوي على كلمة "java" ولا تحتوي على كلمة "indonesia".
- ستكتب في القسم التالي تابعًا يُنفَّذ تلك العمليات.

16.4 تمرين 13

ستجد ملفات شيفرة هذا التمرين في مستودع الكتاب:

- WikiSearch.java: يُعرّف كائنًا يحتوي على نتائج البحث ويُطبّق العمليات عليها.
 - WikiSearchTest.java: يحتوي على شيفرة اختبار للصنف WikiSearch.
 - Card.java: يوضّح طريقة استخدام التابع sort المُعرّف بالنوع java.util.Collections.
- ستجد أيضًا بعض الأصناف المساعدة التي استخدمناها من قبل هذا الكتاب.

انظر بداية تعريف الصنف WikiSearch:

```
public class WikiSearch {

    // يربط مُحدّات الموارد التي تحتوي على الكلمة بدرجة الارتباط
    private Map<String, Integer> map;

    public WikiSearch(Map<String, Integer> map) {
        this.map = map;
    }

    public Integer getRelevance(String url) {
        Integer relevance = map.get(url);
    }
}
```

```

        return relevance==null ? 0: relevance;
    }
}

```

يحتوي كائن النوع `WikiSearch` على خريطة `map` تربط مُحَدِّدات الموارد الموحدة `URLs` بدرجة الارتباط `relevance score`، والتي تُمَثَّل -ضمن سياق استرجاع البيانات- عددًا يشير إلى المدى الذي يستوفي به مُحَدِّد الموارد الاستعلام الذي يدخله المُستخدم. تتوقَّر الكثير من الطرائق لحساب درجة الارتباط، ولكنها تعتمد في الغالب على "تردد الكلمة" أي عدد مرات ظهورها في الصفحة. يُعَدُّ `TF-IDF` واحدًا من أكثر درجات الارتباط شيوعًا، وتُمَثِّل الأحرف اختصارًا لعبارة تردد المصطلح `term frequency` - معكوس تردد المستند `inverse document frequency`.

- إذا احتوى الاستعلام على كلمة بحث واحدة، فإن درجة الارتباط لصفحة معينة تساوي تردد الكلمة، أي عدد مرات ظهورها في تلك الصفحة.
 - بالنسبة للاستعلامات التي تحتوي على عدة كلمات، تكون درجة الارتباط لصفحة معينة هي حاصل مجموع تردد الكلمات، أي عدد مرات ظهور أي كلمة منها.
- والآن وقد أصبحت مستعدًا لبدء التمرين، نَقِّذ الأمر `ant build` لتصريف ملفات الشيفرة، ثم نَقِّذ الأمر `ant WikiSearchTest`. ستفشل الاختبارات كالعادة لأن ما يزال عليك إكمال بعض العمل.
- أكمل متن كلٍّ من التوابع `and` و `or` و `minus` في الملف `WikiSearch.java` لكي تتمكن من اجتياز الاختبارات المرتبطة بتلك التوابع. لا تقلق بشأن التابع `testSort`، فسنعود إليه لاحقًا.
- يُمكنك أن تُنَفِّذ `WikiSearchTest` بدون إستخدام `Jedis` لأنه لا يعتمد على فهرس قاعدة بيانات `Redis` الخاصة بك، ولكن، إذا أردت أن تَسْتَخِدم الفهرس للاستعلام `query`، فلا بُدَّ أن توقَّر بيانات الخادم في ملف، كما أوضحنا في الفصل 14 حفظ البيانات عبر `Redis`.

نَقِّذ الأمر `JedisMaker ant` لكي تتأكَّد من قدرته على الاتصال بخادم `Redis`، ثم نَقِّذ `WikiSearch` الذي يَطْبَع نتائج الاستعلامات الثلاثة التالية:

• "java"

• "programming"

• "java AND programming"

لن تكون النتائج مُرتَّبة في البداية لأن التابع `WikiSearch.sort` ما يزال غير مكتمل.

أكمل متن التابع `sort` لكي تُصيح النتائج مُرتَّبة تصاعديًا بحسب درجة الارتباط. يُمكنك الاستعانة بالتابع `sort` المُعرَّف بالنوع `java.util.Collections` حيث يُمكنه ترتيب أي نوع قائمة `List`. يُمكنك الاطلاع على توثيق النوع `List`.

تتوفّر نسختان من التابع `sort`:

- نسخة أحادية المعامل تُستقبل قائمةً وتُرتَّب عناصرها باستخدام التابع `compareTo`، ولذلك ينبغي أن تكون العناصر من النوع `Comparable`.
- نسخة ثنائية المعامل تُستقبل قائمةً من أي نوع وكائنًا من النوع `Comparator`، ويُستخدَم التابع `compare` المُعرَّف ضمن الكائن لموازنة العناصر.

سنحدث عن الواجهتين `Comparable` و `Comparator` في القسم التالي إن لم تكن على معرفة بهما.

16.5 الواجهتان `Comparable` و `Comparator`

يتضمَّن مستودع الكتاب الصنف `Card` الذي يحتوي على طريقتين لترتيب قائمة كائنات من النوع `Card`. انظر إلى بداية تعريف الصنف:

```
public class Card implements Comparable<Card> {

    private final int rank;
    private final int suit;

    public Card(int rank, int suit) {
        this.rank = rank;
        this.suit = suit;
    }
}
```

تحتوي كائنات الصنف `Card` على الحقلين `rank` و `suit` من النوع العددي الصحيح. يُنفَّذ الصنف `Card` الواجهة `Comparable<Card>` مما يعني أنه بالضرورة يُوفّر تنفيذًا للتابع `compareTo`:

```
public int compareTo(Card that) {
    if (this.suit < that.suit) {
        return -1;
    }
    if (this.suit > that.suit) {
        return 1;
    }
}
```

```

    if (this.rank < that.rank) {
        return -1;
    }
    if (this.rank > that.rank) {
        return 1;
    }
    return 0;
}

```

تشير بصمة التابع `compareTo` إلى أن عليه أن يعيد عددًا سالبًا إذا كان `this` أقل من `that`، وعددًا موجبًا إذا كان أكبر منه، وصفرًا إذا كانا متساويين.

إذا استخدمت نسخة التابع `Collections.sort` أحادية المعامل، فإنها بدورها تستدعي التابع `compareTo` المُعرّف ضمن العناصر لكي تتمكن من ترتيبها. على سبيل المثال، تُنشئ الشيفرة التالية قائمة تحتوي على 52 بطاقة:

```

public static List<Card> makeDeck() {
    List<Card> cards = new ArrayList<Card>();
    for (int suit = 0; suit <= 3; suit++) {
        for (int rank = 1; rank <= 13; rank++) {
            Card card = new Card(rank, suit);
            cards.add(card);
        }
    }
    return cards;
}

```

ثم تُرتَّبها كالتالي:

```
Collections.sort(cards);
```

تُرتَّب تلك النسخة من التابع `sort` العناصر وفقًا لما يُطلق عليه "الترتيب الطبيعي" لأن الترتيب مُحدّد بواسطة العناصر نفسها.

في المقابل، يُمكننا أيضًا أن نستعين بكائن من النوع `Comparator` لكي نَعرِّض نوعًا مختلفًا من الترتيب. على سبيل المثال، تحتل بطاقات الأص المرتبة الأقلّ بحسب الترتيب الطبيعي للصف `Card`، ومع ذلك، فإنها أحيانًا تحتل المرتبة الأكبر في بعض ألعاب البطاقات، ولذلك، سنُعرِّف كائنًا من النوع `Comparator` يُعامل الأص على أنّها البطاقة الأكبر ضمن مجموعة بطاقات اللعب. انظر إلى شيفرة ذلك النوع:

```

Comparator<Card> comparator = new Comparator<Card>() {
    @Override
    public int compare(Card card1, Card card2) {
        if (card1.getSuit() < card2.getSuit()) {
            return -1;
        }
        if (card1.getSuit() > card2.getSuit()) {
            return 1;
        }
        int rank1 = getRankAceHigh(card1);
        int rank2 = getRankAceHigh(card2);

        if (rank1 < rank2) {
            return -1;
        }
        if (rank1 > rank2) {
            return 1;
        }
        return 0;
    }

    private int getRankAceHigh(Card card) {
        int rank = card.getRank();
        if (rank == 1) {
            return 14;
        } else {
            return rank;
        }
    }
};

```

تُعرّف تلك الشيفرة صنفًا مجهول الاسم anonymous يُنفَّذ التابع `compare` على النحو المطلوب، ثم تُنشئ نسخةً منه. يُمكنك القراءة عن الأصناف مجهولة الاسم `Anonymous Classes` في لغة جافا إذا لم تكن على معرفة بها.

يُمكننا الآن أن نُمرّر ذلك الكائن المنتمي للنوع `Comparator` إلى التابع `sort`، كما هو مبين في الشيفرة

التالية:

```
Collections.sort(cards, comparator);
```

يُعدّ الأص البستوني وفقًا لهذا الترتيب البطاقة الأكبر ضمن مجموعة بطاقات اللعب، بينما تعد البطاقة ذات الرقم 2 البطاقة الأصغر.

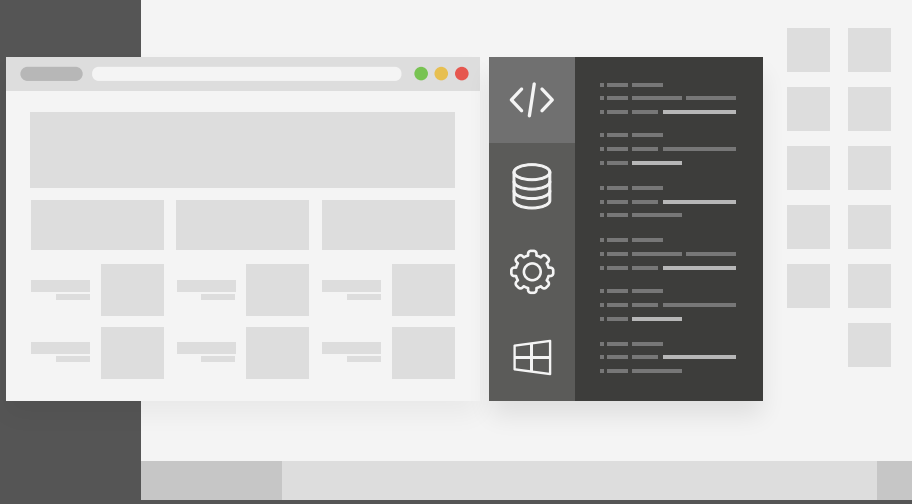
ستجد شيفرة هذا القسم في الملف `Card.java` إن كنت تريد تجربته. قد ترغب أيضًا في كتابة كائن آخر من النوع `Comparator` يُرتّب العناصر بناءً على قيمة `rank` أولاً ثم قيمة `suit`، وبالتالي، تصبح جميع بطاقات الأص معًا وجميع البطاقات الثنائية معًا، وهكذا.

16.6 ملحقات

إذا تمكّنت من كتابة الكائن الذي أشرنا إليه في الأعلى، هاك بعض الأفكار الأخرى التي يُمكنك أن تحاول القيام بها:

- اقرأ عن درجة الارتباط `TF-IDF` ونفّذها. قد تحتاج إلى تعديل الصنف `JavaIndex` لكي تجعله يحسب قيمة ترددات المستند أي عدد مرات ظهور كل كلمة في جميع الصفحات الموجودة بالفهرس.
- بالنسبة للاستعلامات المكوّنة من أكثر من كلمة واحدة، يُمكنك أن تحسب درجة الارتباط الإجمالية لكل صفحة بحساب مجموع درجة الارتباط لجميع الكلمات. فكر متى يُمكن لهذه النسخة المبسطة أن تفشل وجرب بدائل أخرى.
- أنشئ واجهة مُستخدم تسمّح للمستخدمين بإدخال استعلامات تحتوي على عوامل `operators` منطقية. حلّ الاستعلامات المُدخلة، وولّد النتائج، ثم رتّبها بحسب درجة الارتباط، واعرض مُحدّدات الموارد التي أحرزت أعلى درجات. حاول أيضًا أن تولّد مقطع شيفرة يعرض مكان ظهور كلمات البحث في الصفحة.

دورة علوم الحاسوب



مميزات الدورة

- ✓ شهادة معتمدة من أكاديمية حاسوب
- ✓ إرشادات من المدربين على مدار الساعة
- ✓ من الصفر دون الحاجة لخبرة مسبقة
- ✓ بناء معرض أعمال قوي بمشاريع حقيقية
- ✓ وصول مدى الحياة لمحتويات الدورة
- ✓ تحديثات مستمرة على الدورة مجاناً

اشترك الآن



17. الترتيب Sorting

لدى أقسام علوم الحاسوب هوس غير طبيعي بخوارزميات الترتيب، فبناءً على الوقت الذي يمضيه طلبة علوم الحاسوب في دراسة هذا الموضوع، قد تظن أن الاختيار ما بين خوارزميات الترتيب هو حجر الزاوية في هندسة البرمجيات الحديثة. واقع الأمر هو أن مطوري البرمجيات قد يقضون سنوات قد تصل إلى مسارهم المهني بأكمله دون التفكير في طريقة حدوث عملية الترتيب، فهم يستخدمون في كل التطبيقات تقريبًا الخوارزمية متعددة الأغراض التي توفرها اللغة أو المكتبة التي يستخدمونها، والتي تكون كافية في معظم الحالات.

لذلك لو تجاهلت هذا الفصل ولم تتعلم أي شيء عن خوارزميات الترتيب، فما يزال بإمكانك أن تصبح مُطوّر برمجيات جيدًا، ومع ذلك، هناك عدة أسباب قد تدفعك لقراءته:

1. على الرغم من وجود خوارزميات متعددة الأغراض يُمكنها العمل بشكل جيد في غالبية التطبيقات، هنالك خوارزميتان مُتخصّصتان قد تحتاج إلى معرفة القليل عنهما: الترتيب بالجذر radix sort والترتيب بالكومة المقيدة bounded heap sort.

2. تُعدّ خوارزمية الترتيب بالدمج merge sort المثال التعليمي الأمثل، فهي تُوضّح استراتيجية "قسّم واغزّر divide and conquer" المهمة والمفيدة في تصميم الخوارزميات. بالإضافة إلى ذلك، ستتعلم عن ترتيب نمو order of growth لم تُره من قبل، هو ترتيب النمو "الخطي-اللوغاريتمي linearithmic". ومن الجدير بالذكر أن غالبية الخوارزميات الشهيرة تكون عادةً خوارزميات هجينة وتستخدم بشكلٍ أو بآخر فكرة الترتيب بالدمج.

3. أحد الأسباب الأخرى التي قد تدفعك إلى تعلم خوارزميات الترتيب هي مقابلات العمل التقنية، فعادةً ما تُسأل خلالها عن تلك الخوارزميات. إذا كنت تريد الحصول على وظيفة، فسيُساعدك إظهار اطلاعك على أبجديات علوم الحاسوب.

سُحلِّل في هذا الفصل خوارزمية الترتيب بالإدراج Insertion sort، وسننقِّذ خوارزمية الترتيب بالدمج، وأخيرًا، سنشرح خوارزمية الترتيب بالجذر، وسنكتب نسخةً بسيطةً من خوارزمية الترتيب بالكومة المُقيَّدة.

17.1 الترتيب بالإدراج Insertion sort

سنبدأ بخوارزمية الترتيب بالإدراج، لأنها بسيطة ومهمة. على الرغم من أنها ليست الخوارزمية الأكفأ إلا أنها تملك بعض الميزات المتعلقة بتحرير الذاكرة كما سنرى لاحقًا.

لن نشرح هذه الخوارزمية هنا، ولكن يُفضَّل لو قرأت مقالة ويكيبيديا عن الترتيب بالإدراج Insertion Sort، فهي تحتوي على شيفرة وهمية وأمثلة متحركة. وبعدها تفهم فكرتها العامة يمكنك متابعة القراءة هنا. تعرِّض الشيفرة التالية تنفيذًا بلغة جافا لخوارزمية الترتيب بالإدراج:

```
public class ListSorter<T> {

    public void insertionSort(List<T> list, Comparator<T> comparator)
    {

        for (int i=1; i < list.size(); i++) {
            T elt_i = list.get(i);
            int j = i;
            while (j > 0) {
                T elt_j = list.get(j-1);
                if (comparator.compare(elt_i, elt_j) >= 0) {
                    break;
                }
                list.set(j, elt_j);
                j--;
            }
            list.set(j, elt_i);
        }
    }
}
```

عرَّفنا الصنف ListSorter ليَعْمَل كحاوٍ لخوارزميات الترتيب. نظرًا لأننا استخدمنا معامل نوع type parameter، اسمه T، ستمكَّن التوابع التي سنكتبها من العمل مع قوائم تحتوي على أي نوع من الكائنات.

يستقبل التابع `insertionSort` معاملين: الأول عبارة عن قائمة من أي نوع ممتد من الواجهة `List` والثاني عبارة عن كائن من النوع `Comparator` بإمكانه موازنة كائنات النوع `T`. يُرتَّب هذا التابع القائمة في نفس المكان أي أنه يُعدِّل القائمة الموجودة ولا يحتاج إلى حجز مساحة إضافية جديدة.

تستدعي الشيفرة التالية هذا التابع مع قائمة من النوع `List` تحتوي على كائنات من النوع `Integer`:

```
List<Integer> list = new ArrayList<Integer>(
    Arrays.asList(3, 5, 1, 4, 2));

Comparator<Integer> comparator = new Comparator<Integer>() {
    @Override
    public int compare(Integer elt1, Integer elt2) {
        return elt1.compareTo(elt2);
    }
};

ListSorter<Integer> sorter = new ListSorter<Integer>();
sorter.insertionSort(list, comparator);
System.out.println(list);
```

يحتوي التابع `insertionSort` على حلقتين متداخلتين `nested loops`، ولذلك، قد تظن أن زمن تنفيذه تربيعي، وهذا صحيح في تلك الحالة، ولكن قبل أن تتوصل إلى تلك النتيجة، عليك أولاً أن تتأكد من أن عدد مرات تنفيذ كل حلقة يتناسب مع حجم المصفوفة `n`.

تتكرر الحلقة الخارجية من 1 إلى `list.size()`، ولذلك تُعدّ خطيَّةً بالنسبة لحجم القائمة `n`، بينما تتكرر الحلقة الداخلية من 1 إلى صفر، لذلك هي أيضاً خطيَّةً بالنسبة لقيمة `n`. بناءً على ذلك، يكون عدد مرات تنفيذ الحلقة الداخلية تربيعيًّا.

إذا لم تكن متأكدًا من ذلك، انظر إلى البرهان التالي:

- في المرة الأولى، قيمة `i` تساوي 1، وتتكرر الحلقة الداخلية مرةً واحدةً على الأكثر.
 - في المرة الثانية، قيمة `i` تساوي 2، وتتكرر الحلقة الداخلية مرتين على الأكثر.
 - في المرة الأخيرة، قيمة `i` تساوي `n-1`، وتتكرر الحلقة الداخلية عددًا قدره `n-1` من المرات على الأكثر.
- وبالتالي، يكون عدد مرات تنفيذ الحلقة الداخلية هو مجموع المتتالية 1، 2، ... حتى `n-1`، وهو ما يُساوي $n(n-1)/2$. لاحظ أن القيمة الأساسية (ذات الأس الأكبر) بهذا التعبير هي n^2 .

يُعدّ الترتيب بالإدراج تربيعيًّا في أسوأ حالة، ومع ذلك:

1. إذا كانت العناصر مُرتَّبةً أو شبه مُرتَّبةً بالفعل، فإن الترتيب بالإدراج يكون خَطَّيًّا. بالتحديد، إذا لم يكن كل عنصرٍ أبعدَ من موضعه الصحيح مسافةً أكبرَ من k ، فإن الحلقة الداخلية لن تُنقَّذ أكثرَ من عددٍ قدره k من المرات، ويكون زمن التنفيذ الكلي هو $O(kn)$.
 2. نظرًا لأن تنفيذ تلك الخوارزمية بسيط، فإن تكلفته منخفضة، أي على الرغم من أن زمن التنفيذ يساوي n^2 ، إلا أن المعامل a قد يكون صغيرًا.
- ولذلك، إذا عرفنا أن المصفوفة شبه مُرتَّبة أو إذا لم تكن كبيرةً جدًّا، فقد يكون الترتيب بالإدراج خيارًا جيدًا، ولكن بالنسبة للمصفوفات الكبيرة، فهناك خيارات أفضل بكثير.

17.2 تمرين 14

تُعدّ خوارزمية الترتيب بالدمج merge sort واحدةً من ضمن مجموعةٍ من الخوارزميات التي يتفوق زمن تنفيذها على الزمن التريبيعيّ. نصحك قبل المتابعة بقراءة مقالة ويكيبيديا عن الترتيب بالدمج merge sort. بعد أن تفهم الفكرة العامة للخوارزمية، يُمكنك العودة لاختبار فهمك بكتابة تنفيذٍ لها.

ستجد ملفات الشيفرة التالية الخاصة بالتمرين في مستودع الكتاب:

• ListSorter.java

• ListSorterTest.java

نقِّذ الأمر `ant build` لتصريف ملفات الشيفرة ثم نقِّذ الأمر `ant ListSorterTest`. سيفشل الاختبار كالعادة لأن ما يزال عليك إكمال بعض الشيفرة.

ستجد ضمن الملف `ListSorter.java` شيفرةً مبدئيةً للتابعين `mergeSort` و `mergeSortInPlace`:

```
public void mergeSortInPlace(List<T> list, Comparator<T>
comparator) {
    List<T> sorted = mergeSortHelper(list, comparator);
    list.clear();
    list.addAll(sorted);
}

private List<T> mergeSort(List<T> list, Comparator<T> comparator)
{
    // TODO: fill this in!
    return null;
}
```

يقوم التابعان بنفس الشيء، ولكنهما يوفران واجهاتٍ مختلفة. يَسْتَقْبِلُ التابع mergeSort قائمةً ويعيد قائمةً جديدةً تحتوي على نفس العناصر بعد ترتيبها ترتيبًا تصاعديًا. في المقابل، يُعَدِّلُ التابع mergeSortInPlace القائمة ذاتها ولا يعيد أيّة قيمة.

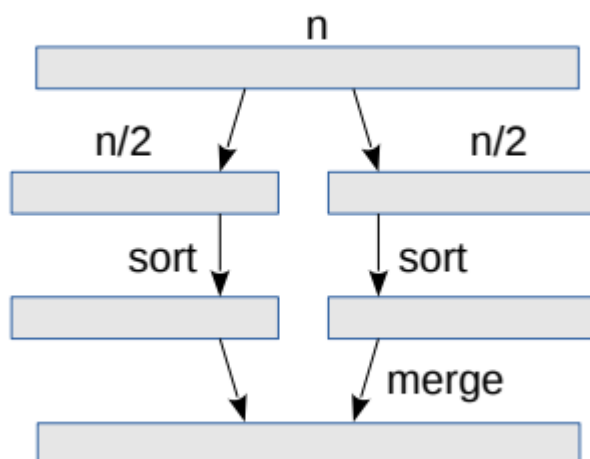
عليك إكمال التابع mergeSort. ويمكنك بدلًا من كتابة نسخة تعاودية recursive بالكامل، أن تتبع الطريقة التالية:

1. قَسِّمِ القائمة إلى نصفين.
 2. رَتِّبِ النصفين باستخدام التابع Collections.sort أو التابع insertionSort.
 3. ادمج النصفين المُرتَّبين إلى قائمة واحدة مُرتَّبة.
- سيعطيك هذا التمرين الفرصة لتنقيح شيفرة الدمج دون التعامل مع تعقيدات التوابيع التعاودية.
- والآن أضف حالة أساسية `base case`. إذا كان لديك قائمة تحتوي على عنصر واحد فقط، يُمكنك أن تعيدها مباشرةً لأنها نوعًا ما مُرتَّبة بالفعل، وإذا كان طول القائمة أقل من قيمة معينة، يُمكنك أن تُرتِّبها باستخدام التابع Collections.sort أو التابع insertionSort. اختبر الحالة الأساسية قبل إكمال القراءة.
- أخيرًا، عدِّل الحل واجعله يُنفِّذ استدعاءين تعاوديين لترتيب نصفي المصفوفة. إذا عدلته بالشكل الصحيح، ينبغي أن ينجح الاختباران `testMergeSort` و `testMergeSortInPlace`.

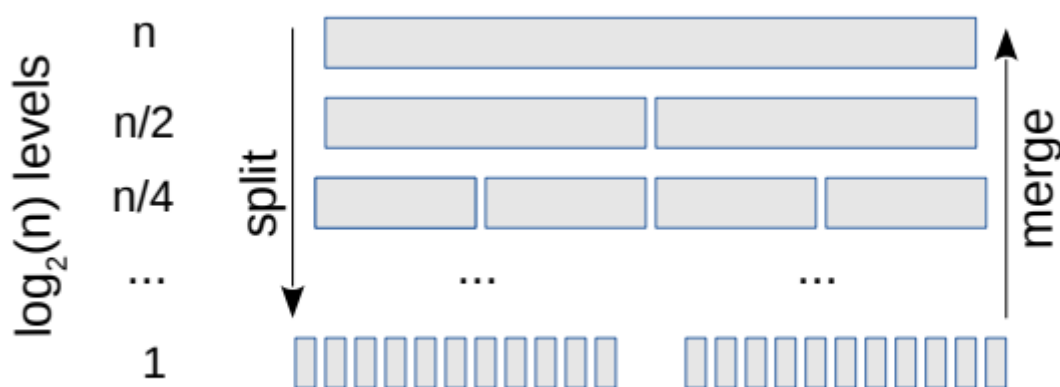
17.3 تحليل أداء خوارزمية الترتيب بالدمج

لكي نصنف زمن تنفيذ خوارزمية الترتيب بالدمج، علينا أن نفكر بمستويات التعاود وبكمية العمل المطلوب في كل مستوى. لنفترض أننا سنبدأ بقائمةٍ تحتوي على عددٍ قدره n من العناصر. وفيما يلي خطوات الخوارزمية:

1. نُنشِئ مصفوفتين وننسخ نصف العناصر إليهما.
2. نُرتِّب النصفين.
3. ندمج النصفين.



تنسخ الخطوة الأولى كل عنصر مرة واحدة، أي أنها خطيّة. بالمثل، تنسخ الخطوة الثالثة كل عنصر مرة واحدة فقط، أي أنها خطيّة كذلك. علينا الآن أن نُحدّد تعقيد الخطوة الثانية. ستساعدنا على ذلك الصورة التالية التي تُعرض مستويات التعاود.



في المستوى الأعلى، سيكون لدينا قائمة واحدة مُكوّنة من عددٍ قدره n من العناصر. للتبسيط، سنفترض أن n عبارة عن قيمة مرفوعة للأس 2، وبالتالي، سيكون لدينا في المستوى التالي قائمتان تحتويان على عدد $n/2$ من العناصر. ثم في المستوى التالي، سيكون لدينا 4 قوائم تحتوي على عدد قدره $n/4$ من العناصر، وهكذا حتى نصل إلى عدد n من القوائم تحتوي جميعها على عنصر واحد فقط.

لدينا إذاً عدد قدره n من العناصر في كل مستوى. أثناء نزولنا في المستويات، قسّمنا المصفوفات في كل مستوى إلى نصفين، وهو ما يستغرق زمناً يتناسب مع n في كل مستوى، وأثناء صعودنا للأعلى، علينا أن ندمج عدداً من العناصر مجموعه n وهو ما يستغرق زمناً خطياً أيضاً.

إذا كان عدد المستويات يساوي h ، فإن العمل الإجمالي المطلوب يساوي $O(nh)$ ، والآن، كم هو عدد المستويات؟ يُمكننا أن نفكر في ذلك بطريقتين:

1. كم عدد المرات التي سنضطر خلالها لتقسيم n إلى نصفين حتى نصل إلى 1.
 2. أو كم عدد المرات التي سنضطرّ خلالها لمضاعفة العدد 1 قبل أن نصل إلى n .
- يُمكننا طرح السؤال الثاني بطريقة أخرى: "ما هي قيمة الأس المرفوع للعدد 2 لكي نحصل على n ؟".

$$2^h = n$$

بحساب لوغاريتم أساس 2 لكلا الطرفين، نحصل على التالي:

$$h = \log_2 n$$

أي أن الزمن الكلي يساوي $O(n \log(n))$. لاحظ أننا تجاهلنا قيمة أساس اللوغاريتم لأن اختلاف أساس اللوغاريتم يؤثر فقط بعامل ثابت، أي أن جميع اللوغاريتمات لها نفس ترتيب النمو order of growth. يُطلق أحيانًا على الخوارزميات التي تنتمي إلى $O(n \log(n))$ اسم "خطي-لوغاريتمي linearithmic"، ولكن في العادة نقول " $n \log n$ ".

في الواقع، يُعدّ $O(n \log(n))$ الحد الأدنى من الناحية النظرية لخوارزميات الترتيب التي تعتمد على موازنة العناصر مع بعضها البعض. يعني ذلك أنه لا توجد خوارزمية ترتيب بالموازنة ذات ترتيب نمو أفضل من $n \log n$.

ولكن كما سنرى في القسم التالي، هناك خوارزميات ترتيب لا تعتمد على الموازنة وتستغرق زمنًا خطيًا.

17.4 خوارزمية الترتيب بالجذر Radix sort

إن واجهنا سؤالاً عن أكفأ طريقة لترتيب مليون عدد صحيح من نوع 32 بت فلن تكون خوارزمية ترتيب الفقاعات الطريقة الأفضل، فخوارزمية ترتيب الفقاعات bubble sort صحيح أنها بسيطة وسهلة الفهم، لكنها تستغرق زمنًا تربيعيًا، كما أن أدائها ليس جيدًا بالموازنة مع خوارزميات الترتيب التربيعية الأخرى.

ربما خوارزمية الترتيب بالجذر radix sort هي الإجابة الأدق عن السؤال، فهي خوارزمية ترتيب غير مبنية على الموازنة، كما أنها تعمل بنجاح عندما يكون حجم العناصر مقيّدًا كعدد صحيح من نوع 32 بت أو سلسلة نصية مكوّنة من 20 حرفًا.

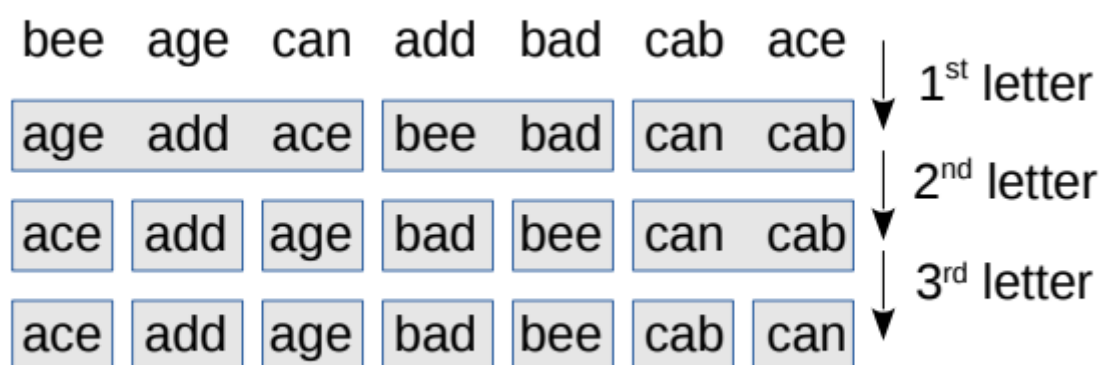
لكي نفهم طريقة عملها، لنتخيل أن لدينا مكدّسًا stack من البطاقات، وكل واحدة منها تحتوي على كلمة مكوّنة من ثلاثة أحرف. ها هي الطريقة التي يُمكن أن نرتب بها تلك البطاقات:

1. مرّ عبر البطاقات وقسمها إلى مجموعات بناءً على الحرف الأول، أي ينبغي أن تكون الكلمات البادئة بالحرف a ضمن مجموعة واحدة، يليها الكلمات التي تبدأ بحرف b، وهكذا.

2. قسّم كل مجموعة مرة أخرى بناءً على الحرف الثاني، بحيث تصبح الكلمات البادئة بالحرفين aa معًا، يليها الكلمات التي تبدأ بالحرفين ab، وهكذا. لن تكون كل المجموعات مملوءة بالتأكيد، ولكن لا بأس بذلك.

3. قسّم كل مجموعة مرة أخرى بحسب الحرف الثالث.

والآن، أصبحت كل مجموعة مُكوّنة من عنصر واحد فقط، كما أصبحت المجموعات مُرتّبة ترتيبًا تصاعديًا. تُعرض الصورة التالية مثالًا عن الكلمات المكوّنة من ثلاثة أحرف.



يُعرض الصف الأول الكلمات غير المُرتّبة، بينما يُعرض الصف الثاني شكل المجموعات بعد اجتيازها أو التنقل فيها للمرة الأولى. تبدأ كلمات كل مجموعة بنفس الحرف.

بعد اجتياز الكلمات للمرة الثانية، تبدأ كلمات كل مجموعة بنفس الحرفين الأوليين، وبعد اجتيازها للمرة الثالثة، سيكون هنالك كلمة واحدة فقط في كل مجموعة، وستكون المجموعات مُرتّبة.

أثناء كل اجتياز، نمرّ عبر العناصر ونضيفها إلى المجموعات. يُعدّ كل اجتياز منها خطيًّا طالما كانت تلك المجموعات تسمّح بإضافة العناصر إليها بزمّن خطي.

تعتمد عدد مرات الاجتياز أو التنقل -التي سنطلق عليها w- على عرض الكلمات، ولكنه لا يعتمد على عدد الكلمات n، وبالتالي، يكون ترتيب النمو $O(wn)$ وهو خطي بالنسبة لقيمة n.

تتوفّر نسخ أخرى من خوارزمية الترتيب بالجذر، ويُمكن تنفيذ كُلٍّ منها بطرق كثيرة. يُمكنك قراءة المزيد عن خوارزمية الترتيب بالجذر، كما يُمكنك أن تحاول كتابة تنفيذ لها.

17.5 خوارزمية الترتيب بالكومة Heap sort

إلى جانب خوارزمية الترتيب بالجذر التي تُطبَّق عندما يكون حجم الأشياء المطلوب ترتيبها مقيّدًا، هنالك خوارزمية مُخصّصة أخرى هي خوارزمية الترتيب بالكومة المُقيّدة، والتي تُطبَّق عندما نعمل مع بيانات ضخمة جدًا ونحتاج إلى معرفة أكبر 10 أو أكبر عدد k حيث k قيمة أصغر بكثير من n .

لنفترض مثلًا أننا نراقب خدمة عبر الإنترنت تتعامل مع بلايين المعاملات يوميًا، وأننا نريد في نهاية كل يوم معرفة أكبر k من المعاملات (أو أبطأ أو أي معيار آخر). يُمكننا مثلًا أن نُخزّن جميع المعاملات، ثم نرتبها في نهاية اليوم، ونختار أول k من المعاملات. سيستغرق ذلك زمنًا يتناسب مع $n \log n$ ، وسيكون بطيئًا جدًا لأننا من المحتمل ألا نتمكّن من ملاءمة بلايين المعاملات داخل ذاكرة برنامج واحد، وبالتالي، قد نضطرّ لاستخدام خوارزمية ترتيب بذاكرة خارجية (خارج النواة).

يُمكننا بدلًا من ذلك أن نستخدم كومة مُقيّدة heap. إليك ما سنفعله في ما تبقى من هذا الفصل:

1. سنشرح خوارزمية الترتيب بالكومة (غير المقيدة).

2. سننفذ الخوارزمية.

3. سنشرح خوارزمية الترتيب بالكومة المُقيّدة ونحلّها.

لكي تفهم ما يعنيه الترتيب بالكومة، عليك أولاً فهم ماهية الكومة. الكومة ببساطة عبارة عن هيكل بياني data structure مشابه لشجرة البحث الثنائية binary search tree. تتلخص الفروق بينهما في النقاط التالية:

- تتمتع أي عقدة x بشجرة البحث الثنائية بـ "خاصية BST" أي تكون جميع عقد الشجرة الفرعية subtree الموجود على يسار العقدة x أصغر من x كما تكون جميع عقد الشجرة الفرعية الموجودة على يمينها أكبر من x .
- تتمتع أي عقدة x ضمن الكومة بـ "خاصية الكومة" أي تكون جميع عقد الشجرتين الفرعيتين للعقدة x أكبر من x .
- تتشابه الكومة مع أشجار البحث الثنائية المتزنة من جهة أنه عندما تضيف العناصر إليها أو تحذفها منها، فإنها قد تقوم ببعض العمل الإضافي لضمان استمرارية اتزان الشجرة، وبالتالي، يُمكن تنفيذها بكفاءة باستخدام مصفوفة من العناصر.

دائمًا ما يكون جذر الكومة هو العنصر الأصغر، وبالتالي، يُمكننا أن نعرّض عليه بزمّن ثابت. تستغرق إضافة العناصر وحذفها من الكومة زمنًا يتناسب مع طول الشجرة h ، ولأن الكومة دائمًا ما تكون متزنة، فإن h يتناسب مع $\log n$. يُمكنك قراءة المزيد عن الكومة لو أردت.

تُنفَّذ جافا الصنف `PriorityQueue` باستخدام كومة. يحتوي ذلك الصنف على التوابع المُعرَّفة في الواجهة `Queue` ومن بينها التابعان `offer` و `poll` اللذان نلخص عملهما فيما يلي:

1. `offer`: يضيف عنصراً إلى الرتل `queue`، ويُحدِّث الكومة بحيث يَضمَّن استيفاء "خاصية الكومة" لجميع العقد. لاحظ أنه يستغرق زمناً يتناسب مع $\log n$.

2. `poll`: يَحذف أصغر عنصر من الرتل من الجذر ويُحدِّث الكومة. يستغرق أيضاً زمناً يتناسب مع $\log n$.

إذا كان لديك كائن من النوع `PriorityQueue`، تستطيع بسهولة ترتيب تجميعية عناصر طولها n على النحو التالي:

1. أضف جميع عناصر التجميعية إلى كائن الصنف `PriorityQueue` باستخدام التابع `offer`.
 2. احذف العناصر من الرتل باستخدام التابع `poll` وأضفها إلى قائمة من النوع `List`.
نظراً لأن التابع `poll` يعيد أصغر عنصر مُتبقِّ في الرتل، فإن العناصر تُضاف إلى القائمة مُرتَّبةً تصاعدياً. يُطلق على هذا النوع من الترتيب اسم الترتيب بالكومة.
- تستغرق إضافة عددٍ قدره n من العناصر إلى رتلٍ زمناً يتناسب مع $n \log n$ ، ونفس الأمر ينطبق على حذف عددٍ قدره n من العناصر منه، وبالتالي، تنتمي خوارزمية الترتيب بالكومة إلى المجموعة $O(n \log(n))$.
- ستجد ضمن الملف `java.ListSorter` تعريفاً مبدئياً لتابع اسمه `heapSort`. أكمله ونفِّذ الأمر `ant ListSorterTest` لكي تتأكد من أنه يَعْمَل بشكل صحيح.

17.6 الكومة المُقيَّدة Bounded heap

تَعْمَل الكومة المقيدة كأى كومة عادية، ولكنها تكون مقيدة بعدد k من العناصر. إذا كان لديك عدد n من العناصر، يُمكنك أن تحتفظ فقط بأكبر عدد k من العناصر باتباع التالي:

- ستكون الكومة فارغة مبدئياً، وعليك أن تُنفَّذ التالي لكل عنصر x :
- التفريغ الأول: إذا لم تكن الكومة ممتلئة، أضف x إلى الكومة.
- التفريغ الثاني: إذا كانت الكومة ممتلئة، وازن قيمة x مع أصغر عنصر في الكومة. إذا كانت قيمة x أصغر، فلا يُمكن أن تكون ضمن أكبر عدد k من العناصر، ولذلك، يُمكنك أن تتجاهلها.
- التفريغ الثالث: إذا كانت الكومة ممتلئة، وكانت قيمة x أكبر من قيمة أصغر عنصر بالكومة، احذف أصغر عنصر، وأضف x مكانه.

بوجود أصغر عنصر أعلى الكومة، يُمكننا الاحتفاظ بأكبر عدد k من العناصر. لنحل الآن أداء هذه الخوارزمية. إننا نفِّذ ما يلي لكل عنصر:

- التفريغ الأول: تستغرق إضافة عنصر إلى الكومة زمناً يتناسب مع $O(\log k)$.
- التفريغ الثاني: يستغرق العثور على أصغر عنصر بالكومة زمناً يتناسب مع $O(1)$.
- التفريغ الثالث: يستغرق حذف أصغر عنصر زمناً يتناسب مع $O(\log k)$ ، كما أن إضافة x تستغرق نفس مقدار الزمن.

في الحالة الأسوأ، تكون العناصر مُرتَّبة تصاعدياً، وبالتالي، نُنفِّذ التفريغ الثالث دائماً، ويكون الزمن الإجمالي لمعالجة عدد n من العناصر هو $O(n \log K)$ أي خطِّي مع n .

ستجد في الملف `ListSorter.java` تعريفاً مبدئياً لتابع اسمه `topK`. يستقبل هذا التابع قائمةً من النوع `List` وكائنًا من النوع `Comparator` وعدداً صحيحاً k ، ويعيد أكبر عدد k من عناصر القائمة بترتيب تصاعدي. أكمل متن التابع ثم نفِّذ الأمر `ant ListSorterTest` لكي تتأكد من أنه يعمل بشكل صحيح.

17.7 تعقيد المساحة Space complexity

تحدثنا حتى الآن عن تحليل زمن التنفيذ فقط، ولكن بالنسبة لكثير من الخوارزميات، ينبغي أن نُولي للمساحة التي تتطلبها الخوارزمية بعض الاهتمام. على سبيل المثال، تحتاج خوارزمية الترتيب بالدمج `merge sort` إلى إنشاء نسخ من البيانات، وقد كانت مساحة الذاكرة الإجمالية التي تطلبها تنفيذنا لتلك الخوارزمية هو $O(n \log n)$. في الواقع، يُمكننا أن نُخفِّض ذلك إلى $O(n)$ إذا نفذنا نفس الخوارزمية بطريقة أفضل.

في المقابل، لا تنسخ خوارزمية الترتيب بالإدراج `insertion sort` البيانات لأنها تُرتَّب العناصر في أماكنها، وتستخدم متغيرات مؤقتة لموازنة عنصرين في كل مرة، كما تستخدم عدداً قليلاً من المتغيرات المحلية `local` الأخرى، ولكن المساحة التي تتطلبها لا تعتمد على n .

تُنشئ نسختنا من خوارزمية الترتيب بالكومة كائناً جديداً من النوع `PriorityQueue`، وتُخزَّن فيه العناصر، أي أن المساحة المطلوبة تنتمي إلى المجموعة $O(n)$ ، وإذا سمحنا بترتيب عناصر القائمة في نفس المكان، يُمكننا أن نُخفِّض المساحة المطلوبة لتنفيذ خوارزمية الترتيب بالكومة إلى المجموعة $O(1)$.

من مميزات النسخة التي نفَّذناها من تلك الخوارزمية هي أنها تحتاج فقط إلى مساحة تتناسب مع k (أي عدد العناصر المطلوب الاحتفاظ بها)، وعادةً ما تكون k أصغر بكثير من قيمة n .

يميل مطورو البرمجيات إلى التركيز على زمن التشغيل وإهمال حيز الذاكرة المطلوب، وهذا في الحقيقة، مناسب لكثير من التطبيقات، ولكن عند التعامل مع بيانات ضخمة، تكون المساحة المطلوبة بنفس القدر من الأهمية إن لم تكن أهم، كما في الحالات التالية مثلاً:

1. إذا لم تكن مساحة ذاكرة البرنامج ملائمة للبيانات، عادةً ما يزداد زمن التشغيل إلى حد كبير وقد لا يعمل البرنامج من الأساس. إذا اخترت خوارزمية تتطلب حيزاً أقل من الذاكرة، وتسمح بملائمة المعالجة ضمن

الذاكرة، فقد يَعْمَل البرنامج بسرعةٍ أعلى بكثير. بالإضافة إلى ذلك، تَسْتَغِل البرامج التي تَتَطَلَّب مساحة ذاكرة أقل الذاكرة المؤقتة لوحدة المعالجة المركزية CPU caches بشكل أفضل وتَعْمَل بسرعة أكبر.

2. في الخوادم التي تُشغَّل برامج كثيرة في الوقت نفسه، إذا أمكنك تقليل المساحة التي يتطلبها كل برنامج، فقد تَتِمَكَّن من تشغيل برامج أكثر على نفس الخادم، مما يُقلِّل من تكلفة الطاقة والعتاد المطلوبة.

كانت هذه بعض الأسباب التي توضح أهمية الاطلاع على متطلبات الخوارزميات المتعلقة بالذاكرة. يمكنك التوسع أكثر في الموضوع بقراءة توثيق خوارزميات الترتيب في توثيق موسوعة حسوب.



أكبر موقع توظيف عن بعد في العالم العربي

ابحث عن الوظيفة التي تحقق أهدافك وطموحاتك
المهنية في أكبر موقع توظيف عن بعد

تصفح الوظائف الآن

أحدث إصدارات أكاديمية حسوب

