**Student Assessment Submission and Declaration**

When submitting evidence for assessment, each student must sign a declaration confirming that the work is their own.

|  |  |  |  |
| --- | --- | --- | --- |
| Student name: Shahed Montaser | | Assessor name:  **Tariq Aburashed** | |
| Issue date (1St Submission):  **12/5/2024** | Submission date (1St Submission):  **20/06/2024** | | Submitted on: |
| In case of resubmission | | | |
| Issue date (2nd Submission):  23/6/2024 | Submission date (2nd Submission):  25/6/2024 | | Submitted on: |
| Programme: Higher National Diploma in Cloud computing – Cloud Software Development Higher National Diploma in Computing | | | |
|  | | | |
| Assignment number and title: 1 implementing and using data structures and algorithms | | | |

Emergency Room System (ER)

1.

للتعامل مع المرضى في غرفة الطوارئ (ER) باتباع نهج First In First Out (FIFO)، فإن بنية البيانات الأكثر ملاءمة هي queue. تعمل قائمة الانتظار على أساس FIFO، حيث يكون العنصر الأول المضاف هو أول عنصر تتم إزالته. يتوافق هذا تمامًا مع متطلبات تقديم الخدمة للمرضى بناءً على وقت وصولهم.

استراتيجية التنفيذ

للتعامل مع المرضى في غرفة الطوارئ، يمكن تنفيذ queue حيث يمثل كل عنصر مريضًا. يمكن تمثيل كل مريض بهيكل أو فئة تتضمن المعلومات ذات الصلة مثل الاسم والمشكلة الصحية ووقت الوصول وأي تفاصيل أخرى ضرورية.

A purple and white rectangular object with white text

Description automatically generated

كيف يساعد في ER

المعالجة المنظمة: تضمن قائمة الانتظار معالجة المرضى بالترتيب الذي وصلوا به، وهو أمر عادل وسهل الإدارة.

أوقات الانتظار المتوقعة: يمكن للمرضى رؤية موقعهم في قائمة الانتظار، مما يمنحهم فكرة عن وقت انتظارهم.

قابلية التوسع: يمكن أن تنمو قائمة الانتظار ديناميكيًا مع وصول المزيد من المرضى وتتقلص مع تقديم الخدمة للمرضى.

سهولة التنفيذ: تتميز قوائم الانتظار بسهولة تنفيذها ودمجها في أنظمة أكبر.

Enqueue (الوصول): إضافة مريض إلى الجزء الخلفي من قائمة الانتظار.

Dequeue (الخدمة): إزالة المريض من مقدمة قائمة الانتظار.

يضمن هيكل البيانات البسيط والفعال هذا قدرة غرفة الطوارئ على التعامل مع المرضى بكفاءة ونزاهة بناءً على أوقات وصولهم.

2.3.4:

للتعامل مع طلبات المرضى في غرفة الطوارئ باستخدام قائمة الانتظار، نحتاج إلى تحديد بنية بيانات قائمة الانتظار بالسمات والعمليات المناسبة.

صفات للمريض

الاسم: اسم المريض.

العمر: عمر المريض.

الجنس: جنس المريض.

ID: معرف للمريض.

القضية الصحية: وصف موجز للقضية الصحية.

لقد استخدمت في التطبيق البرمجي Double linkedlist لاستيعاب اي عدد من المرضى ولسهولة التعامل.

عمليات queue:

Enqueue (إضافة مريض): إضافة مريض جديد إلى الجزء الخلفي من قائمة الانتظار.

Dequeue (خدمة المريض): إزالة المريض وإعادته من مقدمة قائمة الانتظار.

peek (المريض التالي): عرض المريض في مقدمة قائمة الانتظار دون إزالته.

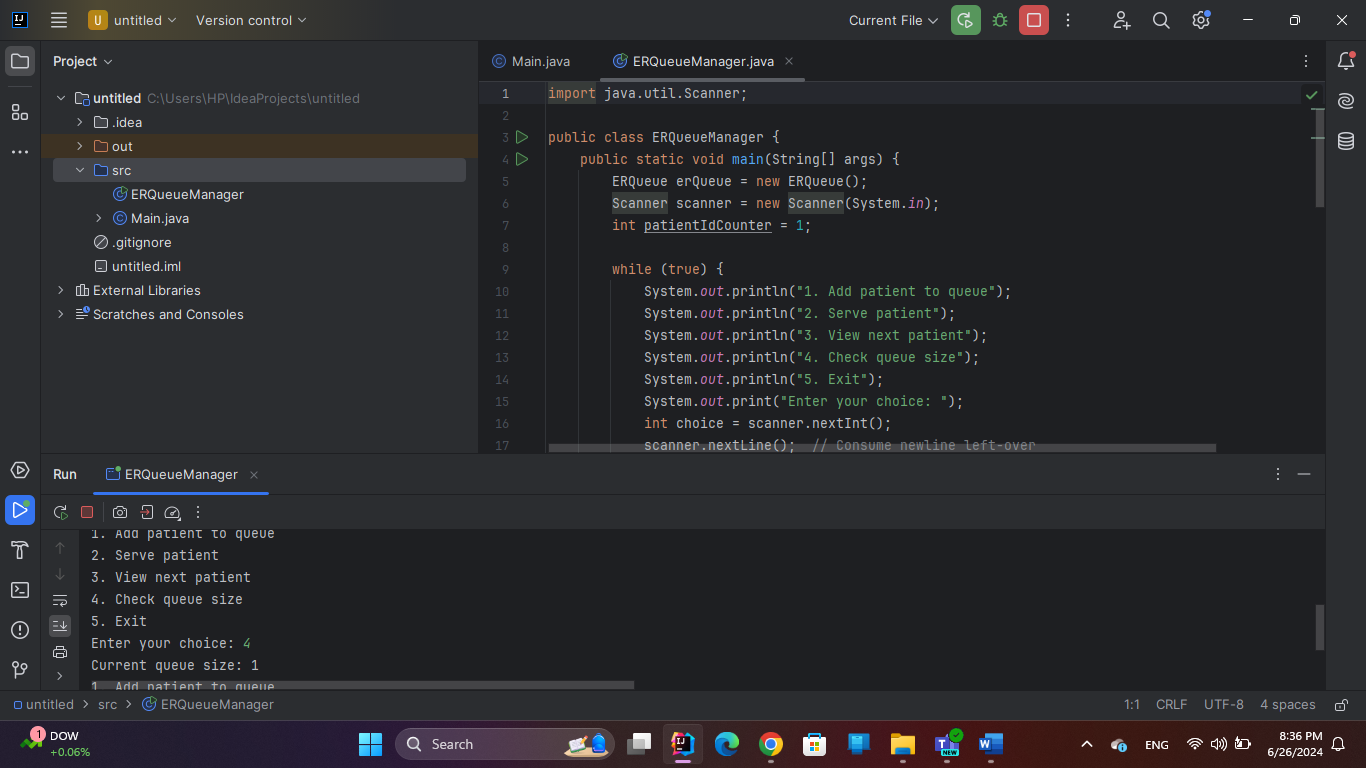
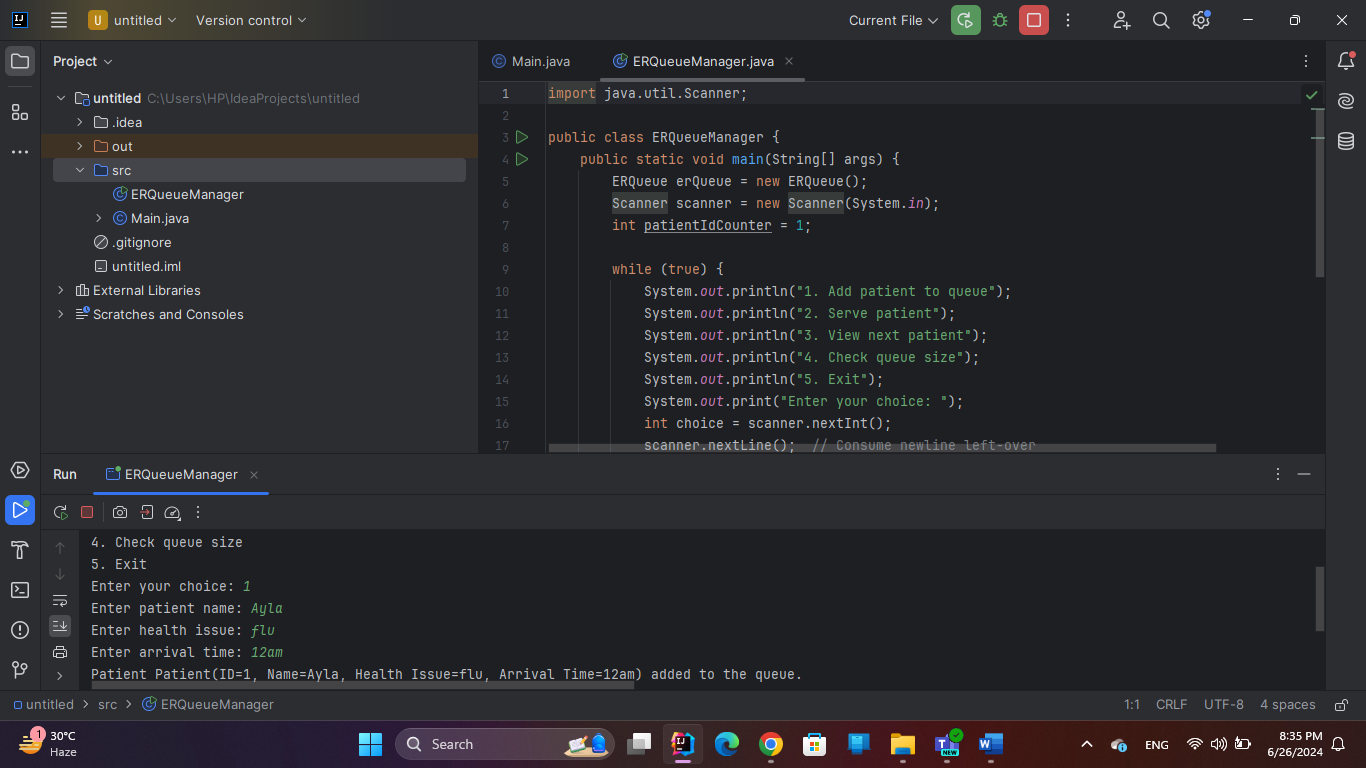
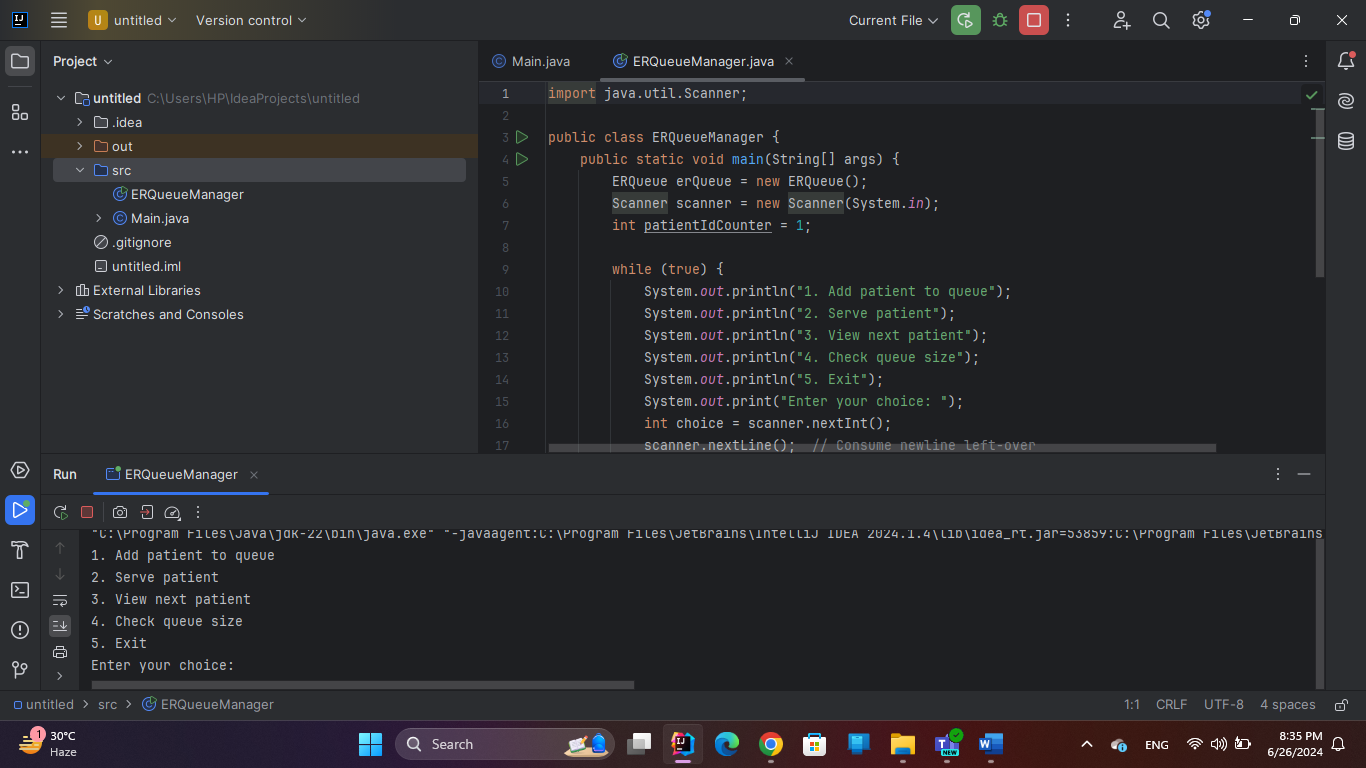
IsEmpty: تحقق مما إذا كانت قائمة الانتظار فارغة.

size: احصل على العدد الحالي للمرضى في قائمة الانتظار.

لا تتضمن العمليات is full لانني قمت باستعمال ال Double linkedlist وليس لها حجم محدد.

Practical code:





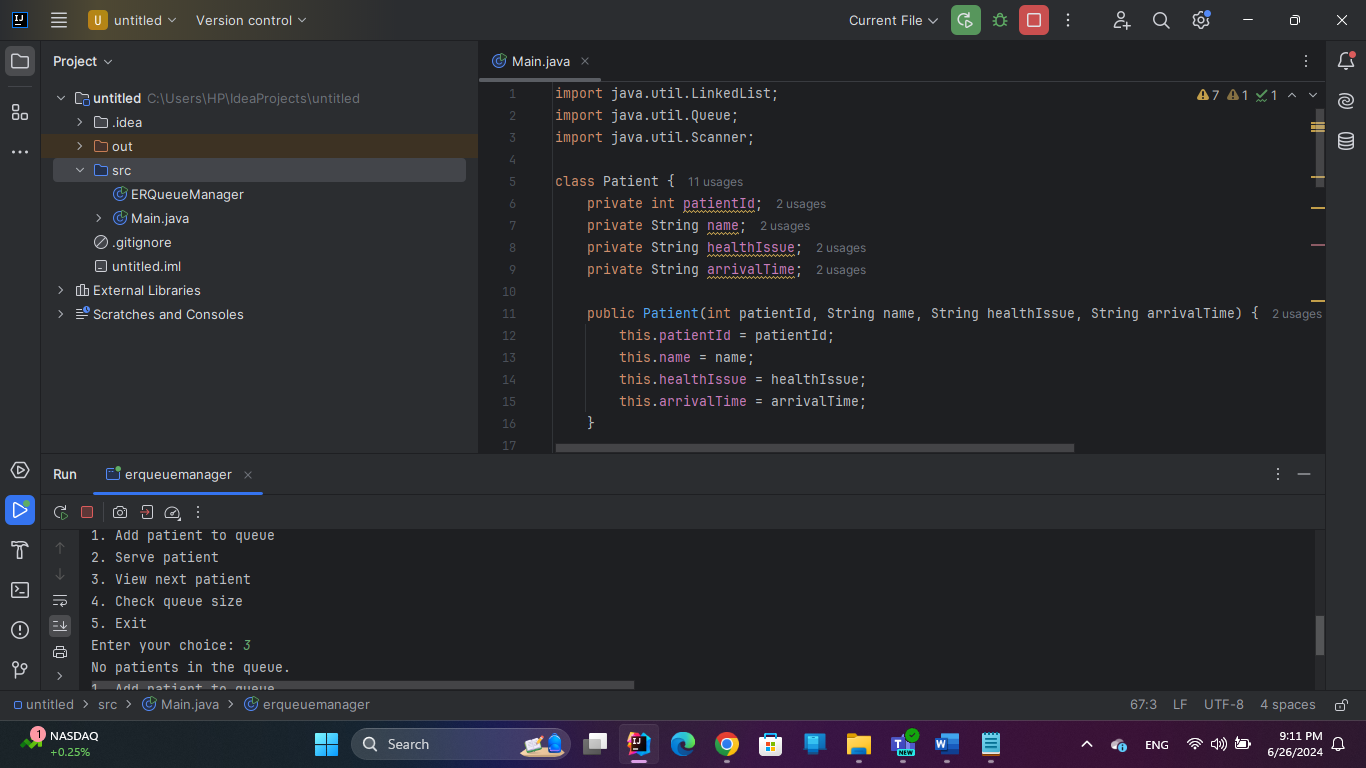
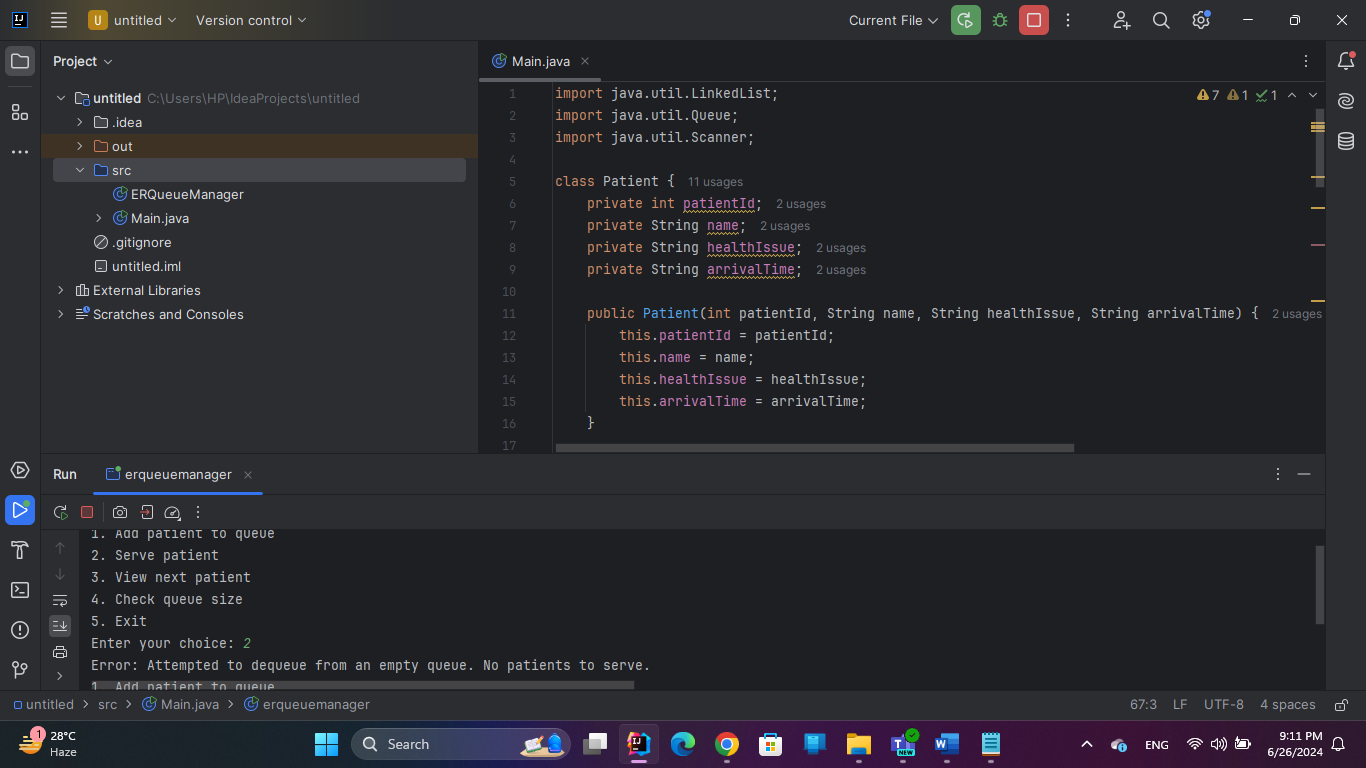
معالجة خطأ قائمة الانتظار الفارغة

للتعامل مع الحالة التي يحاول فيها المستخدم إلغاء قائمة الانتظار الفارغة، فإننا نتحقق مما إذا كانت قائمة الانتظار فارغة قبل محاولة إلغاء الصف. إذا كان فارغًا، فإننا نقوم بتشغيل طريقة معالجة الأخطاء.

فحص ومعالجة قائمة الانتظار الفارغة

يتحقق الأسلوب isEmpty مما إذا كانت قائمة الانتظار فارغة. إذا أعادت isEmpty القيمة true أثناء عملية dequeue، فسيتم استدعاء الأسلوب HandleEmptyQueueError لمعالجة الخطأ.

Practical work:





التعامل الفعال مع طلبات مرضى الطوارئ

تتعامل قائمة الانتظار المطبقة بشكل فعال مع طلبات مرضى الطوارئ من خلال ضمان:

الإدارة المنظمة: تتم خدمة المرضى حسب ترتيب وصولهم، باتباع نهج FIFO.

الكفاءة التشغيلية: تبسيط إدارة تدفق المرضى وتقليل الأخطاء.

خدمة يمكن التنبؤ بها: يمكن للمرضى والموظفين معرفة من هو التالي بسهولة، مما يؤدي إلى تحسين التواصل والرضا.

قابلية التوسع: يمكن لقائمة الانتظار التعامل مع أعداد مختلفة من المرضى بكفاءة.

يضمن هذا التنفيذ قدرة قسم الطوارئ على التعامل مع المرضى بكفاءة ونزاهة بناءً على أوقات وصولهم، مع توفير معالجة قوية للأخطاء في حالات الحافة مثل محاولة إلغاء قائمة الانتظار من قائمة انتظار فارغة.

كيف تتعامل قائمة الانتظار مع طلبات مرضى الطوارئ بشكل فعال

تقوم قائمة الانتظار المطبقة في كود Java المقدم بإدارة طلبات مرضى الطوارئ بشكل فعال باستخدام نهج (FIFO). فيما يلي شرح تفصيلي لكيفية عمل عمليات قائمة الانتظار وسيناريو مثال يوضح فعاليتها.

لقد استخدمت في التطبيق البرمجي Double LinkedList لاستيعاب اي عدد من المرضى ولسهولة التعامل.

عمليات queue

Enqueue (إضافة مريض): إضافة مريض جديد إلى الجزء الخلفي من queue.

الطريقة: Enqueue (اضافة مريض)

السلوك: يتم إلحاق المريض بنهاية queue، مما يضمن تقديم الخدمة له بعد جميع المرضى الذين وصلوا قبله.

Dequeue (ازالة مريض): إزالة المريض وإخراجه من مقدمة queue.

الطريقة: dequeue()

السلوك: تتم إزالة المريض الذي كان في مقدمة queue (والذي كان ينتظر لفترة أطول) وتقديم الخدمة له. إذا كانت queue فارغة، يتم عرض رسالة خطأ.

Peek (المريض التالي): لعرض المريض في مقدمة queue دون إزالته.

الطريقة: Peek ()

السلوك: إعادة المريض إلى مقدمة queue، مما يسمح للموظفين بمعرفة من سيتم تقديم الخدمة له بعد ذلك.

IsEmpty: التحقق مما إذا كانت queue فارغة.

الطريقة: isempty ()

السلوك: يُرجع true إذا كانت queue فارغة، ويُرجع خطأً بخلاف ذلك.

الحجم: يحصل على العدد الحالي للمرضى في queue.

الطريقة: الحجم ()

السلوك: إرجاع عدد المرضى الموجودين حاليًا في قائمة الانتظار.

مثال:

فيما يلي سيناريو يوضح كيفية تعامل قائمة الانتظار مع طلبات المرضى في غرفة الطوارئ بشكل فعال:

الحالة الأولية

قائمة الانتظار فارغة.

الخطوة 1: وصول المرضى

تصل سارة في الساعة الثالثة فجرا مصابة بأزمة تنفسية.

Queue.enqueue(new Patient(1, "Sarah", "Asthma", "3:00 AM"));

queue: [Sarah]

يصل سام الساعة 3:15 صباحًا بارتفاع السكري.

Queue.enqueue(new Patient(2, "Sam", "Diabetes", "3:15 AM"));

queue: [Sam، Sarah]

يصل آدم الساعة 3:20 صباحًا وهو يعاني من كسر في الكاحل.

Queue.enqueue(new Patient(3, "Adam", " Broken ankle", "3:20 AM"));

queue: [Adam، Sam، Sarah]

الخطوة الثانية: خدمة المرضى

خدمة سارة:

Queue.dequeue();

الإخراج: تتم خدمة المريضة سارة.

queue: [Adam، Sam]

المريض التالي:

Queue.peek();

المخرجات: المريض التالي: المريض (المعرف=2، الاسم=سام، المشكلة الصحية=السكري، وقت الوصول=3:15 صباحًا)

queue: [Adam، Sam]

خدمة سام:

Queue.dequeue();

الإخراج: يتم تقديم الخدمة للمريض سام.

queue: [Adam]

خدمة آدم:

Queue.dequeue();

الإخراج: يتم تقديم الخدمة للمريض آدم.

queue: []

الخطوة 3: محاولة التقديم من queue فارغة

Queue.dequeue();

الإخراج: خطأ: جرت محاولة إلغاء queue من queue فارغة. لا يوجد مرضى للخدمة.

queue: []

يضمن هذا التنفيذ قدرة غرفة الطوارئ على التعامل مع المرضى بكفاءة ونزاهة بناءً على أوقات وصولهم. يوفر واجهة تفاعلية لإضافة المرضى وخدمة المرضى وعرض المريض التالي والتحقق من حجم queue. يتم أيضًا تضمين معالجة الأخطاء عند محاولة إلغاء queue من queue فارغة، مما يجعل النظام قويًا وموثوقًا.

5:

تقييم مدى تعقيد العمليات في قائمة انتظار المرضى

دعونا نقيم مدى تعقيد العمليات على قائمة انتظار المرضى التي يتم تنفيذها باستخدام Double LinkedList (Java).

سنقوم بتقييم العمليات: enqueue، وdequeue، وgetFirst، وgetLast، وisEmpty، مع التركيز على تعقيدها المقارب وعملية الحساب.

1. Enqueue:

العملية: إضافة مريض إلى نهاية قائمة الانتظار.

Time Complexity:

Average Time Complexity: 𝑂(1)

Amortized Time Complexity:𝑂(1)

Worst-case Time Complexity: 𝑂(𝑛)

السبب: في التنفيذ القائم على المصفوفة، عادةً ما تستغرق إضافة عنصر إلى النهاية وقتًا ثابتًا. ومع ذلك، عندما يكون المصفوفة ممتلئة، يجب تغيير حجمها (مضاعفتها عادةً)، وهو ما يتضمن نسخ جميع العناصر إلى المصفوفة الجديدة. يستغرق تغيير الحجم هذا O(n) الوقت، ولكن يحدث ذلك بشكل غير متكرر. ولذلك، فإن التعقيد الزمني هو O(1).

Space Complexity: O(1)

السبب: تتطلب كل عملية قائمة انتظار مساحة إضافية ثابتة، باستثناء أثناء تغيير الحجم، والذي يتضمن تخصيص مصفوفة جديدة.

2. Dequeue:

العملية: إزالة المريض من مقدمة قائمة الانتظار.

Time Complexity: O(n)

السبب: في قائمة الانتظار القائمة على المصفوفة، تتضمن إزالة العنصر الأول نقل جميع العناصر اللاحقة موضعًا واحدًا إلى اليسار، مما يؤدي إلى تعقيد الوقت الخطي.

Space Complexity: O(1)

السبب: تعقيد المساحة لعملية قائمة الانتظار ثابت حيث لا توجد حاجة لمساحة إضافية.

3. Get First:

العملية: استرجاع أول مريض في قائمة الانتظار دون إزالتهم.

Time Complexity: O(1)

السبب: تتضمن عملية getFirst الوصول إلى العنصر الأول من المصفوفة، وهي عملية ذات وقت ثابت.

Space Complexity: O(1)

السبب: تعقيد المساحة ثابت حيث لا توجد مساحة إضافية مطلوبة لهذه العملية.

4. GetLast:

العملية: استرجاع آخر مريض في قائمة الانتظار دون إزالتهم.

Time Complexity: O(1)

السبب: في قائمة الانتظار القائمة على المصفوفة، يكون الوصول إلى العنصر الأخير عملية ثابتة الوقت حيث يمكننا فهرسته مباشرة.

Space Complexity: O(1)

السبب: يظل تعقيد المساحة ثابتًا حيث لا يلزم وجود مساحة إضافية لهذه العملية.

5. IsEmpty:

العملية: التحقق مما إذا كانت قائمة الانتظار فارغة.

Time Complexity: O(1)

السبب: تتحقق العملية isEmpty مما إذا كان حجم المصفوفة صفرًا، وهي عملية ذات وقت ثابت.

Space Complexity: O(1)

السبب: تعقيد المساحة ثابت حيث لا توجد حاجة إلى مساحة إضافية لإجراء هذا الفحص.

ملخص:

Enqueue: O(1) التعقيد الزمني في كل من القائمة المرتبطة وتطبيقات المصفوفة الديناميكية.

Dequeue: O(1) التعقيد الزمني في القائمة المرتبطة، O(n) في الصفيف الديناميكي.

GetFirst: O(1) التعقيد الزمني في كلا التطبيقين.

GetLast: تعقيد الوقت O(n) في القائمة المرتبطة، O(1) في المصفوفة الديناميكية.

IsEmpty: O(1) التعقيد الزمني في كلا التطبيقين.

توضح هذه التحليلات أن تنفيذ Linked list أكثر كفاءة لعمليات إلغاء الانتظار، بينما توفر المصفوفة الديناميكية أداءً أفضل للحصول على العنصر الأخير. يضمن كلا التطبيقين أن عمليات قائمة الانتظار الشائعة مثل enqueue وgetFirst وisEmpty تعمل بكفاءة.

6.

A diagram of a patient

Description automatically generated

توضيح

في البداية تكون queue فارغة، يأتي مريض حالته غير حرجة يتم ادخاله الى اول موقع في queue كالوضع الطبيعي. ثم يأتي مريض جديد، اذا كان بحالة حرجة فإنه تتم اضافته الى اول موقع في ال queue اما اذا كان مريض غير حرج تتم اضافته بشكل طبيعي الى اول موقع فارغ.

بعد ذلك تتم معالجة المرضى من اول موقع فالتالي. حيث سيكون مريض الحالة الحرجة في بداية ال queue ويتم التعامل معهم واحدا تلو الاخر بالترتيب الى ان تصبح فارغة.

7.

تقييم مدى ملاءمة Stack لسيناريو الطوارئ

تعمل Stack على مبدأ آخر ما يدخل يخرج أولاً (LIFO)، مما يعني أن العنصر الذي تمت إضافته مؤخرًا هو أول عنصر تتم إزالته. ويختلف هذا بشكل أساسي عن نهج "الداخل، يخرج أولاً" (FIFO) المطلوب للتعامل مع المرضى غير ذوي الحالات الحرجة في غرفة الطوارئ (ER). إن استخدام Stack يعني أن آخر مريض يصل سيكون أول مريض يتم علاجه، وهو أمر غير مناسب للمرضى غير ذوي الحالات الحرجة الذين يجب تقديم الخدمة لهم حسب ترتيب وصولهم. بالإضافة إلى ذلك، لا توفر Stack أي آلية لتحديد أولويات المرضى الحرجة.

نوع البيانات المجردة (ADT) ل Stack

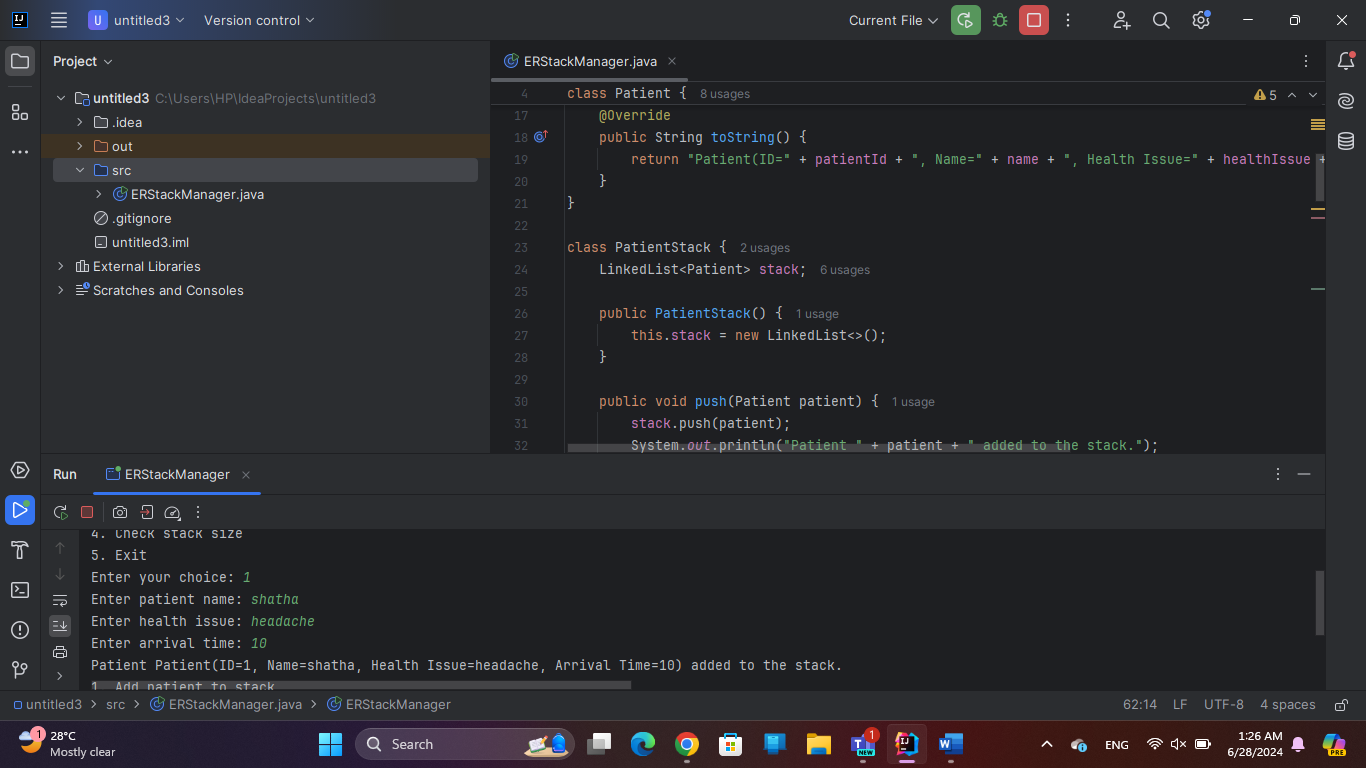
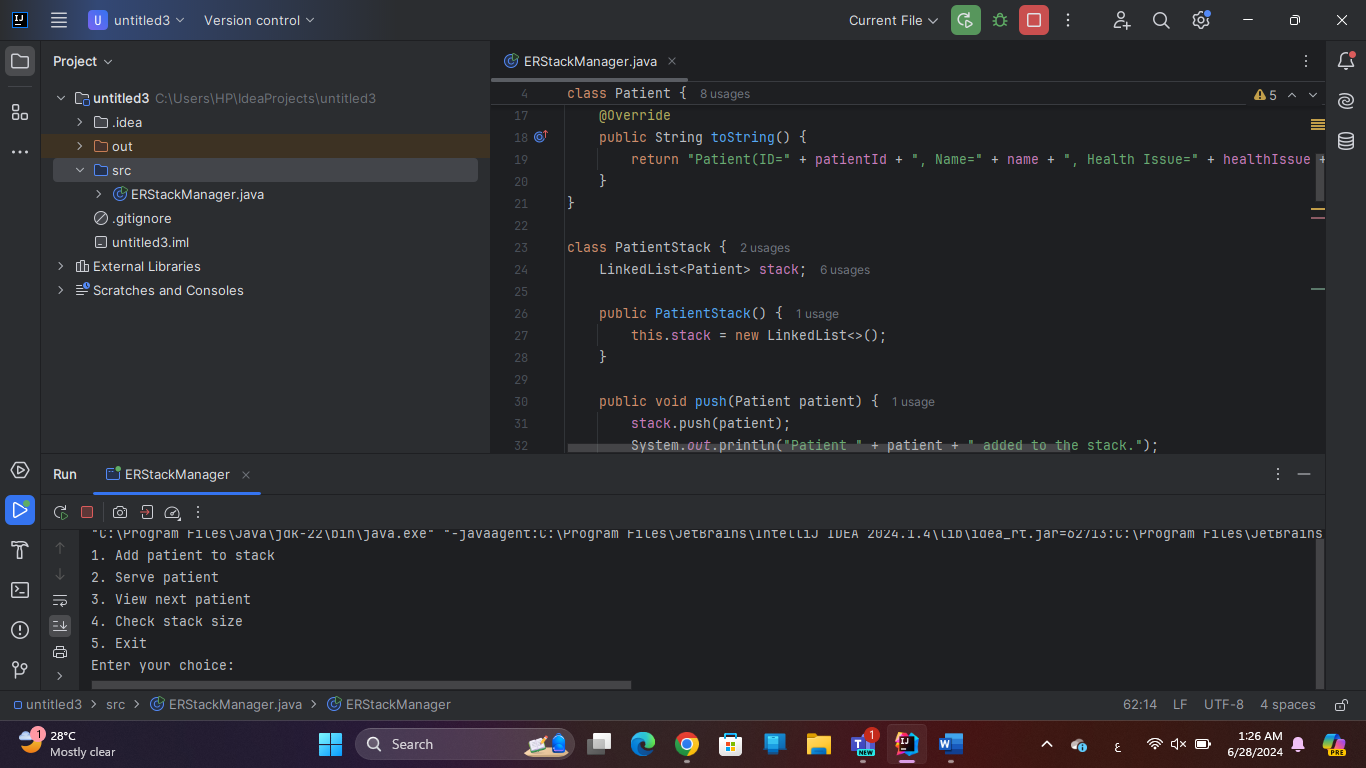
Push: إضافة عنصر إلى أعلى Stack.

Pop: إزالة العنصر من أعلى Stack.

Peek/Top: استرداد العنصر الموجود في الأعلى دون إزالته.

IsEmpty: يتحقق مما إذا كان Stack فارغًا.

التطبيق:





تعقيد الوقت

Push:

O(1) - تستغرق إضافة عنصر إلى Stack وقتًا ثابتًا.

Pop:

O(1) - تستغرق إزالة العنصر العلوي من Stack وقتًا ثابتًا.

Peek/Top:

O(1) - يستغرق استرداد العنصر العلوي وقتًا ثابتًا.

IsEmpty:

O(1) - يستغرق التحقق من كون Stack فارغًا وقتًا ثابتًا.

Stack غير مناسب للتعامل مع طلبات مرضى الطوارئ نظرًا لطبيعة LIFO الخاصة به. قد يتم دفع المرضى ذوي الحالات الحرجة إلى Stack ، ولكن يجب تقديم الخدمات للمرضى غير ذوي الحالات الحرجة حسب ترتيب الوصول، وهو ما لا تستطيع Stack التعامل معه بفعالية.

Medical Information System

1.1.

Understanding Stack Operations in Recursive Function Calls

التحليل والشرح

عند استدعاء recursive function، يؤدي كل استدعاء للوظيفة إلى new entry (or frame) في stack الاستدعاءات. يحتوي هذا الإطار على المتغيرات المحلية للوظيفة وحالة التنفيذ. في هذه الحالة، يتم استدعاء function countDays بشكل متكرر حتى يتم تقليل الجرعة إلى الصفر أو إلى كمية ضئيلة.

يمكن شرح عمليات stack للتعامل مع recursive callsعلى النحو التالي:

Function Call: في كل مرة يتم فيها استدعاء accountDays، يتم دفع إطار جديد إلى stack الاستدعاءات.

Function execution: يتم تنفيذ الوظيفة، وتنفيذ منطقها (في هذه الحالة، تقليل الجرعة).

Recursive Call: قبل أن تتمكن function من إرجاع نتيجة، قد تستدعي نفسها بشكل متكرر، وتدفع إطارًا آخر إلى stack.

الحالة الأساسية: بمجرد الوصول إلى الحالة الأساسية (عندما تكون الجرعة صغيرة بما فيه الكفاية)، تبدأ function في العودة.

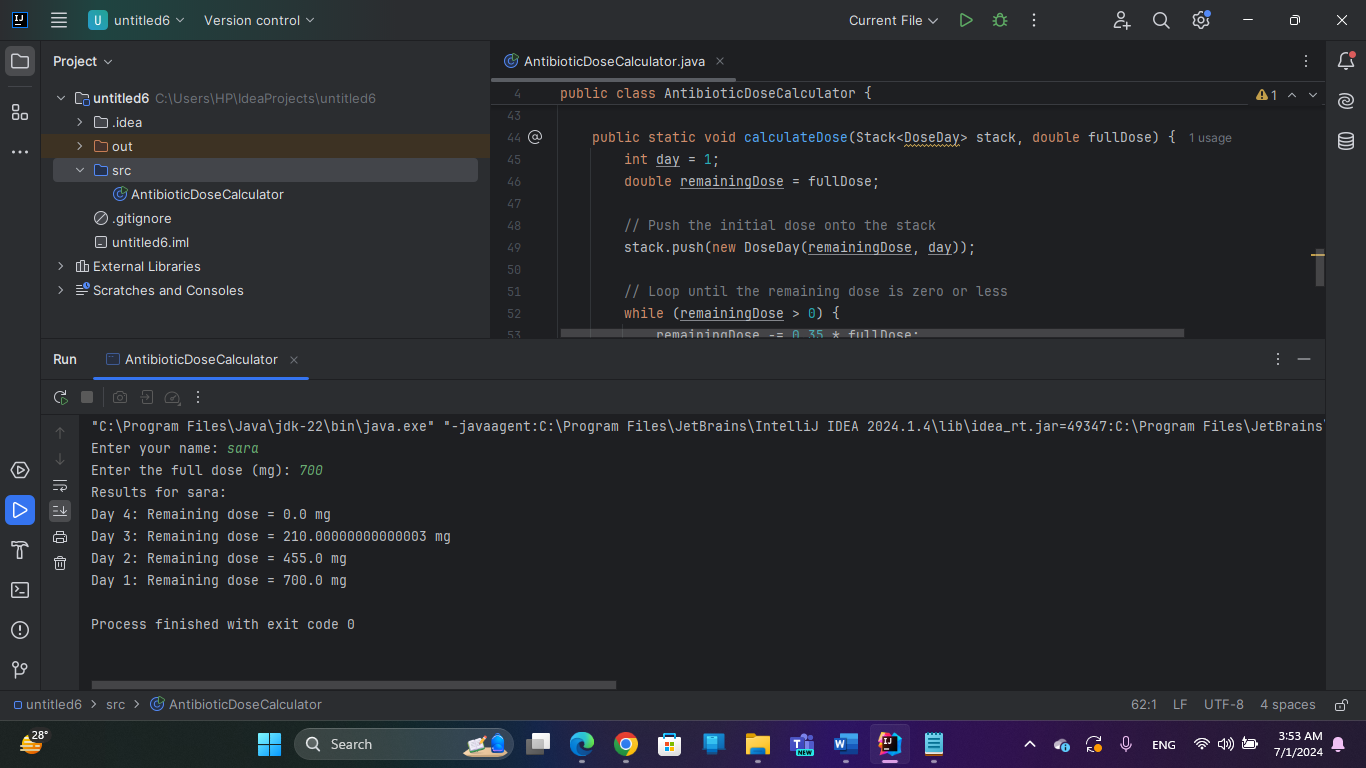
Function Return: تؤدي كل عملية إرجاع إلى إخراج الإطار المقابل من stack، مما يؤدي إلى فك stack.

تعقيد الوقت

التعقيد الزمني لهذه الوظيفة هو O(n)، حيث n هو عدد recursive calls اللازمة لتقليل الجرعة إلى الصفر. تقوم كل استدعاء بإنشاء وإزالة إطار على stack.

تتعامل stacks مع recursive function calls من خلال الحفاظ على إطار منفصل لكل استدعاء. يضمن ذلك الحفاظ على حالة الوظيفة عبر الاستدعاءات المتكررة ويمكن استعادتها عندما تعود الوظيفة. في هذا المثال، تقوم stack بإدارة recursive calls بشكل فعال لحساب عدد الأيام التي يحتاجها المريض لتناول المضاد الحيوي.

المريض يحتاج الى 3 ايام لانهاء الجرعة الكاملة 700mg.





1.2.

مقارنة الأداء

QuickSort

Best Case (Sorted Array): O(n log n)

Worst Case (Reverse Sorted Array): O(n^2)

Average Case (Random Array): O(n log n)

Partial Sorted Array: O(n log n)

يعمل QuickSort بكفاءة مع تعقيد زمني قدره O(n log n) في أفضل الحالات ومتوسطها. ومع ذلك، في أسوأ الحالات، يتحلل التعقيد الزمني إلى O(n^2). يُفضل QuickSort بشكل عام لمجموعات البيانات الكبيرة نظرًا لكفاءته المتوسطة.

SelectionSort

Best Case (Sorted Array): O(n^2)

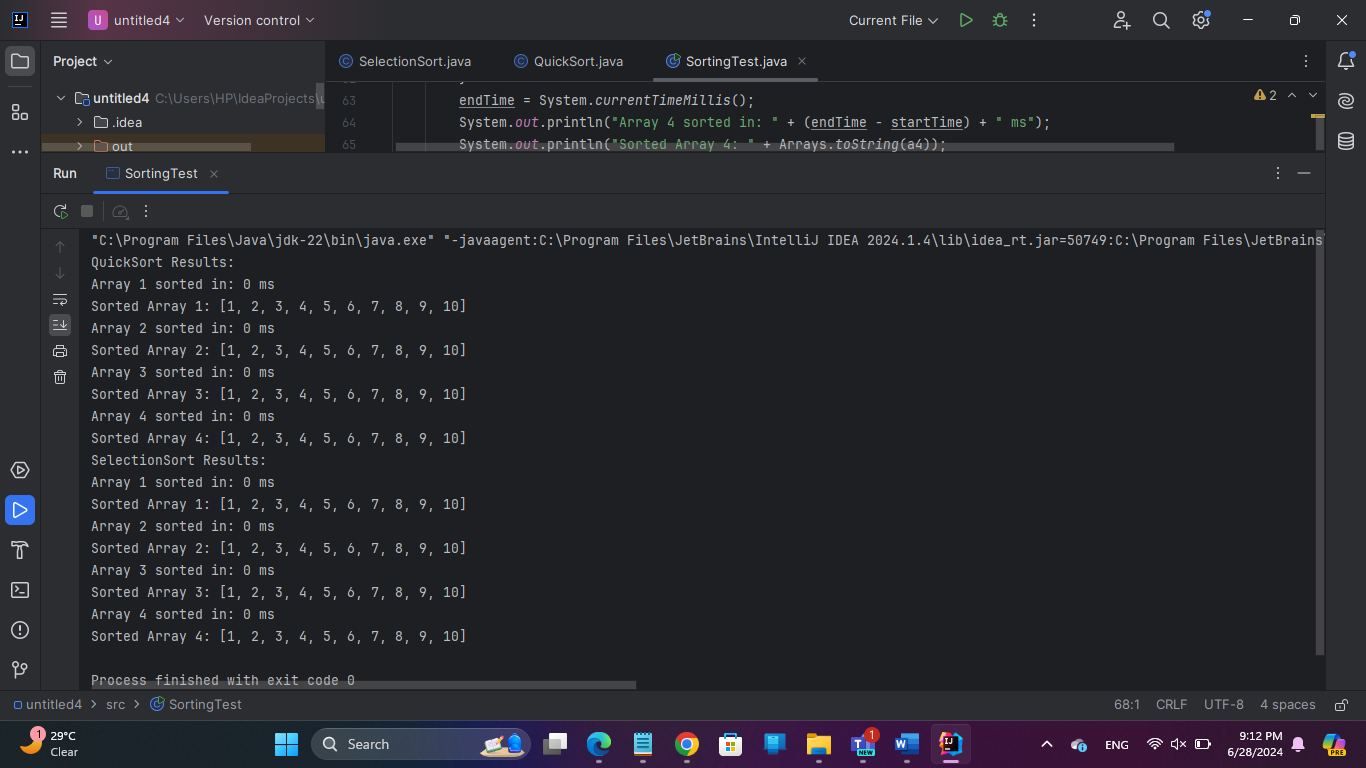
Worst Case (Reverse Sorted Array): O(n^2)

Average Case (Random Array): O(n^2)

Partial Sorted Array: O(n^2)

يتميز SelectionSort بتعقيد زمني قدره O(n^2) لجميع الحالات، مما يجعله أقل كفاءة من QuickSort لمجموعات البيانات الكبيرة. إنها بسيطة ولها أداء يمكن التنبؤ به، ولكنها بشكل عام غير مناسبة للمصفوفات الكبيرة.

استنادًا إلى تحليل الأداء، عادةً ما يكون QuickSort هو الخيار الأفضل نظرًا لتعقيده الزمني المتوسط ​​للحالة الذي يبلغ O(n log n). إنه يؤدي أداءً أفضل بكثير من SelectionSort، خاصة بالنسبة لمجموعات البيانات الكبيرة. ومع ذلك، في السيناريوهات التي تكون فيها مجموعة البيانات صغيرة أو تكون البساطة مفضلة، فقد يتم أخذ SelectionSort في الاعتبار.





شرح الإخراج

يختبر الكود كلا من QuickSort وSelectionSort على أربع صفائف مختلفة (مرتبة، ومرتبة عكسيًا، وعشوائية، ومرتبة جزئيًا).

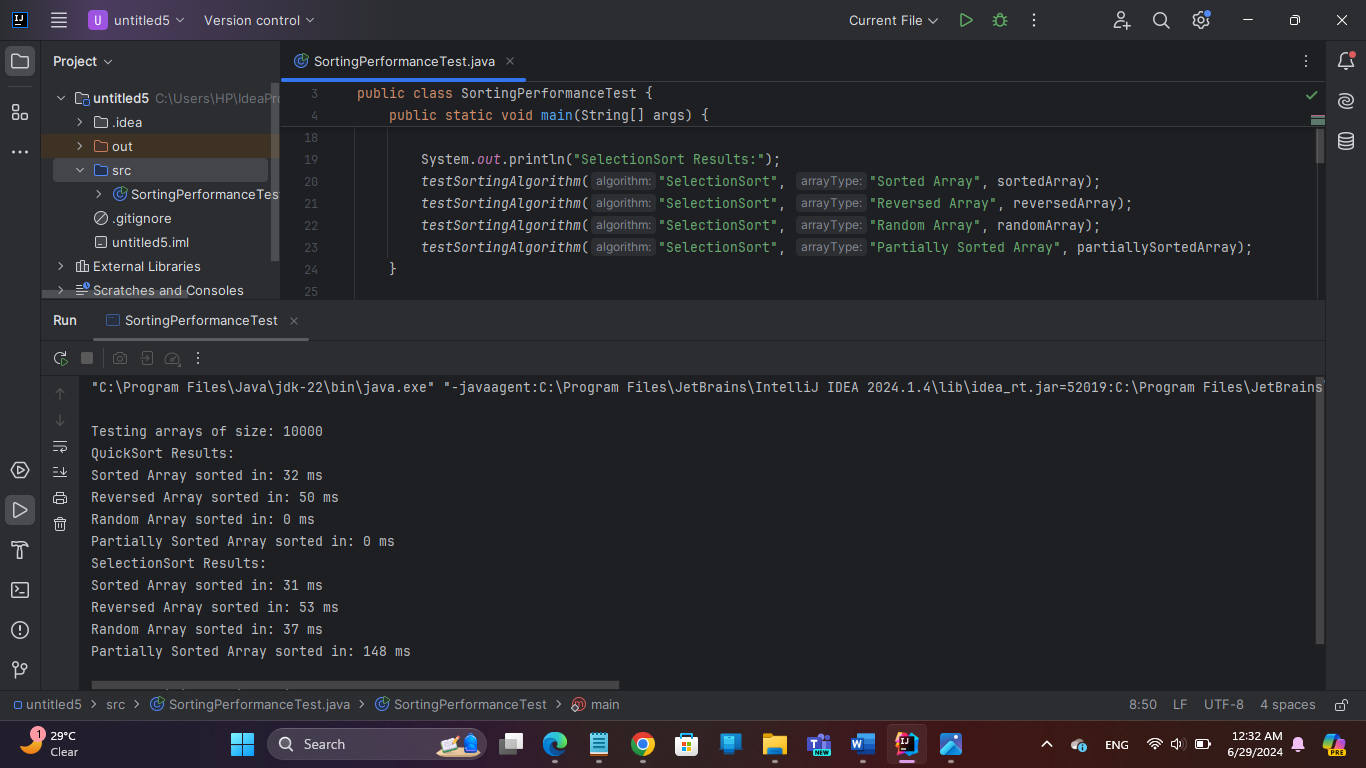
إنه يقيس الوقت المستغرق لفرز كل مجموعة بالمللي ثانية كانت النتيجة 0 مللي ثانية لان وقت التنفيذ اسرع من ذلك يمكن قياسه بالنانو ثانية.

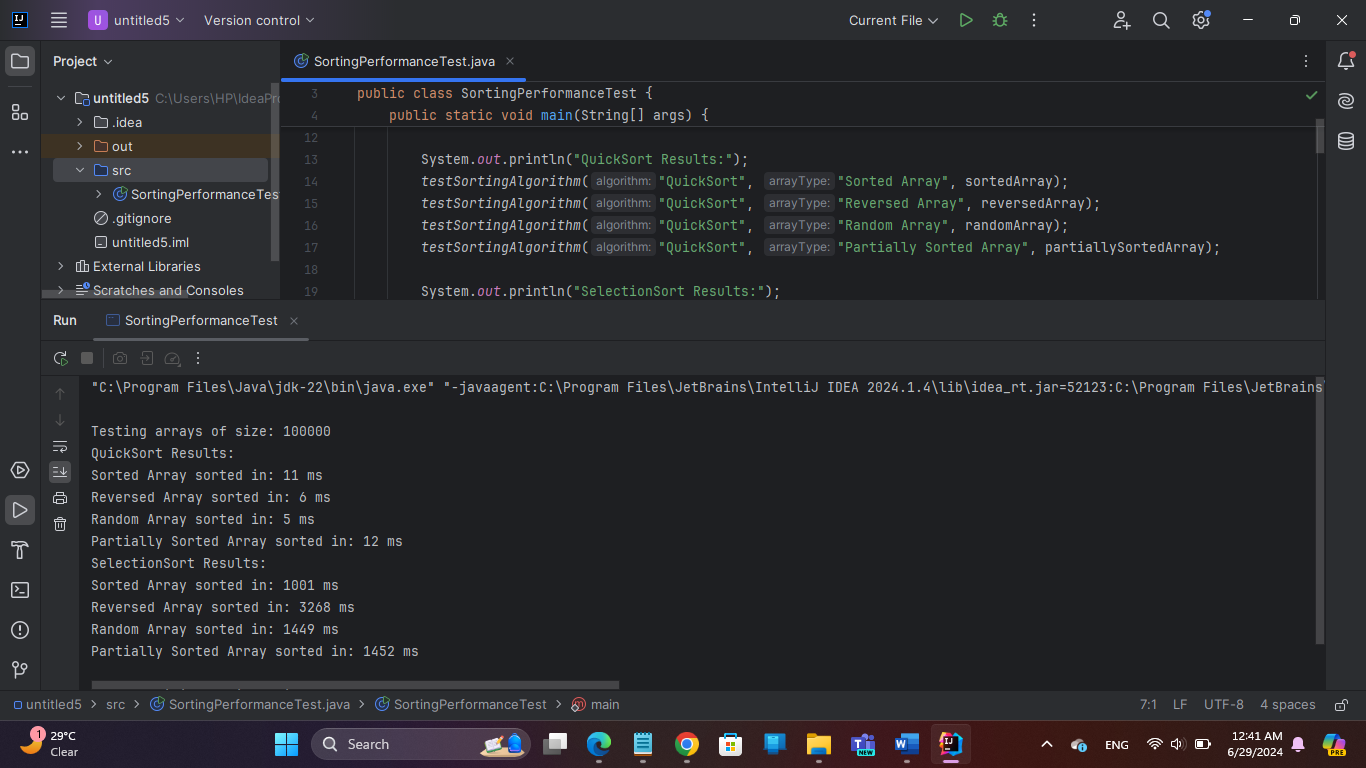
يقوم بطباعة المصفوفات التي تم فرزها مع الوقت المستغرق لكل عملية فرز.

عند تشغيل فئة SortingTest، سيتم عرض الوقت المستغرق لفرز كل مصفوفة والمصفوفات التي تم فرزها نفسها. بهذه الطريقة، يمكنك مقارنة أداء QuickSort وSelectionSort لأنواع مختلفة من صفائف الإدخال.

الان سنقوم بتطبيق كلتا الخوارزميات على احجام مدخلات مختلفة، لكن كل حجم مدخلات على حدى نظرا لان البرنامج لن يتحمل كمية البيانات ولن يعطي ناتج.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sorting Algorithm** | **Input Size** | **Sorted** | **Reversed** | **Random** | **Partially Sorted** |
| **Selection Sort** | **N = 10 000** | 31 ms | 53 ms | 37 ms | 128 ms |
| **N = 100 000** | 1001 ms | 3268 ms | 1449 ms | 1452 ms |
| **N = 1000 000** | 645864ms | 1375437ms | 2145304ms | 1093252ms |
| **Quick Sort** | **N = 10 000** | 32 ms | 50 ms | 0 ms | 0 ms |
| **N = 100 000** | 11 ms | 6 ms | 5 ms | 12 ms |
| **N = 1000 000** | 97546 ms | 198997 ms | 79 ms | 63 ms |





A screenshot of a computer

Description automatically generated



2.1.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sorting Algorithm** | **Best Case** | **Worst Case** | **Average Case** |
| **Quick**  **Sort** | O(nlogn) | O(n2) | O(nlogn) |
| **Selection**  **Sort** | O(n2) | O(n2) | O(n2) |

2.2

يتمتع quick sort بأداء أفضل من selection sort لمجموعات البيانات الكبيرة. وذلك لأن التعقيدات الزمنية المتوسطة وأفضل الحالات ل quick sort هي O(nlogn)، وهي أكثر كفاءة بشكل ملحوظ من selection sort O(n2) التعقيد الزمني للحالات المتوسطة والأسوأ.

2.3

يوفر التحليل المقارب فهمًا عالي المستوى لكفاءة الخوارزمية من خلال التركيز على معدل نمو وقت تشغيلها بالنسبة لحجم الإدخال. وإليك كيف يساعد:

قابلية التوسع: تساعد في تحديد كيفية قياس الخوارزمية مع زيادة أحجام المدخلات. على سبيل المثال، ستتعامل خوارزمية O(nlogn) مثل Quick Sort مع مجموعات البيانات الأكبر بشكل أكثر كفاءة من خوارزمية O(n2) مثل Selection Sort.

المقارنة: تسمح بمقارنة الخوارزميات المختلفة بغض النظر عن الأجهزة أو تفاصيل التنفيذ. من خلال معرفة التعقيدات، يمكن للمرء أن يقرر أي الخوارزمية من المتوقع أن تؤدي أداءً أفضل لمشكلة معينة.

القدرة على التنبؤ: تساعد في التنبؤ بأداء الخوارزمية للمدخلات الكبيرة. على سبيل المثال، معرفة أن Quick sort يتمتع بمتوسط ​​تعقيد يبلغ O(nlogn) يمنح الثقة بأنه سيتعامل مع المدخلات الأكبر بشكل جيد إلى حد معقول.

2.4.

التعقيد الزمني: يقيس كيفية زيادة وقت تشغيل الخوارزمية مع حجم الإدخال. نقوم بتقييم ذلك من خلال تحليل التعقيد الزمني للخوارزمية على سبيل المثال، O(nlogn) لQuick sort مقابل O(n2) لSelection sort.

تعقيد المساحة: يقيس مقدار الذاكرة التي تحتاجها الخوارزمية بالنسبة لحجم الإدخال. لا ينبغي للخوارزمية الفعالة أن تعمل بسرعة فحسب، بل يجب أيضًا أن تستخدم الذاكرة بحكمة.

على سبيل المثال:

يحتوي Quick sort على تعقيد مساحة يبلغ O(logn) نظرًا لطبيعته المتكررة (المساحة المطلوبة لstack الاستدعاءات).

يتميز Selection sort بتعقيد مساحة قدره O(1) لأنه يقوم بالفرز في مكانه دون الحاجة إلى ذاكرة إضافية.

ملخص سريع لجميع الاسئلة اعلاه:

التعقيد المقارب: يتفوق Quick sort بشكل عام على Selection sort للمصفوفات الأكبر حجمًا نظرًا لتعقيد O(nlogn) مقارنة بـ O(n2).

أفضل خوارزمية: استنادًا إلى نتائج الاختبار، يجب تفضيل Quick sort لمجموعات البيانات الكبيرة والحالات التي يكون فيها الأداء المتوسط ​​والأفضل مهمًا. يمكن مراعاة Selection sort لمجموعات البيانات الصغيرة أو عندما تكون البساطة أكثر أهمية.

قياس الكفاءة: التعقيد الزمني وتعقيد المساحة هما طريقتان أساسيتان لقياس الكفاءة. يتميز Quick sort بتعقيد زمني أفضل في المتوسط، بينما يتميز Selection sort بتعقيد مساحة ثابت ولكن تعقيد زمني أسوأ.

3.

لحل هذه المهمة، سنحتاج إلى تطبيق خوارزميات Dijkstra وBelman-Ford للعثور على أقصر مسار من الغرفة A إلى الغرفة K. فيما يلي خطوات كل خوارزمية.

التهيئة:

نضبط المسافة إلى عقدة البداية (A) على 0 وجميع المسافات الأخرى إلى ما لا نهاية.

نضع علامة على جميع العقد على أنها لم تتم زيارتها.

نقوم بتعيين العقدة الأولية (A) كالعقدة الحالية.

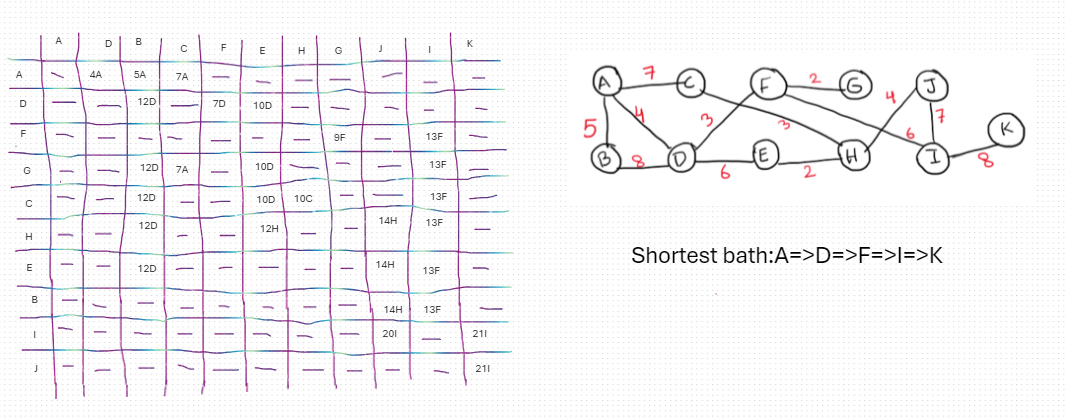
التنفيذ خطوة بخطوة:

بالنسبة للعقدة الحالية، نضع في الاعتبار جميع جيرانها الذين لم تتم زيارتهم ونحسب مسافاتهم المؤقتة خلال العقدة الحالية.

نقارن المسافة المؤقتة المحسوبة حديثًا بالقيمة المخصصة الحالية ونقوم بتعيين المسافة الأصغر.

نقوم بتمييز العقدة الحالية على أنها تمت زيارتها.

نحدد العقدة التي لم تتم زيارتها مع أصغر مسافة مؤقتة باعتبارها "العقدة الحالية" الجديدة ونكرر الأمر لنصل الى النتيجة النهائية:



Shortest path: 21m

4.1.

Advantages of Encapsulation in Object-Oriented Programming for Abstraction

التغليف هو مفهوم أساسي في البرمجة الشيئية (OOP) الذي يتضمن تجميع البيانات (السمات) والأساليب (الوظائف) التي تعمل على البيانات في وحدة واحدة تسمى فئة. كما أنه يقيد الوصول المباشر إلى بعض مكونات الكائن، وهو وسيلة لمنع التدخل العرضي وإساءة استخدام البيانات. فيما يلي المزايا الرئيسية:

النمطية: يتيح التغليف للمبرمج تجميع البيانات والوظائف ذات الصلة معًا، مما يسهل فهم البرنامج وصيانته وتصحيح أخطائه.

إخفاء المعلومات: من خلال تقييد الوصول إلى الأعمال الداخلية للكائن وكشف الواجهة الضرورية فقط، يخفي التغليف التعقيدات والتفاصيل الداخلية عن المستخدم، مما يقلل التعقيد.

قابلية إعادة الاستخدام: يمكن إعادة استخدام الكائنات أو الفئات المغلفة عبر برامج مختلفة أو أجزاء من نفس البرنامج، مما يعزز إمكانية إعادة استخدام التعليمات البرمجية.

المرونة وقابلية الصيانة: يمكن إجراء تغييرات على التعليمات البرمجية المغلفة بأقل تأثير على الأجزاء الأخرى من البرنامج. هذه المرونة تجعل صيانة التعليمات البرمجية وتحديثها أسهل.

تحسين الأمان: من خلال التحكم في الوصول إلى البيانات، يساعد التغليف في حماية البيانات من الوصول والتعديل غير المصرح به.

علاقة التغليف بالتعقيد الزمني والتعقيد المكاني

تتضمن العلاقة بين التغليف والتعقيدات (الزمان والمكان) لهياكل البيانات المختلفة فهم كيفية أداء الأساليب (الوظائف) المغلفة لهياكل البيانات هذه فيما يتعلق بالزمان والمكان. فيما يلي تفاصيل لبعض هياكل البيانات الشائعة:

1. المصفوفات الثابتة

التغليف: تقوم المصفوفات الثابتة بتغليف مجموعة من العناصر في كتلة متجاورة من الذاكرة.

تعقيد الوقت:

الوصول: O(1) - الوصول المباشر إلى الفهرس.

الإدراج/الحذف: O(n) - قد تكون هناك حاجة إلى تبديل العناصر.

تعقيد المساحة:

تم الإصلاح: يتم تحديد حجم المصفوفة عند الإنشاء، مما يؤدي إلى احتمال عدم استخدام مساحة أو نقص مساحة للعناصر الجديدة.

2. المصفوفات الديناميكية

التغليف: تدير المصفوفات الديناميكية مصفوفة أساسية ذات حجم ثابت، وتغيير حجم العناصر ونسخها إلى مصفوفة جديدة حسب الحاجة.

تعقيد الوقت:

الوصول: O(1) - الوصول المباشر إلى الفهرس.

الإدراج المطفأ: O(1) - معظم عمليات الإدراج هي O(1)، ولكن عمليات تغيير الحجم هي O(n).

الحذف: O(n) - قد تكون هناك حاجة إلى تبديل العناصر.

تعقيد المساحة:

ديناميكي: يخصص مساحة إضافية للتعامل مع النمو المستقبلي، ولكنه قد يؤدي إلى إهدار المساحة عندما يكون الاستخدام الفعلي أقل بكثير من المساحة المخصصة.

3. القوائم المرتبطة

التغليف: تقوم القوائم المرتبطة بتغليف العقد حيث تحتوي كل عقدة على بيانات ومرجع (رابط) إلى العقدة التالية.

تعقيد الوقت:

الوصول: O(n) - الوصول التسلسلي مطلوب.

الإدراج/الحذف: O(1) - بالنظر إلى إشارة إلى العقدة، يمكن إجراء الإدراج والحذف في وقت ثابت.

تعقيد الفضاء:

الحمل الزائد لكل عقدة: تتطلب كل عقدة مساحة إضافية لمرجع/مؤشر.

4. قوائم الانتظار

التغليف: تقوم قوائم الانتظار بتغليف مجموعة من العناصر التي تتبع مبدأ FIFO (الوارد أولاً يخرج أولاً).

تعقيد الوقت:

Enqueue: O(1) - إضافة عنصر في النهاية.

Dequeue: O(1) - إزالة عنصر من الأمام.

تعقيد الفضاء:

يعتمد على التنفيذ الأساسي (على سبيل المثال، القائمة المرتبطة أو المصفوفة الديناميكية).

5. الأكوام

التغليف: تقوم الأكوام بتغليف مجموعة من العناصر التي تتبع مبدأ LIFO (آخر ما يدخل، يخرج أولاً).

تعقيد الوقت:

Push: O(1) - إضافة عنصر في الأعلى.

: O(1)Pop - إزالة العنصر العلوي.

تعقيد المساحة:

يعتمد على التنفيذ الأساسي (على سبيل المثال، القائمة المرتبطة أو المصفوفة الديناميكية).

يلخص التغليف تعقيدات هياكل البيانات وعملياتها، مما يسمح للمبرمجين باستخدامها بكفاءة دون الحاجة إلى فهم التنفيذ الأساسي. عندما يتعلق الأمر بتعقيدات الزمان والمكان، فإن التغليف لا يؤثر بشكل مباشر على هذه التعقيدات ولكنه يوفر واجهة واضحة للتفاعل مع بنية البيانات. تعتمد التعقيدات الزمنية والمكانية الفعلية على أنماط تنفيذ واستخدام بنية البيانات.

4.2.

Alignment of ADTs with OOP Principles

تشترك أنواع البيانات المجردة (ADTs) والبرمجة الموجهة للكائنات (OOP) في العديد من المبادئ الأساسية التي تدعم وتعزز بعضها البعض:

فهم مبادئ ADTs وOOP مع هياكل بيانات محددة

أنواع البيانات المجردة (ADTs):

التعريف: تحدد ADTs هياكل البيانات من خلال عملياتها بدلاً من تنفيذها. تشمل الأمثلة قوائم الانتظار، والمكدسات، والقوائم، والمصفوفات، وما إلى ذلك.

التغليف: تقوم أدوات ADT بتغليف البيانات والعمليات، وإخفاء تفاصيل التنفيذ عن المستخدم.

مبادئ OOP:

التغليف: تجميع البيانات والأساليب التي تعمل على تلك البيانات ضمن وحدة واحدة (فئة) وتقييد الوصول إلى بعض مكونات الكائن.

التجريد: تحديد نموذج مبسط لواقع معقد.

الوراثة: السماح للفئات الجديدة بوراثة الخصائص والسلوكيات من الفئات الموجودة.

تعدد الأشكال: تمكين الكائنات من التعامل معها كمثيلات لفئتها الأصلية بدلاً من فئتها الفعلية.

دعم مبادئ OOP بهياكل بيانات محددة

طابور:

ADTs: قائمة الانتظار هي ADT التي تحدد عمليات مثل enqueue (إضافة عنصر)، وdequeue (إزالة عنصر)، وإلقاء نظرة خاطفة (عرض العنصر الأمامي).

OOP:

التغليف: يؤدي تنفيذ فئة قائمة الانتظار إلى تغليف العناصر والعمليات التي يمكن إجراؤها عليها.

التجريد: تحدد واجهة قائمة الانتظار العمليات دون تحديد كيفية تنفيذها (على سبيل المثال، استخدام قائمة مرتبطة أو مصفوفة).

الوراثة: يمكننا إنشاء قوائم انتظار متخصصة (على سبيل المثال، قائمة الانتظار ذات الأولوية) عن طريق الوراثة من فئة قائمة الانتظار الأساسية.

تعدد الأشكال: يمكن استخدام أنواع مختلفة من قوائم الانتظار بالتبادل من خلال واجهة قائمة انتظار مشتركة.

كومة:

ADTs: المكدس عبارة عن ADT يحدد عمليات مثل الدفع (إضافة عنصر)، والبوب ​​(إزالة عنصر)، والنظرة الخاطفة (عرض العنصر العلوي).

OOP:

التغليف: يؤدي تنفيذ فئة المكدس إلى تغليف العناصر والعمليات التي يمكن إجراؤها عليها.

التجريد: تحدد واجهة المكدس العمليات دون تحديد كيفية تنفيذها (على سبيل المثال، باستخدام قائمة مرتبطة أو مصفوفة).

الوراثة: يمكننا إنشاء مكدسات متخصصة (على سبيل المثال، مكدس بأقصى حجم) عن طريق الوراثة من فئة المكدس الأساسية.

تعدد الأشكال: يمكن استخدام أنواع مختلفة من الأكوام بالتبادل من خلال واجهة مكدس مشتركة.

قائمة مرتبطة:

ADTs: القائمة المرتبطة هي ADT التي تحدد عمليات مثل الإضافة (إدراج عنصر)، والإزالة (حذف عنصر)، والبحث (البحث عن عنصر).

OOP:

التغليف: يؤدي تنفيذ فئة القائمة المرتبطة إلى تغليف العقد والعمليات التي يمكن إجراؤها عليها.

التجريد: تحدد واجهة القائمة المرتبطة العمليات دون تحديد تفاصيل العقد أو الروابط.

الوراثة: يمكننا إنشاء قوائم مرتبطة متخصصة (على سبيل المثال، قائمة مرتبطة بشكل مضاعف) عن طريق الوراثة من فئة القائمة المرتبطة الأساسية.

تعدد الأشكال: يمكن استخدام أنواع مختلفة من القوائم المرتبطة بالتبادل من خلال واجهة قائمة مرتبطة مشتركة.

المصفوفة الديناميكية:

ADTs: المصفوفة الديناميكية هي ADT التي تحدد عمليات مثل إضافة (إلحاق عنصر)، وإزالة (حذف عنصر)، والحصول على (استرداد عنصر في فهرس محدد).

OOP:

التغليف: يؤدي تنفيذ فئة المصفوفة الديناميكية إلى تغليف العناصر والعمليات التي يمكن إجراؤها عليها.

التجريد: تحدد واجهة المصفوفة الديناميكية العمليات دون تحديد آلية تغيير الحجم.

الوراثة: يمكننا إنشاء مصفوفات ديناميكية متخصصة (على سبيل المثال، المصفوفات الديناميكية ذات عوامل النمو المخصصة) عن طريق الوراثة من فئة المصفوفة الديناميكية الأساسية.

تعدد الأشكال: يمكن استخدام أنواع مختلفة من المصفوفات الديناميكية بالتبادل من خلال واجهة مصفوفة ديناميكية مشتركة.

مصفوفة ثابتة:

ADTs: المصفوفة الثابتة هي ADT التي تحدد عمليات مثل المجموعة (تحديث عنصر)، والحصول (استرداد عنصر)، والطول (استرداد الحجم).

OOP:

التغليف: يؤدي تنفيذ فئة المصفوفة الثابتة إلى تغليف العناصر والعمليات التي يمكن إجراؤها عليها.

التجريد: تحدد واجهة الصفيف الثابت العمليات دون تحديد التخزين الأساسي.

الوراثة: يمكننا إنشاء مصفوفات ثابتة متخصصة (على سبيل المثال، المصفوفات ذات التحقق من الحدود) عن طريق الوراثة من فئة المصفوفة الثابتة الأساسية.

تعدد الأشكال: يمكن استخدام أنواع مختلفة من المصفوفات الثابتة بالتبادل من خلال واجهة مصفوفة ثابتة مشتركة.

الدفاع لفريق التطوير

النمطية وقابلية الصيانة:

يؤدي استخدام ADTs لهياكل البيانات مثل Queue وStack وLinkedList وDynamic Array وFixed Array إلى تعزيز النمطية وقابلية الصيانة. لا تؤثر التغييرات في التنفيذ على بقية قاعدة التعليمات البرمجية بسبب التغليف والتجريد.

إعادة استخدام التعليمات البرمجية وقابلية التوسعة:

من خلال الاستفادة من الميراث وتعدد الأشكال، يمكننا إعادة استخدام تطبيقات بنية البيانات الحالية وتوسيع وظائفها. على سبيل المثال، يمكن أن ترث قائمة الانتظار ذات الأولوية من فئة قائمة الانتظار الأساسية، ويمكن أن ترث القائمة المرتبطة بشكل مزدوج من فئة القائمة المرتبطة الأساسية.

المتانة والأمن:

يساعد التغليف على حماية الحالة الداخلية لهياكل البيانات لدينا، مما يضمن تعديلها فقط من خلال عمليات محددة جيدًا. وهذا يقلل من مخاطر الآثار الجانبية غير المقصودة ويعزز قوة الكود الخاص بنا.

تصميم أوضح:

إن استخدام ADTs كأداة تصميم يتوافق مع مبدأ التجريد لـ OOP، مما يؤدي إلى تصميمات أكثر وضوحًا وقابلية للفهم. وهذا يسهل على أعضاء الفريق فهم النظام واستخدامه وتوسيعه.

في الختام، فإن الفكرة الرئيسية لـ ADTs تدعم وتتوافق مع المبادئ الأساسية لـ OOP، خاصة عند تطبيقها على هياكل بيانات محددة مثل Queue وStack وLinkedList وDynamic Array وFixed Array. توفر ADTs مواصفات واضحة للعمليات وتغليف التفاصيل، بما يتناسب بشكل طبيعي مع مبادئ التغليف والتجريد الخاصة بـ OOP. يتيح الوراثة وتعدد الأشكال أيضًا تطبيقات مرنة وقابلة لإعادة الاستخدام. يؤدي دمج ADTs ضمن إطار عمل OOP إلى تحسين النمطية وقابلية الصيانة والوضوح في عمليات تصميم البرامج وتطويرها.

4.3.

Benefits of Using Built-in Data Structures

يوفر استخدام هياكل البيانات المضمنة التي توفرها لغات البرمجة والمكتبات العديد من الفوائد:

1. الكفاءة والأداء

الفائدة: تم تحسين هياكل البيانات المضمنة في Java بشكل كبير من أجل الأداء.

التبرير:

ADTs وOOP: تم تصميم هياكل البيانات المضمنة في Java، مثل تلك الموجودة في حزمة java.util، باستخدام خوارزميات فعالة وتقنيات معالجة البيانات، بما يتماشى مع مبادئ التجريد والتغليف الخاصة بـ ADTs وOOP. يتم تنفيذ هذه الهياكل من قبل خبراء وتم تحسينها بشكل كبير من أجل الأداء.

مثال: يتم تنفيذ فئة ArrayList في Java (مصفوفة ديناميكية) وفئة HashMap (خريطة التجزئة) باستخدام خوارزميات محسنة للوصول السريع وأوقات التعديل. تقوم ArrayList بتغيير حجم نفسها ديناميكيًا عند الحاجة، وتوفر HashMap متوسط ​​التعقيد في الوقت الثابت لعمليات الحصول والوضع.

2. الموثوقية والمتانة

الفائدة: يتم اختبار هياكل البيانات المضمنة في Java بشكل صارم ويتم استخدامها على نطاق واسع، مما يجعلها موثوقة وقوية.

التبرير:

ADTs وOOP: تم تصميم هياكل البيانات المضمنة في Java لتكون قوية، وتتعامل مع حالات الحافة والأخطاء بأمان. وتتوافق هذه الموثوقية مع مبدأ التغليف الخاص بـ ADTs وOOP، حيث تتم حماية الحالة الداخلية والسلوك من التداخل الخارجي.

مثال: توفر فئة LinkedList في Java تطبيق قائمة مرتبطة بشكل مزدوج. لقد تم اختباره على نطاق واسع واستخدامه في العديد من التطبيقات، مما يضمن أنه يتعامل مع السيناريوهات المختلفة بفعالية. بالإضافة إلى ذلك، توفر فئات مثل ConcurrentHashMap عمليات آمنة لسلسلة العمليات، مما يدل على المتانة في البيئات المتزامنة.

3. سرعة التطوير وسهولة الاستخدام

الفائدة: يؤدي استخدام هياكل البيانات المضمنة في Java إلى تسريع وقت التطوير وتبسيط التعليمات البرمجية.

التبرير:

ADTs وOOP: توفر هياكل البيانات المضمنة في Java واجهات واضحة وموثقة جيدًا، بما يتماشى مع مبدأ تجريد ADTs وOOP. يمكن للمطورين استخدام هذه الهياكل دون الحاجة إلى فهم تفاصيل التنفيذ، مما يسمح لهم بالتركيز على المنطق ذي المستوى الأعلى.

مثال: توفر واجهة قائمة الانتظار وتطبيقاتها مثل LinkedList وPriorityQueue عمليات قائمة انتظار جاهزة للاستخدام (قائمة الانتظار، وإلغاء قائمة الانتظار) دون أن يضطر المطورون إلى تنفيذ القائمة المرتبطة الأساسية أو الكومة. يؤدي هذا إلى تطوير أسرع وصيانة أسهل، حيث أن الكود أكثر قابلية للقراءة والفهم.

يوفر استخدام هياكل البيانات المضمنة في Java فوائد كبيرة من حيث الكفاءة والموثوقية وسرعة التطوير، والتي تتوافق مع مبادئ ADTs وOOP:

الكفاءة والأداء: تم تحسين هياكل البيانات المضمنة في Java وتوفر مزايا كبيرة في الأداء بفضل خوارزمياتها الفعالة وتقنيات معالجة البيانات.

الموثوقية والمتانة: يتم اختبار هذه الهياكل بدقة واستخدامها على نطاق واسع، مما يضمن قوتها وتعاملها مع مختلف الحالات الطرفية بفعالية.

سرعة التطوير وسهولة الاستخدام: توفر هياكل البيانات المضمنة واجهات واضحة وتبسط التطوير، مما يسمح للمطورين بالتركيز على منطق التطبيق عالي المستوى وتحسين إمكانية قراءة التعليمات البرمجية وقابلية الصيانة.

من خلال الاستفادة من هياكل البيانات المضمنة في Java، يمكننا التأكد من أن التعليمات البرمجية الخاصة بنا فعالة وموثوقة وسهلة الصيانة، مما يؤدي إلى تحسين جودة البرامج بشكل عام.

4.4.

Understanding Trade-offs in Software Development

تتضمن المقايضة الموازنة بين سمتين أو أكثر من السمات أو المتطلبات المتعارضة في تطوير البرمجيات. عادةً ما يتضمن اختيار سمة واحدة التنازل عن سمة أخرى. يعد فهم وإجراء المقايضات أمرًا بالغ الأهمية لتحسين الأداء وتحقيق النتائج المرجوة في أنظمة البرمجيات.

تشير المقايضة في البرمجة إلى الموقف الذي تضحي فيه بجودة أو جانب واحد من النظام من أجل الحصول على نوعية أو جانب آخر. غالبًا ما يكون هذا ضروريًا لأن تحسين عوامل متعددة في وقت واحد (مثل السرعة واستخدام الذاكرة وسهولة القراءة) أمر صعب وأحيانًا مستحيل. إن التعرف على المقايضات وإجرائها يسمح للمطورين بتخصيص حلول لمتطلبات وقيود محددة.

مثال البرمجة في جافا: المقايضات في فرز الخوارزميات

دعونا نستخدم خوارزميات الفرز لتوضيح المقايضة في تحقيق أداء أفضل. على وجه التحديد، سنقوم بمقارنة المفاضلة بين استخدام خوارزمية عالية الكفاءة مثل QuickSort مقابل خوارزمية أكثر استقرارًا ولكن من المحتمل أن تكون أقل أداءً مثل MergeSort.

الفرز السريع مقابل MergeSort

فرز سريع:

الايجابيات:

متوسط ​​التعقيد الزمني للحالة: O(n log n)

بشكل عام أسرع في الممارسة العملية بسبب الأداء الجيد لذاكرة التخزين المؤقت وانخفاض الحمل.

سلبيات:

أسوأ تعقيد زمني: O(n^2) (على الرغم من أن هذا أمر نادر مع التحديد الجيد للمحور)

غير مستقرة (قد لا تحتفظ العناصر المتساوية بترتيبها الأصلي)

فرز الدمج:

الايجابيات:

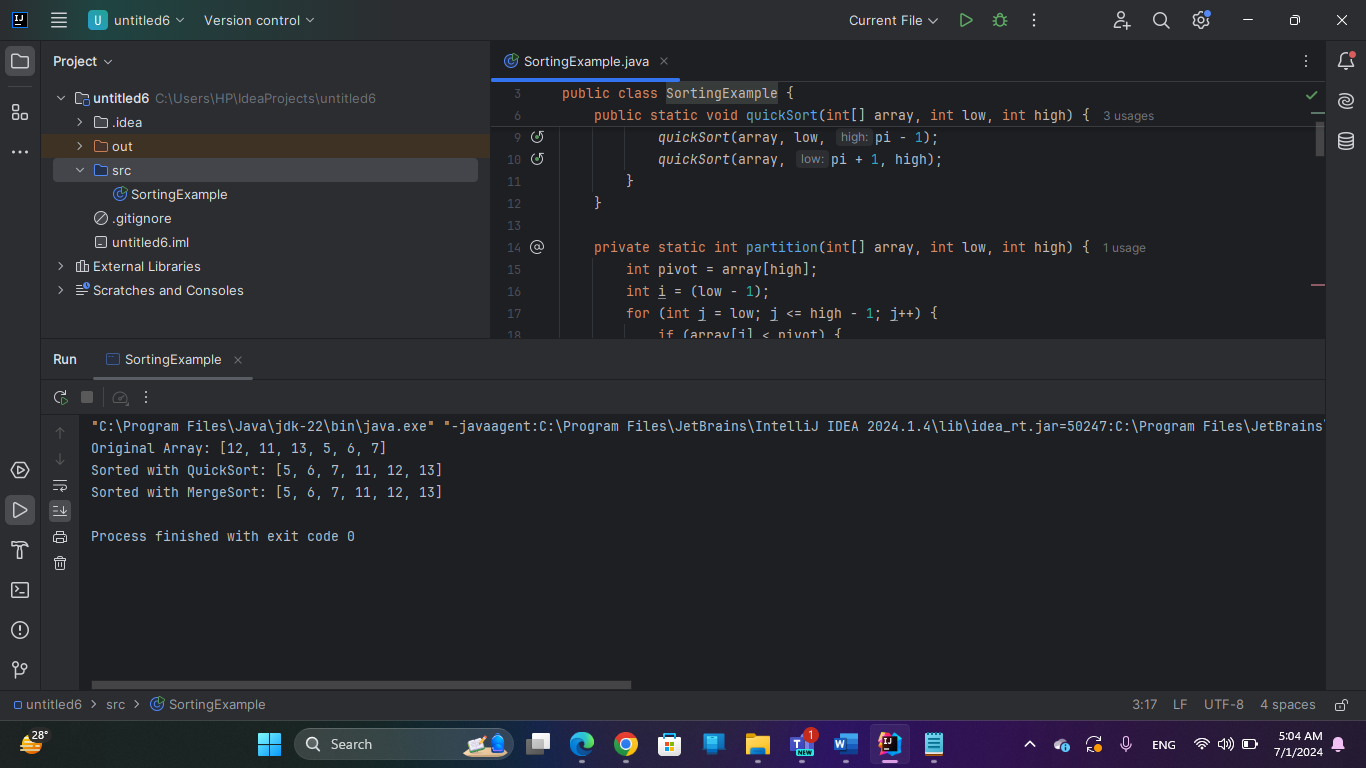
التعقيد الزمني: O(n log n) في جميع الحالات

مستقرة (تحتفظ العناصر المتساوية بترتيبها الأصلي)

سلبيات:

يتطلب ذاكرة إضافية لعملية الدمج (تعقيد مساحة O(n))

أبطأ بشكل عام بسبب الحمل الزائد للدمج واستخدام الذاكرة الإضافية



شرح المقايضات في المثال

الكفاءة والأداء:

QuickSort: تم اختياره لكفاءته المتوسطة في الحالة وانخفاض الحمل، مما يجعله مناسبًا لمجموعات البيانات الكبيرة حيث يكون الأداء أمرًا بالغ الأهمية.

MergeSort: تم اختياره لأدائه المتسق O(n log n)، والذي يمكن التنبؤ به والموثوق به، وإن كان ذلك على حساب استخدام ذاكرة إضافية.

استقرار:

QuickSort: غير مستقر، لذلك إذا كان ترتيب العناصر المتساوية مهمًا (على سبيل المثال، فرز الكائنات حسب مفاتيح متعددة)، فقد لا يكون هذا هو الخيار الأفضل.

MergeSort: مستقر، مما يضمن احتفاظ العناصر المتساوية بترتيبها الأصلي، وهو أمر مهم في بعض التطبيقات مثل فرز السجلات في قاعدة البيانات حسب حقول متعددة.

يتضمن الاختيار بين QuickSort وMergeSort مقايضة بين الأداء والاستقرار. من خلال فهم هذه المقايضات، يمكننا اتخاذ قرارات مستنيرة بناءً على المتطلبات والقيود المحددة لتطبيقنا، مما يؤدي في النهاية إلى تحقيق أداء ووظائف أفضل بشكل عام.

**Plagiarism**

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalised. It is your responsibility to ensure that you understand correct referencing practices. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

**Student Declaration**

|  |
| --- |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.  Student signature: Date: |