1 - 4

Wie funktioniert die Divide and Conquer-Methode?

: r1 Unterteilt das Problem in Teilaufgaben, die unabhängig sein müssen.

: r2 Unterteilt das Problem in möglicherweise abhängige Unteraufgaben.

: r3 Teilt Aufgaben auf mehrere Computer auf

: r4 Sucht keine bestimmte Lösung, sondern nur einen geeigneten Ansatz

: r1 ok 2

-

Wie funktioniert ein hungriger Algorithmus?

: r1 Ruft sich selbst an, bis er eine Lösung findet

: r2 Trifft Entscheidungen basierend auf zufälligen (pseudozufälligen) Wahlen

: r3 Sucht nach lokalen Extremen, um globale Extreme zu finden

: r4 Unterteilt das Problem in abhängige Unteraufgaben.

: r3 ok 2

--

Was ist ADT - Abstrakter Datentyp?

: r1 Abstrakte Daten, die nicht implementiert werden können

: r2 Bezeichnet Datentypen, die von der Implementierung unabhängig sind

: r3 Allgemeine Datentypen wie Interger, Real, Boolean…

: r4 Spezifische Implementierung des Datentyps

: r2 ok 2

-

Die grundlegenden Operationen mit ADT sind:

: r1 Konstruktor, Selektor, Modifizierer

: r2 Konstruktor, Destruktor

: r3 Konstruktor, Selektor, Modifizierer, Destruktor

: r4 Selector, Destruktor

: r1 ok 2

--

Experimentelle Analyse des zeitaufwendigen Algorithmus:

: r1 Läuft in einer Umgebung, in der das Programm (der Algorithmus) ausgeführt wird

: r2 Hängt nicht von der Umgebung ab, in der das Programm ausgeführt wird

: r3 Erfordert keine Implementierung eines Algorithmus

: r4 Konzentriert sich auf den einfachsten (am wenigsten herausfordernden) Fall

: r1 ok 2

-

Pseudocode:

: r1 Abhängig von der Programmiersprache

: r2 Zeigt Probleme der spezifischen Implementierung

: r3 Ist eine höhere Beschreibungsebene, strukturierter als eine klassische Beschreibung, weniger detailliert als eine Implementierung

: r4 Eine niedrigere Beschreibungsebene, die weniger strukturiert ist als die klassische Beschreibung, erfordert Kenntnisse einer bestimmten Programmiersprache

: r3 ok 2

--

Primitive Operation

: r1 Abhängig von der jeweiligen Programmiersprache kann sie in einem Pseudocode nicht identifiziert werden

: r2 Ein anderer Name für eine bestimmte Prozedurimplementierung (Methode, Algorithmus)

: r3 Ein Programmblock, der mehrere Operationen enthält, die bei einem einzigen Aufruf ausgeführt werden

: r4 Grundlegende Operation, die von einem Algorithmus ausgeführt wird, z. B. Auswerten eines Ausdrucks, Zuweisen eines Werts zu einer Variablen, Aufrufen einer Prozedur ...

: r4 ok 2

-

Wenn die vom Algorithmus <b> 2n <sup> 2 </ sup> -3 </ b> ausgeführte Zahl die Zeit und die Big O-Notation ist:

: r1 Exponential, O (c ^ n -3)

: r2 logarithmisch, O (log n)

: r3 Konstante, O (2)

: r4 Quadratisch, O (n ^ 2)

: r4 ok 2

--

ADT Magazine

: r1 Das Einfügen und Löschen erfolgt nach dem FIFO-Prinzip

: r2 Die Grundoperationen sind: push (Object), findMin (), findMax (), deleteAll ()

: r3 Wirft niemals eine Ausnahme

: r4 Folgt dem LIFO-Prinzip und die grundlegenden Operationen sind: push (Objekt), pop ()

: r4 ok 2

-

Wenn der Stack feldbasiert ist,

: r1 Dann ist die für jede Operation erforderliche Zeit O (n ^ 2 + n), wenn n die Anzahl der Elemente im Stapel ist

: r2 Zu Beginn müssen wir die Größe des Stapels definieren

: r3 Es können beliebig viele Elemente gespeichert werden, die Ablage kann jederzeit problemlos vergrößert werden

: r4 Löst beim Hinzufügen eines Elements zu einem vollständigen Stapel eine EmptyStackException aus

: r2 ok 2

--

ADT-Warteschlange

: r1 Das Einfügen und Löschen erfolgt nach dem FIFO-Prinzip

: r2 Die Grundoperationen sind: push (Object), findMin (), findMax (), deleteAll ()

: r3 Das Einsetzen und Schmieren erfolgt nach dem LIFO-Prinzip

: r4 Wirft niemals eine Ausnahme

: r1 ok 2

-

Warteschlange

: r1 Speichert eine Sammlung von Elementen, wobei das Element ein geordnetes Prioritätswertpaar ist, um die Warteschlange in Ordnung zu halten

: r2 Zur Implementierung verwenden wir ein kreisförmiges Array

: r3 Die Grundoperationen sind: enqueue (Objekt), dequeue (), first (), last (), order ()

: r4 Zeitkomplexität für jede Implementierung ist am besten O (n ^ 2 log n)

: r2 ok 2

--

Vektor

: r1 Definiert die Vorher / Nachher-Beziehung zwischen Positionen

: r2 Elemente sind nach Größe sortiert

: r3 Erweitert das Konzept eines Arrays, um eine Folge beliebiger Objekte zu speichern

: r4 Ermöglicht das Speichern von Objekten an negativen Positionen (negativer Index)

: r3 ok 2

-

Liste

: r1 Dynamische Datenstruktur, Größenänderungen aufgrund des Hinzufügens von Elementen (Entfernen)

: r2 Statische Datenstruktur, Größe ändert sich nicht

: r3 Ein Element kann durch Angabe seiner Reihenfolge gelesen, eingefügt und entfernt werden

: r4 Ein allgemein gültiger Grunddatentyp zum Speichern einer geordneten Menge von Elementen

: r1 ok 2

--

Einzelne verknüpfte Liste

: r1 Der Knoten enthält einen Verweis auf den vorherigen und nächsten Knoten

: r2 Kann verwendet werden, um Stapel und Warteschlange mit linearem Speicher und konstanter Zeit für jede Basisoperation zu implementieren

: r3 Statische Datenstruktur

: r4 Definiert die Pre / Post-Beziehung zwischen Positionen

: r2 ok 2

--

Sequenz

: r1 ist die Kombination aus Vektor und Liste, die nach Reihenfolge und Position auf Elemente zugreift

: r2 Auf Elemente kann nicht durch Angabe ihrer Reihenfolge zugegriffen werden

: r3 FIFO-basierter Zugriff auf Elemente

: r4 LIFO-basierter Zugriff auf Elemente

:r1 ok 2

5 – 8

Baum

: r1 Lineare statische Datenstruktur

: r2 Enthält Knoten mit Eltern-Kind-Beziehung

: r3 Der externe Knoten heißt root

: r4 Der interne Knoten wird Blatt genannt

: r2 ok 2

-

Binärer Baum

: r1 Enthält zwei Wurzeln

: r2 Enthält nur zwei Blätter

: r3 Jeder interne Knoten hat mindestens zwei untergeordnete Knoten, von denen mindestens einer ein Blatt ist

: r4 Jeder interne Knoten hat genau zwei Kinder, die ein geordnetes Paar bilden (linkes Kind, rechtes Kind)

:r4 ok 2

--

Prioritätswarteschlange

: r1 Führt den Datentyp Comparator ein, mit dem Sie zwei Objekte vergleichen können

: r2 Zwei verschiedene Elemente müssen immer unterschiedliche Priorität haben (Schlüssel)

: r3 Anordnung muss auf der Menge der Elemente definiert werden

: r4 Grundlegende Operationen mit Prioritätswarteschlange sind: insertItem (k, o), removeMin (), upheap (), downheap ()

: r1 ok 2

-

Prioritätswarteschlange

: r1 Auf gespeicherte Elemente kann basierend auf ihrem Index in der Warteschlange zugegriffen werden

: r2 Schlüssel sind beliebige Objekte, für die die Reihenfolge definiert werden kann, wobei zwei verschiedene Elemente denselben Schlüssel haben können

: r3 Lineare statische Datenstruktur

: r4 Das Element mit der niedrigsten Priorität wird immer zuerst gezeichnet

:r2 ok 2

--

Haufen

: r1 Wird als Binärbaum dargestellt, in dem Schlüssel als interne Knoten gespeichert werden, deren Knotenschlüssel immer größer oder gleich dem übergeordneten Schlüssel ist

: r2 Die Wurzel enthält den größten Schlüssel

: r3 Der letzte Knoten des Stapels ist der äußerste linke Knoten

: r4 Die Funktion downheap () stellt die Heap-Anordnung beim Einfügen eines neuen Elements wieder her

: r1 ok 2

-

Haufen

: r1 FIFO-basierter Zugriff auf Elemente

: r2 LIFO-basierter Zugriff auf Elemente

: r3 Nach dem Einfügen eines Knotens müssen Sie die Funktion upheap () aufrufen, um den Stack neu zu erstellen

: r4 Das Einfügen und Entfernen von Elementen aus dem Heap beeinträchtigt die Anordnung nicht

: r3 ok 2

--

Wörterbuch

: r1 Die Größe muss unabhängig von der Implementierung beim Erstellen des Wörterbuchs immer festgelegt werden

: r2 Erfordert, dass sowohl eine Schlüssel- als auch eine Wertereihenfolge in einem Schlüssel-Wert-Paar definiert sind

: r3 Ein geordnetes Schlüssel-Wert-Paar, bei dem Schlüssel eindeutig sind, Werte jedoch nicht

: r4 Wir entfernen immer das Element mit dem kleinsten Schlüssel

: r3 ok 2

-

Protokolldatei

: r1 Gilt nicht, wenn in der Wertemenge keine Reihenfolge definiert ist

: r2 Wird für kleine Wörterbücher oder Anwendungen verwendet, in denen die häufigste Suchoperation ausgeführt wird

: r3 Wird verwendet, wenn der Einfügevorgang am häufigsten vorkommt und das Suchen und Entfernen selten durchgeführt wird.

: r4 Die Elemente werden immer in der Mitte der Sequenz eingefügt

:r3 ok 2

--

Nachschlagetabelle

: r1 Wir verwenden zur Implementierung immer ungeordnetes Vokabular

: r2 Wird für kleine Wörterbücher oder Anwendungen verwendet, in denen die häufigste Suchoperation ausgeführt wird

: r3 Wird verwendet, wenn das Einfügen die häufigste Operation ist, während das Suchen und Entfernen selten durchgeführt wird

: r4 LIFO-basierter Zugriff auf Elemente

: r2 ok 2

-

Hash-Tabelle

: r1 Für einen bestimmten Schlüsseltyp verfügt er über eine Hash-Funktion, die dem Schlüssel einen ganzzahligen Wert und ein Feld (Tabelle) der Größe N zuweist

: r2 Kann nicht verwendet werden, wenn die Hash-Funktion denselben Wert an verschiedene Schlüssel zurückgibt (Kollision auftritt)

: r3 Erfordert, dass sowohl eine Schlüssel- als auch eine Wertereihenfolge in einem Schlüssel-Wert-Paar definiert sind

: r4 Ordnet die Werte unabhängig vom Hash-Wert nach Schlüsseln, die nur zur Überprüfung der gespeicherten Daten dienen

:r1 ok 2

--

Sortieralgorithmen

: r1 Wir nennen es stabil, wenn sie die Einfügereihenfolge von Elementen mit demselben Schlüssel beibehalten

: r2 Sortiert den Datensatz nach Wert und nicht nach Schlüssel

: r3 Benötigt eine definierte Reihenfolge der Werte

: r4 Kann nicht verwendet werden, wenn die Daten mehrere Schlüssel mit demselben Wert enthalten

:r1 ok 2

--

Bubblesort

: r1 Vergleicht das erste und das letzte Element einer Reihe

: r2 Benötigt zusätzlichen Zusatzspeicher für die Größe der sortierten Datendatei

: r3 Ein einziger Durchgang reicht aus, um einen Datensatz zu vergleichen

: r4 Universal, die Elemente mit dem höchsten Wert sprudeln ans Ende der Liste

: r4 ok 2

-

Haufen sortieren

: r1 Ist ein stabiler Sortieralgorithmus unter Verwendung der ADT-Liste

: r2 Erstellt einen Heap aus dem Array und platziert das kleinste Element als Root, das wir dann entfernen

: r3 Anspruchsvoll, seine minimale zeitliche Komplexität ist O (n ^ 3)

: r4 Daten können nicht an Ort und Stelle sortiert werden. Erfordert zusätzlichen Zusatzspeicher in der Größe der sortierten Datendatei

:r2 ok 2

--

Einfügung sortieren

: r1 Vergleicht immer zwei benachbarte Elemente in einer Sequenz

: r2 Schlägt fehl, wenn der Satz teilweise sortiert ist

: r3 Stabil, effizient bei teilweise bestellten Sets, kann Daten online sortieren (wenn sie eingegeben werden)

: r4 Instabil, wird für große Datenmengen verwendet

: r3 ok

9 - 12

Sortierung zusammenführen

: r1 instabil, kann nicht parallelisiert werden, zusätzlicher Speicher benötigt O (1)

: r2 Stabil, parallelisierbar, schlechteste Zeitkomplexität ist O (nlog n)

: r3 Ein einziger Durchgang reicht aus, um einen Datensatz zu vergleichen

: r4 Vergleicht das erste und das letzte Element einer Reihe

:r2 ok 2

--

Quicksort

: r1 Stabil, unkompliziert, nur ein Durchgang, um einen Datensatz zu vergleichen

: r2 Schlägt fehl, wenn der Satz teilweise sortiert ist

: r3 Universal, die Elemente mit dem höchsten Wert sprudeln ans Ende der Liste

: r4 Im Durchschnitt ist dies der schnellste bekannte, instabile Algorithmus, der einen Pivot zum Sortieren verwendet

: r4 ok 2

-

Auswahl sortieren

: r1 Ein Algorithmus, der komplex zu implementieren ist, aber sehr schnell, mit nur einem Durchgang, um einen Datensatz zu vergleichen

: r2 Universell, lokal, instabil, geeignet für kleine Datenmengen

: r3 Benötigt zusätzlichen O (n ^ 2) Speicher

: r4 Minimale und durchschnittliche Zeitkomplexität ist O (n)

: r2 ok 2

--

Für Sortieralgorithmen:

: r1 Im schlimmsten Fall ist die zeitliche Komplexität von Heapsort und Mergesort O (nlog n)

: r2 Heapsort, Quicksort und Insertion sort sind alle instabil

: r3 Heapsort, Quicksort, Mergesort und Insertion sortieren die Daten vor Ort (vorhanden, zusätzlicher Speicherbedarf ist O (1))

: r4 Bubble Sort und Insertion Sort haben idealerweise eine konstante zeitliche Komplexität - O (1)

: r1 ok 2

-

Mustervergleich

: r1 bedeutet Suche nach Muster P in einer gegebenen T-Sequenz (Teilstringsuche in einem String)

: r2 Vergleichen Sie zwei Objekte anhand ihrer Größe

: r3 Suchen Sie nach sich wiederholenden Textteilen

: r4 Findet eine Sequenz, um den Text zu kodieren

: r1 ok 2

--

Mustererkennung - Brute Force

: r1 Scrollt den Text von rechts nach links (von hinten), scrollt nicht durch alle Positionen

: r2 Vergleicht P-Muster mit T-Text für alle möglichen Positionen

: r3 Ist ein einfacher und zeitsparender Algorithmus, dessen Komplexität mit längerem Text abnimmt

: r4 Entfernen Sie Zeichen aus dem Text, bis T eine Zeichenfolge mit der Länge von P bleibt

: r2 ok 2

-

Pattern Matching - KMP-Algorithmus

: r1 Durchsucht den Text von links nach rechts, genau wie der Brute-Force-Algorithmus alle Vergleiche vornimmt

: r2 Wenn eine Nichtübereinstimmung festgestellt wird, wird die Suche um mehr als einen Buchstaben verschoben, sodass nicht alle Vergleiche möglich sind

: r3 Ist ein einfacher und zeitsparender Algorithmus, dessen Komplexität mit längerem Text abnimmt

: r4 Vergleicht Zeichenfolgen von hinten und scrollt jeweils um ein Zeichen

:r2 ok 2

--

Trie

: r1 Die Knotenreihenfolge auf dieser Ebene gibt die Reihenfolge des Buchstabens im Wort an

: r2 Wenn Trie komprimiert ist, enthält jeder Knoten genau einen Buchstaben

: r3 Struktur für die Textverarbeitung, bei der jeder Knoten einen Buchstaben oder ein Wort enthält

: r4 Es ist nicht möglich, den gesamten Text zu speichern

: r3 ok 2

-

Graphentheorie

: r1 Wenn das Diagramm ausgerichtet ist, enthält es mindestens eine ausgerichtete Kante (nicht alle Kanten müssen ausgerichtet sein)

: r2-Sequenz ist eine Folge von Scheitelpunkten, sodass zwischen aufeinanderfolgenden Scheitelpunkten eine Kante besteht und sich die Scheitelpunkte nicht wiederholen

: r3 Die Kante wird immer durch mehr als zwei Eckpunkte, Gewicht und Richtung, bestimmt

: r4 Der Graph ist ein geordnetes Paar V, E, wobei V eine Menge von Eckpunkten und E eine Menge von Kanten ist

:r4 ok 2

--

Den kürzesten Weg finden

: r1 Im ungünstigsten Fall erfolgt der Graph unabhängig vom gewählten Algorithmus immer mit der Zeitkomplexität O (n)

: r2 Funktioniert auf allen Arten von Graphen mit beliebigen Kantenwerten unter Verwendung des Dijkstra-Algorithmus

: r3 Mit dem Floyd-Warshall-Algorithmus wird ein orientiertes Diagramm mit positiven Kanten benötigt und der kürzeste Weg zwischen allen Scheitelpunkten ermittelt

: r4 Kann nicht für Diagramme mit negativen Flanken verwendet werden

: r3 ok 2

-

Genetische Algorithmen

: r1 Sie ahmen evolutionsbiologische Techniken nach, sie gehören zur künstlichen Intelligenz

: r2 Es ist ein genauer deterministischer Algorithmus, der keine Zufälle verwendet

: r3 Die Art der Beschreibung (Kodierung) von Personen hat keinen Einfluss auf den Erfolg oder Misserfolg der Lösung einer bestimmten Aufgabe

: r4 Wenn sie mehrmals hintereinander ausgeführt werden, geben sie immer das gleiche Ergebnis

:r1 ok 2

--

Auswahl - Auswahl von Personen für die Elternschaft

: r1 Es wird so gemacht, dass Eltern zu Individuen mit dem niedrigsten Fitnesswert werden

: r2 Bei der zufälligen Auswahl von Eltern verwenden wir deren Fitnesswert

: r3 Es wird versucht, die am wenigsten profitablen Personen für die nächste Generation zu gewinnen

: r4 Beeinflusst nicht den Erfolg oder Misserfolg der Lösung einer bestimmten Aufgabe

: r3 ok 2

-

Überqueren

: r1 Es gibt keine Möglichkeit, mehr als zwei Individuen in einer Generation zu kreuzen

: r2 Eltern tauschen einen Teil des genetischen Codes aus;

: r3 Jede Person in einer bestimmten Generation muss Eltern werden

: r4 Der Elternteil hört immer auf und sein Kind nimmt den Platz ein, dieses Kind hat den gleichen Fitnesswert wie sein Elternteil

:r2 ok 2

--

Mutation

: r1 Eine zufällige Veränderung im Genom eines Individuums tritt mit sehr geringer Wahrscheinlichkeit auf

: r2 Niemals bringt neue Funktionen, die in der ursprünglichen Generation nicht zu finden wären

Jeder Mensch aus jeder Generation mutiert an mindestens einem Ort

Mutationen bei verschiedenen Individuen treten immer an derselben Stelle im Genom auf

: r1 ok 2