Inhalt

Motivation

Ziel:

- Programmier- (Java)-technische Realisierung von großen Mengen gleichartiger Daten

Dabei will man (mindestens)

- auf einzelne Datensätze zugreifen (lesend oder schreibend)
- Datensätze hinzufügen
- Datensätze entfernen

können.

Dazu gibt es prinzipell zwei unterschiedliche Ansätze:

- Arraybasierte Strukturen
- Verkettete Strukturen

Inhalt

Arrays

schon bekannt: Arrays

- beliebiger Basisdatentyp (simple type oder bel. Referenztyp)
- „wahlfreier“ Zugriff auf beliebige Position innerhalb des Arrays

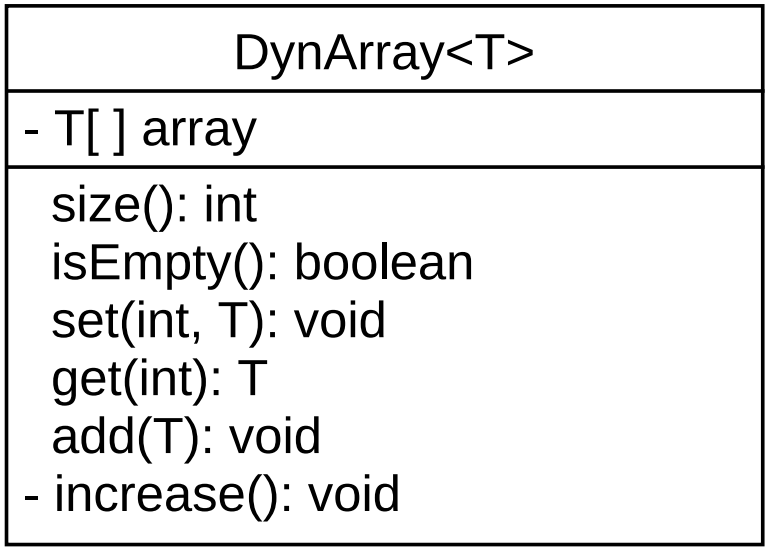
Nachteil:

- feste Länge ($\leadsto$ begrenzte maximale Anzahl speicherbarer Datensätze)

Ausweg:

- Definiere (generische) Klasse, die „bequemen“ Umgang mit Arrays erlaubt, insbesondere:
- Einfügen von Elementen mit Verlängerung des Arrays, falls nötig

Klasse DynArray



(zunächst noch ohne remove() und ohne decrease())

- `size` bezeichnet die Anzahl der gespeicherten Elemente, nicht die Länge des Arrays!
- keine simple types als Basistyp möglich (Wrapper-Klassen nutzen)

Semantik von set, get und add

Bei „unseren“ dynamischen Arrays: Einschränkung des wahlfreien Zugriffs

- set(**int** pos, T v)
  ändern dh überschreiben von bereits bestehenden Einträgen
  $\rightsquigarrow$ nur auf bereits besetzte Positionen anwendbar
- get(**int** pos)
  lesen eines bereits vorhandenen Elements
  $\rightsquigarrow$ nur auf bereits besetzte Positionen anwendbar
- add(T v)
  einfügen von Elementen: keine Wahlfreiheit der Position, sondern
  $\rightsquigarrow$ immer Einfügen an erste unbelegte Position des Arrays

Implementierung von set und get

- Zugriff auf ungültige Array-Positionen löst eine IndexOutOfBoundsException („IOOBE“) aus.

```
public void set(int pos, T e) throws IOOBE {
    if (pos < 0 || pos >= size)
        throw new IOOBE();
    array[pos]= e;
}

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

Implementierung von add

- Verwende ein (privates) Attribut size, das bei jedem Hinzufügen (und später auch beim Entfernen) eines Elementes aktualisiert wird.
- Das erstmalige Belegen einer Komponente (Hinzufügen eines weiteren Datensatzes) geschieht über die Methode add.
  Neue Elemente werden „hinter“ der zuletzt belegten Position angehängt.
  Dadurch erhöht sich die size des dynamischen Arrays.
- Falls die „Kapazität“ (= Länge) des Arrays erschöpft ist, muss vor dem Hinzufügen ein neues Array von größerer Länge angelegt werden:
  $\rightsquigarrow$ increase
- führe erst dann die gewünschte add-Aktion aus

```
public void add(T e) {
    if(size >= array.length)
        increase();
    array[size++] = e;
}
```

Implementierung von increase

- Bei Erzeugung eines Objektes von DynArray wird ein Array mit einer definierten Startgröße angelegt.
- Falls das Array voll besetzt ist und eine weitere add-Aktion ausgeführt werden soll, dann
  - erzeuge ein neues Array von doppelter Länge
  - kopiere die bisherigen Elemente in das neue Array
  - ersetze (die Referenz auf) das alte Array durch (eine Referenz auf) das neue Array

```
private void increase() {
    T[] neu = (T[]) new Object[array.length*2];
    for (int i = 0; i < size; i++)
        neu[i] = array[i];
    array = neu;
}
```

Löschen von Elementen

Zum Entfernen (Löschen) von Elementen aus einem dynamischen Array gibt es verschiedene Anforderungen und verschiedene Strategien:

- unterschiedliche Anforderungen
  - entferne das Element mit dem Eintrag e
  - entferne das Element an der Array-Position pos
  - entferne das erste (oder: das letzte) Element
- unterschiedliche Strategien
  - alle folgenden Elemente rücken eine Position nach vorne
  - die Position wird nur als „frei“ markiert
- Durch Löschen von Elementen können in einem Array sehr viele Positionen unbesetzt sein
  $\leadsto$ entsprechend zum increase auch ein decrease (wieder Verkleinern der Kapazität) eines Arrays sinnvoll

Semantik von remove, delete, decrease

- remove() entfernt das erste Element des Arrays (array [0])
- delete (T e) (sucht und) entfernt das Element mit Eintrag e
- sowohl bei remove() als auch bei delete (T e):
  alle folgenden Feldelemente rücken eine Position nach vorne
  am Ende „rausfallende“ Elemente werden nicht „gelöscht“
- decrease () reduziert das Array auf halbe Länge, falls es nur noch zu einem Viertel gefüllt ist

Implementierung $\rightsquigarrow$ Übung

Beispiel: Verhalten eines dynamischen Arrays

```
DynArray<Integer> d = new DynArray<>();
for (int i=0; i < 20; i++)
    d.add(i);
for (int i = 0; i < 15; i++)
    d.delete(i);
d.remove();
d.remove();
```

Speicher-Inhalt nach Ablauf:
size = 3, length = 8, Arrayelemente: 17 18 19 19 null null null null

Inhalt

Erinnerung: Motivation

Ziel:

- Programmier- (Java)-technische Realisierung von großen Mengen gleichartiger Daten

Dabei will man

- auf einzelne Datensätze zugreifen (lesend oder schreibend)
- Datensätze hinzufügen
- Datensätze entfernen

können.

- Wie gut werden diese Anforderungen von der Implementierung mittels Dynamischer Arrays erfüllt?
- Was bedeutet „gut“?

Qualität einer Implementierung

Wesentliches Kriterium: Ressourcenverbrauch der Operationen
Relevante Ressourcen:

- Speicherplatz-Bedarf
- i.F. betrachtet: Zeitbedarf
- abhängig von der „Problemgröße“, d.i. bei uns:
  Anzahl der gespeicherten Datenelemente

Landau-Symbole

- „Groß-O-Notation“, um asymptotisches Verhalten von Funktionen beschreiben zu können
- Bei uns meist betrachtete Funktion:
  $f(n)$ = Anzahl der benötigten „elementaren Rechenschritte“ bei Anwendung einer Methode auf n Datensätze.
- „asymptotisch“: wie verändert sich $f(n)$ bei Veränderung (zB Verdopplung) von n?
  (Ohne Rücksicht auf additive und multiplikative Konstanten)
- Definition:

$$f \in \mathcal{O}(g) :\Leftrightarrow \exists C > 0 \exists x_0 > 0 \forall x > x_0 : |f(x)| \leq C \cdot |g(x)|$$

Beispiel und pragmatische Schreibweise für $g(n) = n$:

$$f(n) = \mathcal{O}(n)$$

Komplexitätsklassen (aufsteigend sortiert)

$f = \ldots$	Bezeichnung	bedeutet: wenn n sich verdoppelt ...
$\mathcal{O}(1)$	konstant	... ändert sich $f(n)$ nicht
$\mathcal{O}(\log n)$	logarithmisch	... wächst $f(n)$ um eine (add.) Konstante
$\mathcal{O}(n)$	linear	... verdoppelt sich $f(n)$
$\mathcal{O}(n \log n)$	superlinear	(etwas schlechter als linear)
$\mathcal{O}(n^2)$	quadratisch	... vervierfacht sich $f(n)$
$\mathcal{O}(n^k)$	polynomiell	... wächst $f(n)$ um den Faktor 2^k
$\mathcal{O}(2^n)$	exponentiell	$f(n)$ verdoppelt sich, wenn n nur um 1 wächst (!)

Beispiel: Dynamische Arrays

- Speicherbedarf: maximal $4 \cdot n$, also $\mathcal{O}(n)$
- Zeitbedarf von `get(int pos)`:
  1 Zugriff, 2 Vergleiche, 1 return, also $\mathcal{O}(1)$
- Zeitbedarf von `contains(T e)` (vgl Übungsaufgabe)
  Maximal (im worst case) n Zugriffe und n Vergleiche, also $\mathcal{O}(n)$
- Zeitbedarf von `add(T e)`
  $\leadsto$ das kommt auf die aktuelle `size` an!
  - konstant, wenn `size` $<$ „`capacity`“
  - $\mathcal{O}(n)$, wenn `size` $\geq$ „`capacity`“

  „amortisierter“ Zeitbedarf: $\mathcal{O}(1)$
  (Beweis: Video)