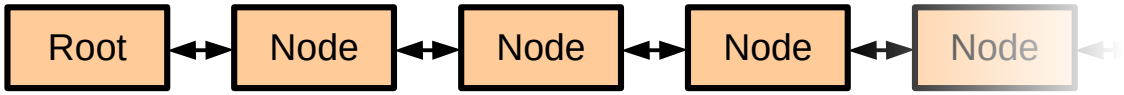
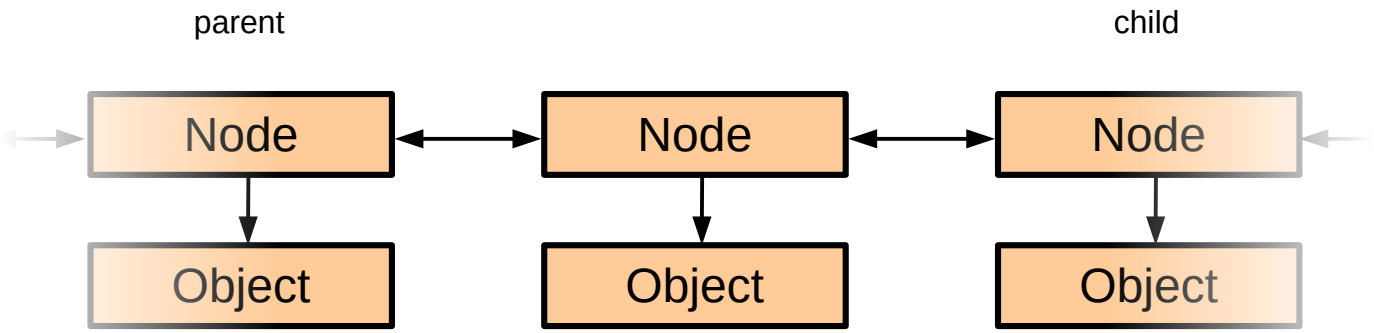
Dynamische Datenstrukturen

1. Verlinkte Objekte



In diesem Praktikum schreiben Sie eine dynamische Liste, die durch eine Verkettung von Objekten entsteht. Dabei sollen die Kettenglieder durch einzelne *Nodes* aufgebaut werden, welche jeweils ein Objekt tragen. Das Objekt selber ist dann der Listen-Element-Inhalt.

Schreiben Sie eine Klasse *Node*, die jeweils eine Referenz zu einem vorherigen (parent) und darauffolgenden (child) *Node* besitzt. Dabei soll jede *Node* eine Referenz zu einem *Object* besitzen:

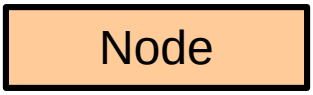


Implementieren Sie folgende Methoden in die *Node*-Klasse:

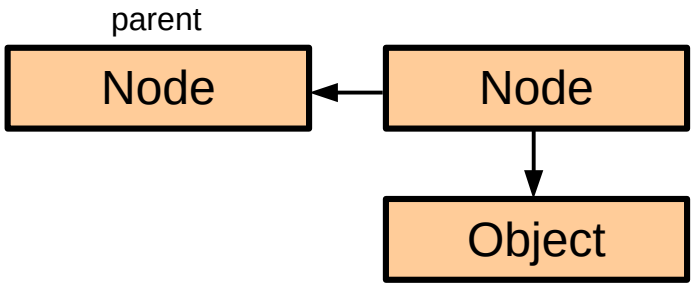
- *next()* Gibt den nächsten (child) *Node* zurück.
- *previous()* Gibt den vorherigen (parent) *Node* zurück.
- *setChild()* Setzt den nächsten (child) *Node*.
- *getContent()* Gibt das referenzierte Objekt zurück.

Beim Instanziieren soll ein *Object*, sowie der vorherige (parent) *Node* aufgenommen werden. Das Objekt stellt dabei den *Node*-Inhalt dar. Während dem Instanziieren soll auch beim vorherigen (parent) *Node* der instanziierende *Node* als nächster (child) *Node* durch die Methode *setChild* gesetzt werden:

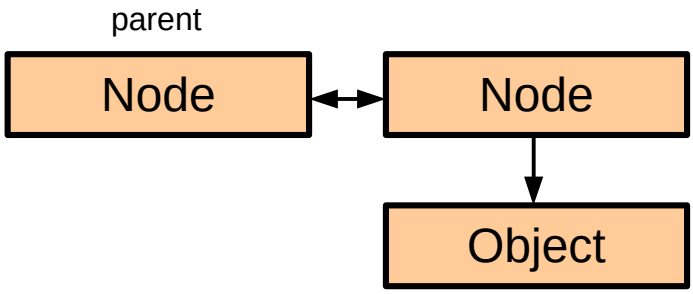
1. Instanziierung von *Node wird gestartet*.



2. *Node* nimmt Objekt und *parent Node* auf und speichert diese Referenzen ab.



3. *Node* greift auf den *parent Node* zu und ruft dessen *setChild*-Methode auf um sich selber dort zu setzten.



Schreiben Sie nun eine Klasse *NodeList*. Diese Klasse soll mithilfe der *Node*-Klasse die dynamische Liste darstellen. Durch eine Referenz auf den ersten *Node (root)* soll ein Einstiegspunkt in die Verkettung bestehen. Schreiben Sie dazu folgende Methoden:

- *add(Object)* Fügt einen neuen *Node* mit entsprechendem Inhalt (*Object)* hinzu. Ist kein *Node* vorhanden, wird der neue *Node* zum *root Node*
- *get(int)* Gibt das Objekt des entsprechenden *Nodes* zurück.
- *remove(*int*)* «Löscht» den entsprechend Node. Nutzen Sie dazu die Funktion des Garbage Collectors.
- *getLegth()* Gibt die Grösse der Liste (Anzahl *Nodes* zurück)

Testen Sie Ihre Implementierung, indem Sie mehrere Zahlen in die Liste abspeichern und diese durch eine *for*-Schleife wieder an die Konsole ausgeben.

2. Statisch vs. Dynamisch

In diesem Praktikum untersuchen Sie das zeitliche Verhalten der beiden Array-Arten. Dazu verwenden Sie ein normales (statisches) Array und Ihre *NodeList* (dynamisch).

1. Erstellen Sie ein statisches Array und eine *NodeList* mit Integer-Werten.
2. Berechnen Sie die entsprechende Summe der einzelnen Listen, indem Sie **jeweils** mit einer *for*-Schleife durch die Elemente durch gehen. Messen Sie bei beiden Durchläufe die Anfangs- und End-Zeit (*System.currentTimeMillis()*) und geben Sie die Differenz an die Konsole aus. Testen Sie dabei die Arrays mit folgenden Bereichen aus:
    1. 0 bis 100
    2. 0 bis 10000
    3. 0 bis 50000
    4. 0 bis 100000
3. Was stellen Sie fest? Warum verhält sich das Programm so?
4. Überlegen Sie sich, wie Sie Ihre *NodeList* optimieren können, damit diese Liste bei dieser Situation (Summenberechnung) noch besser abschneidet. Implementieren und Testen Sie diese Überlegung.

3. Array vs Verkettet

In diesem Praktikum untersuchen Sie das Verhalten von zwei verschiedenen dynamischen Datenstrukturen. Dazu nutzen Sie aus der Standard-Bibliothek die Klasse *ArrayList* und Ihre Klasse *NodeList*. Nutzen Sie jeweils die optimierte Variante von *NodeList*.

Gehen Sie dabei wie folgt vor:

1. Erstellen Sie eine ArrayList und eine *NodeList* ausgelegt für Integer-Werten.
2. Füllen Sie zuerst 50000 Werte in beide Listen und messen Sie die Zeit für die jeweilige Ausführung. Welche Liste ist dabei schneller?
3. Berechnen Sie die Summe von jedem Element und messen Sie die Zeit für die jeweilige Ausführung. Welche Liste ist dabei schneller?
4. Berechnen Sie die Summe von jedem dritten Element und messen Sie die Zeit für die jeweilige Ausführung. Welche Liste ist dabei schneller?
5. Löschen Sie die **ersten** 1000 Einträge aus der entsprechenden Liste heraus und messen Sie die Zeit für die jeweilige Ausführung. Welche Liste ist dabei schneller?
6. Sollte Ihre *NodeList* bei Punkt 2 einiges schlechter als die *ArrayList* abgeschlossen haben (mehr als Faktor 2), überlegen Sie sich, wie Sie die *NodeList* verbessern können.

4. Telefonliste

Ihnen steht die Datei *phone_list.txt* zur Verfügung. Diese Datei beinhaltet den Verlauf von Anrufen innerhalb einer Firma. Dabei entsprechen die Daten folgendes Format:

Vorname_Anrufer *Vorname_Empfänger*

1) Schreiben Sie ein Programm, welches die Daten sammelt und diese den Personen zuordnet. Am Schluss sollen alle Personen mit den Angaben, wie oft sie jemanden angerufen haben und wie oft sie angerufen wurden, an die Konsole ausgegeben werden.

   Für das Zuordnen der Daten zu den Personen sollen Sie die Klasse *HashMap* zur Hilfe nehmen.

2) Ordnen Sie die Ausgabe, so, dass die Person mit den meisten getätigten Anrufen zuoberst steht. Wandeln Sie dazu die *HashMap*-Einträge in eine *ArrayList* um und nutzen Sie dessen *sort*-Methode. Dazu müssen Sie eine *Comperator*-Objekt übergeben. Erstellen Sie dieses und überschreiben Sie die Methode *compare*. Diese nimmt die beiden zu vergleichenden Objekte auf und gibt einen Integer-Wert zurück. Dabei werden die Integer-Werte wie folgt interpretiert:

positiv	→ erstes Objekt ist grösser
0	→ beide Objekte sind gleich
negativ	→ erstes Objekt ist kleiner

3) Kopieren Sie Ihr Projekt und ersetzten Sie Ihre *HashMap*-Implementierung mit einer *ArrayList* aus. Messen Sie in beiden Varianten die Zeit und vergleichen Sie diese. Beachten Sie, dass Sie beide Varianten in einem separatem Programmdurchlauf ausführen, ansonsten können Optimierungseffekte die Werte «verfälschen».