Algorithmen und Datenstrukturen

1. Praktikumsaufgabe

1. Definieren Sie eine Klasse `Student`, die die Daten
   - Matrikelnummer (`int`)
   - Name, Vorname (`char[10]`, nicht `string`)
   - Geburtsdatum (`char[9]` für 8 Zeichen oder `int`, 06.01.75 oder 60175)
   - ...

   enthält. Als Operationen definieren Sie (außer der automatisch generierten Zuweisung) die Vergleichsoperatoren `==, !=, <=, >=, <, >`, die nur die Matrikelnummern vergleichen.
   Erstellen Sie neben geeigneten Konstruktoren auch Zugriffsfunktionen für die Komponenten und Ein-/Ausgabe-Methoden `write(ostream& ostr)const` und `read(istream& istr)`, z.B.

   ```
   virtual void read(istream& istr) {istr >> name >> vorname >> ...;}
   ```

   Übernehmen Sie außerdem die Ein- und Ausgabestromoperatoren (global, keine Methoden!)

   ```
   ostream& operator << (ostream& ostr, const Student& stud)
      {stud.write(ostr); return ostr;}

   istream& operator >> (istream& istr, Student& stud)
      {stud.read(istr); return istr;}
   ```

Eingebettete Systeme mit Realzeitanforderungen vermeiden oft die dynamische Speicherallokation. Die Größe von Collections (Lineare Listen, Vektoren, Arrays, ...) muss bereits zur Compilezeit mittels Template-Parametern festgelegt werden. New-Operationen sind verboten. Sie sollen Lineare Listen mit fester Maximalgröße mittels Arrays realisieren.
Eine naive Implementierung würde für Einfüge- und Löschoperationen alle nachfolgenden Elemente verschieben (Zeitkomplexität O(n) statt O(1)). *Cursor*-Implementierungen versuchen, die Zeitkomplexität auf Kosten der Speicherkomplexität zu reduzieren. Die Grundidee ist, die Zeigerverwaltung nicht der Speicherverwaltung zu überlassen, sondern dem Programmierer aufzubürden. Statt Listenzeiger werden Vorgänger- und Nachfolgeradressen (Indizes) zusammen mit den Daten in einem Array verwaltet. Gleichzeitig wird eine Freispeicherliste, die den unbenutzten Teil des Arrays umfaßt, aufgebaut.

	index	data	prev	next
start_data →	0	A	*	1
	1	B	0	6
start_free →	2		*	4
	3	D	6	*
	4		2	5
	5		4	*
	6	C	1	3

start → [A] ↔ [B] ↔ [C] ↔ [D]
(entsprechende doppelt verkette lineare Liste)

Neue Elemente werden am Anfang der Freispeicherliste eingefügt und Indizes prev , next und start_free entsprechend "verbogen", z.B. Einfügen von X vor B:

	index	data	prev	next
start_data →	0	A	*	2
	1	B	2	6
	2	X	0	1
	3	D	6	*
start_free →	4		*	5
	5		4	*
	6	C	1	3

start → [A] ↔ [**X**] ↔ [B] ↔ [C] ↔ [D]
(entsprechende doppelt verkette lineare Liste)

Analog wird ein Element gelöscht, indem der next-Index des Vorgängers auf den Nachfolger umgelenkt wird und das gelöschte Element am Kopf der Freispeicherliste eingeklinkt wird und analog prev.

2. Definieren Sie eine Template-Klasse `CursorList`, die das folgende Interface implementiert:[1]

```
template <class T> class List {
public:
  typedef T value_type;
  typedef ListIterator<T> iterator;[2]

  bool empty() const;
  int size() const;
  T& front() const;
  void push_front(const T &);   // add a new value to the front of a list
  void pop_front();

  iterator begin() const;
  iterator end() const;
  iterator insert(iterator itr, const T& value);// insert before itr
  iterator erase(iterator start, iterator stop);// stop exclusive
  iterator erase(iterator itr);                      // return ++itr
};
```

   Die Methoden

```
   iterator insert(iterator itr, const T& value);     // insert before itr
   iterator erase(iterator itr);                      // return ++itr
```

   sollen nur konstante Zeit benötigen (siehe Übungsaufgabe zur Vorlesung).

3. Definieren Sie eine zugehörige Klasse `CursorIterator` mit der Schnittstelle[3]

```
template <class T> class ListIterator {
   typedef ListIterator<T> iterator;[4]
public:
   T& operator *();
   iterator& operator = (const iterator& rhs);
   bool operator != (const iterator& rhs) const;
   bool operator == (const iterator& rhs) const;
   iterator& operator ++();
   iterator  operator ++(int);  // postfix operator, dummy parameter
};
```

4. Definieren Sie eine globale Funktion

```
   template<typename Iterator, typename T>
   Iterator find(Iterator start, Iterator stop, const T& value);
```

   welche die erste Position mit dem Wert `value` zurückgibt bzw. `stop` bei erfolgloser Suche.

5. Schreiben Sie ein Hauptprogramm, das per Menü Studentendaten von `cin` lesen und in einer `CursorList` speichern, anschließend Matrikelnummern von `cin` lesen und die zugehörigen Mitgliedsdaten in der Liste suchen und auf `cout` ausgeben kann. Testen Sie mit weiteren Menüpunkten alle Listen-Methoden.

6. (Nur für Praktikumsgruppen mit drei Studenten)
   Ergänzen Sie die Studentendaten um eine
   - Liste der abgelegten Prüfungen mit ihren Noten (vordefinierte Klasse `pair<string,char>`) .

   Beachten Sie, daß die Prüfungsdaten wieder in einer Cursor-Liste gespeichert werden.
   Modifizieren Sie auch die überladenen Operatoren `>>` und `<<`. Überlegen Sie sich eine einfache Lösung zur Erkennung des Endes der Prüfungen bei der Eingabe.

---

[1] Sie brauchen `CursorList` nicht von `List` abzuleiten. `Cursorlist` soll nur die in `List` aufgeführten Methoden haben.

[2] In `CursorList` (nicht von `List` abgeleitet) lautet diese Zeile: `typedef CursorIterator<T> iterator;`

[3] `CursorIterator` braucht nicht von `ListIterator` abgeleitet zu werden.

[4] In `CursorIterator` lautet diese Zeile: `typedef CursorIterator<T> iterator;`