UE Algorithmen und Datenstrukturen (in C++)

Abgaben der Beispiele grundsätzlich per `git`, zu Beginn evtl. noch parallel per Moodle. Beachte jedenfalls die Abgabemodalitäten unter:
`https://mediacube.at/wiki/index.php/ILV_Datenstrukturen_und_Algorithmen_-_SS_2013`.

Aufwärmaufgaben

Achtung: für diese Aufgaben ist noch *keine* Abgabe erforderlich. Zu bearbeiten bis 01.03.13. Da noch kein Moodle-Kurs für dieses Semester existiert, arbeiten wir solange mit dem Programmieren-Kurs des vorigen Semesters.

Inhalt: Kommandozeilenparameter und Konstruktoren/Destruktoren/Copy-Konstruktoren.

1. Leseaufgabe: Studiere die Dateien (in Moodle) bzw. die Internetseiten:
   - `CommandLineArguments-Slide.pdf`,
   - `hello.cpp`,
   - `PassingCommandLineArguments.pdf`, und
   - `http://www.cs.wmich.edu/~sdflemin/instr_pgs/cmdln_args/index.html`
2. Studiere die Datei (in Moodle) `DemoConstructorsDestructors.cpp`. Kompiliere die Datei einmal mit der gcc-Kommandozeilenoption `-fno-elide-constructors` und einmal ohne. Studiere genau die Unterschiede. Info zu dieser Kommandozeilenoption unter:
   `http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/C_002b_002b-Dialect-Options.html`.
3. Versuche deinen PC/Laptop mit der nötigen `Git`-Software auszustatten (siehe das LV-Wiki) und dann mit `git` eine Datei (zB `Aufwaermen.txt`) in dein Algodat-Repository zu pushen. Für weitere Infos zu `git` konsultiere das LV-Wiki bzw. die Informationen von Brigitte Jellinek, welche du noch erhalten wirst.

1. Übungsblatt

Abgabe der Beispiele bis Freitag 22.03.13 8:00 per `git` UND Moodle.

1. (2P) Verwende die Klasse `IntArray` (source code in Moodle):

   (a) Trenne `.cpp` und `.h` Datei.

   Programmiere Debug-Ausgaben für die Konstruktoren und den Zuweisungsoperator, um anhand verschiedener Anweisungen in `main()` nachzuvollziehen, wann sie jeweils aufgerufen werden. Kommentiere die entsprechenden Stellen im Code, welche zu den Aufrufen führen. (Den Code von a) abgeben.)

   (b) Beschreibe, was passiert, wenn der Referenzoperator `&` beim Zuweisungsoperator weggelassen wird. Begründe!

   (c) Erläutere, was passiert, wenn beim Zuweisungsoperator die Zeile `return *this;` weggelassen wird. (Rückgabewert auf `void` ändern.)

   (d) Erläutere, was passiert, wenn Folgendes für drei bestehende IntArrays `a, b, c` in `main()` aufgerufen wird: `a = b = c;`. Beschreibe den Zusammenhang zur vorigen Aufgabe.

   (e) `const`-Klassenmethoden erlauben ja nicht, dass member-Variablen verändert werden.

   Finde durch Ausprobieren heraus, ob das auch für das Dereferenzieren eines Pointers auf ein Array gilt. Konkret geht es also um die Frage, ob folgende Funktion kompiliert:

   ```
   void modifyArray(int idx, int value) const {
       mem[idx] = value;
   }
   ```

   (f) Beachte, wie bei `getAt` die Referenz auf den Rückgabewerts dazu benützt wird, in `main` das Array `a1` an Index 0 und 1 zu beschreiben!

   Beachte, wie der copy constructor mithilfe des assignment operators implementiert wurde.

   Beachte weiters, dass beim copy constructor `mem(0)` gesetzt wird. Warum? Hilfestellung: copy constructor ruft assigment operator auf. Dieser wiederum ruft `clear()` auf. Was würde also passieren, wenn zu diesem Zeitpunkt `mem != 0` wäre?

   Detaillierte Antworten zu den Aufgaben b) bis f) in das Textfile `antworten.txt` schreiben und abgeben.

2. (2P) Demonstriere die Notwendigkeit eines eigens programmierten copy constructors und assignment operators für die Klasse `IntArray`, indem künstlich **zwei Bugs** eingebaut werden:

   Schreibe `IntArray` um in eine Klasse `IntArrayShallowCopy`, welche wie vorher im Konstruktor Speicher reserviert, im Destruktor wieder freigibt.

   Benutze nun allerdings den (impliziten) default copy constructor und default assigment operator, um zu zeigen dass...

   (a) ... bei einer Kopie des Arrays per *implizitem default copy constructor* die Manipulation der Elemente des Original-Arrays auch zu veränderten Elementen in der Kopie führt! (Bug1)

   (b) ... bei einer Zuweisung eines Arrays an ein anderes per *assignment operator* ebenso eine Änderung des Original-Arrays eine Änderung der Kopie bewirkt. (Bug2)

   Gestalte den Code in `main` so, dass bei Aufruf des Programms beide Bugs klar ersichtlich werden, und der Code sauber und ordentlich dokumentiert ist.

3. (6P) Implementiere eine Klasse `MyString`, welche den Datentyp `string` simuliert. (Benutze als Basis `IntArray`.)

   (a) Die Klasse kapselt ein `char`-Array `data` einer gewissen, erst zur Laufzeit bekannten Größe `unsigned int size`.

   (b) Das gekapselte `char`-Array **muss** mit dem extra Null-Terminierungszeichen[1] enden (`\0`)!

   (c) Der default-Konstruktor setzt `size` und `data` auf 0.

   (d) Ein weiterer Konstruktor `MyString(const char* arr)` initialisiert `data` mit dem übergebenen Array.

   Hinweis: die Anzahl der tatsächlichen Zeichen in von `arr` kann man mit `strlen` [2] ermittelt werden. (Null-Terminierungszeichen wird nicht mitgezählt.)

   (e) Der Destruktor gibt den Speicher wieder frei.

   (f) Die Klasse bietet Methoden an, um auf Elemente des Arrays per Index zuzugreifen bzw. zu manipulieren (etwa `get(unsigned int i)` oder `set(unsigned int i, char c)`.

   (g) Eine Methode `print()` gibt den String in der Konsole aus.

   (h) Ungültige Indizes müssen erkannt und mit einer Fehlermeldung abgewiesen werden (im Falle von `get` wird zusätzlich ein Dummy-Wert zurückgegeben).

   (i) deep-copy Semantik ist erforderlich:

      i. Überlade den Zuweisungsoperator entsprechend.

      ii. Programmiere einen entsprechenden copy-Konstruktor.

   (j) Eine Methode `void append(const MyString& otherStr)` hängt das Array in `otherStr` an das Array der aktuellen Instanz an.

   (Hinweis: Speicher korrekter Größe neu anfordern, altes Array kopieren, den Inhalt von `otherStr` anhängen; Terminierungszeichen berücksichtigen; am Ende den alten Speicher freigeben.)

   (k) Verwende `const` überall, wo dies angebracht ist!

   (l) Speicherverletzungen jedweder Art müssen mit unter allen Umständen vermieden werden!

   (m) Verwende **asserts** in **separaten** Testmethoden, um zu überprüfen, dass die deep-copy Semantik des Zuweisungsoperators und des copy-Konstruktors sichergestellt ist, und sämtliche Methoden korrekt arbeiten.

   Hinweis: Die Testmethoden müssen so gestaltet sein, dass ein Auskommentieren des Zuweisungsoperators bzw. des copy-Konstruktors zum Programmabbruch aufgrund fehlgeschlagener assertions führen. **Achte auch auf Spezialfälle wie Arrays der Größe 0 bzw. Null-Pointer!**

   (n) Programmiere eine separate `.cpp` und `.h` Datei.

   (o) Die Klasse soll etwa so funktionieren[3]

```
MyString leer; //--> default-Konstruktor wird aufgerufen
leer.set(0, 'A'); //--> Fehlermeldung ausgeben, da das Array leer ist
MyString hallo("Hallo"); //--> zweiter Konstruktor wird aufgerufen
hallo.set(0, 'h'); //--> erster Buchstabe wird lower case
hallo.print(); //--> Ausgabe von "hallo"
hallo.append(" du!"); //String wird angehaengt
hallo.print(); //-> Ausgabe von "hallo du!"
cout << hallo.get(9);  //--> Fehlermeldung, falscher Index
MyString cpy = hallo; //--> copy-Konstruktor wird aufgerufen, deep copy
leer = hallo; //deep copy mit Zuweisungsoperator
hallo.set(8, '?'); //hallo wird modifiziert
leer.print(); //Ausgabe von "hallo du!"--> Beweis, dass deep copy angelegt wurde
cpy.print(); //Ausgabe von "hallo du!"--> -''-
MyString cpy2(cpy); //eine deep copy von cpy wird angelegt
```

[1] Siehe `http://www.cs.bu.edu/teaching/cpp/string/array-vs-ptr/`, vor allem Punkt 1 und 4.
[2] `http://www.cplusplus.com/reference/clibrary/cstring/strlen/`
[3] Diese Testfälle sind nur exemplarisch, es müssen in den Testmethoden mehr Fälle behandelt werden!

2. Übungsblatt

Abgabe der Beispiele bis 5.4.13 8:00 per `git` und Moodle. Thema dieses Übungsblatts: *Vererbung*, *Exceptions*, *assert* und geschicktes *Zählen*.

4. (2P) Schreibe eine Klasse `Article` mit einigen Zusatzklassen:
   (a) Sie soll eine Artikelnummer, einen Namen, einen Preis und entsprechende Methoden zum Setzen, Abfragen und Verändern von Namen und Preis besitzen.
   (b) Weiters eine Methode `print()` zum Ausgeben aller Informationen eines Artikels und `addTax(float p)` zum Aufschlagen einer Steuer von `p` Prozent auf den Preis. Demonstriere das Abfangen ungültiger Benutzereingaben mithilfe von Exceptions.
   (c) Leite von `Article` die Unterklassen `Book`, `DVD` und `PC` mit jeweils mindestens zwei zusätzlichen eigenständigen Attributen und Methoden ab (zB `author` für den Autor eines Buches). Alle Unterklassen müssen die Methode `print()` an ihre Gegebenheiten anpassen und die jeweiligen Zusatzinformationen ausgeben.
   (d) Achtung: zur Ausgabe der Informationen der Basisklasse `Article` mithilfe des Scope-Operators `::` die Methode der Basisklasse aufrufen, *nicht* den Code duplizieren. Achte auch auf den korrekten Aufruf des Konstruktors der Basisklasse im Konstruktor des Unterklasse.
   (e) Programmiere weiters eine Methode `edit()`, welche den Artikel per Konsoleneingaben editieren lässt. Bei allen Attributen des Artikels (außer der Artikel-Nr) wird gefragt ob es geändert werden soll, und bei Antwort `j` wird per `cin` der zu ändernden Werte eingelesen.
   (f) Vermeide bei den überschriebenen Methoden der Unterklassen unnötige Code-Verdoppelung, indem ggf. explizit die Methode der Oberklasse aufgerufen wird.
   (g) Verwende `const` überall, wo angebracht.
   (h) Illustriere das korrekte Funktionieren übersichtlich in `main()`.
   (i) Trenne Header- und Implementierungsfiles.
5. (2P) Erweitere dein modifiziertes `IntArray`-Beispiel (Nr. 1) um Exceptions:
   (a) ...indem `int& getAt(unsigned idx)` in zwei Methoden aufgespalten wird:
      i. `int getAt(unsigned int idx) const;` liefert den Wert an Index `idx`.
      ii. `void setAt(int value, unsigned int idx);` schreibt den Wert `value` an Index `idx`.
   (b) Erstelle weiters eine Methode `unsigned in getSize()`, welche die aktuelle Größe des `IntArrays` zurückgibt.
   (c) Erstelle nun manuell eine Exception-Klasse (wieder Header und Implementierung trennen): die Klasse `WrongIndexException`, welche die Attribute `string message`, `unsigned int invalidIndex` und `trueSize` speichert, sowie Methoden `string getMessage()`, `unsigned int getWrongIndex()` und `unsigned int getTrueSize()`. Der Konstruktor muss alle drei Parameter übergeben bekommen.
   (d) Die Exception-Klasse soll nun **statt** dem `assert`[4] gültige Index-Eingaben sicherstellen. Bei falschem Parameter (also wenn `idx >= size`) soll *innerhalb* von `getAt` bzw. `setAt` die Exception geworfen werden[5] Beim Werfen der Exception müss eine aussagekräftige `message` übergeben werden (zB „`Falscher Index beim Aufruf von setAt:`"), sowie der falsche übergebene Index, sowie die aktuelle `size` (zB `"wrong index: 3, size: 3."`).
   (e) In `main()` soll demonstriert werden, wie mithilfe von `catch` auf die fehlerhaft eingebenenen Indizes korrekt reagiert wird, indem jeder Aufruf von `getAt` bzw. `setAt` mit `try/catch` umschlossen wird, und beim Fangen dem Benutzer eine genaue Mitteilung gemacht wird, welcher Index bei welcher Methode und welcher Size den Fehler verursacht hat.

[4] Erinnere dich: `assert` soll die Korrektheit der Programmlogik sicherstellen, nicht die Gültigkeit der Benutzereingaben!
[5] Anstatt eines Programmabbruchs wie bei `assert`.

6. (5P) Komprimiere einen String, welcher lediglich aus Buchstaben ('a–z' und 'A–Z') besteht, mithilfe der sogenannten *Lauflängenkodierung*[6].

   (a) Diese Kodierung arbeitet wie folgt: enthält der String drei oder mehr identische Zeichen hintereinander, so wird diese Sequenz durch ein einziges Zeichen und die Länge der Sequenz ersetzt. Es wird also etwa ein `aaaa` zu `a4`. Wenn ein Zeichen nur ein- oder zweimal vorkommt, wird nichts verändert.

   (b) Beispiel: aus `AaaaddddDDDEfffxxyZZZZZ` wird `Aa3d4D3Ef3xxyZ5`, aus `GGGGGGGGG` wird `G9`, aus `ABCDE` wird `ABCDE`.

   (c) Entwickle ein Kommandozeilenprogramm namens `shrink`, welches nach diesem Schema einen String komprimiert oder dekomprimiert. Es soll mit einem Parameter (`-c` bzw. `-d`) gesteuert werden, ob die Eingabe komprimiert (compress) oder dekomprimiert (decompress) werden soll. Stichwort: *command line arguments*: `argc` bzw. `argv`!

   (d) Geforderte Funktionsweise:
   der Aufruf `shrink -c AaaaddddDDDEfffxyZZZZZ` soll die Ausgabe `Aa3d4D3Ef3xyZ5` und `shrink -d Aa3d4D3Ef3xyZ5` die Ausgabe `AaaaddddDDDEfffxyZZZZZ` liefern.

   (e) Falls ein Zeichen öfter als neunmal hintereinander vorkommt (zB das `a` 15 Mal) genügt vorerst eine Komprimierung der Form `a9a6`[7].

   Wichtig: jeder komprimierte String muss mit `shrink -d` wieder **korrekt** dekomprimiert werden können!

   (f) Teste das Komprimieren und Dekomprimieren **ausgiebig** mit **assert**s in separaten Testfunktionen.

7. (2P) Erweitere den Code von `shrink` um Exceptions:

   (a) Behandle ungültige/fehlende Parameter von der Kommandozeile per eigener Exception-Klasse, zB `InvalidCommandLineException`.

   (b) Behandle ebenso ungültige Formate des Inputs (jeweils für `-c` und `-d`), zB in `InvalidFormatException`. (Überlege dazu, wie gültiger Input aufgebaut ist und behandle jedwede Abweichung.)

   (c) Programmiere eine gemeinsame Basisklasse für alle Exception-Klassen, zB `ShrinkException`. Nutze Vererbung geschickt.

   (d) Teste und demonstriere **systematisch** und **übersichtlich** in `main()`, dass alle Exceptions korrekt geworfen werden, und die Exceptions bei korrektem Verwenden von `shrink` **nicht** geworfen werden.

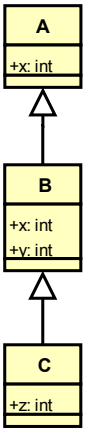
[6] Auch bekannt als *run length encoding* bzw. *RLE*.

[7] Da so die Dekomprimierung einfacher wird. Wer `a15` als komprimierten String ausgibt, muss in jedem Fall dazu die korrekte Dekompression programmieren! Um dies zu vereinfachen, könnten `atoi`, `stringstream`s bzw. `c_str()` hilfreich sein, oder mit einer Schleife manuell die Zahl aus den Ziffern zusammenbauen.

3. Übungsblatt

Abgabe der Beispiele bis 12.04.13 8:00 per `git` und Moodle. Thema dieses Übungsblatts: *Vererbung*, *Komposition* und *Polymorphismus*, sowie *memory debugging*.

8. (2P) Implementiere die Klassen des unten befindlichen UML-Diagramms.
   (a) Erzeuge in `main` ein Objekt `c` vom Typ `C`.
   (b) Gib die Adressen von den Membervariablen, welche von `A`, `B` und `C` stammen, aus.
   (c) Schreibe eine Funktion `test`, welche einen Parameter `B& b` bekommt und rufe `test(c)` in `main` auf. In `test` werden die Adressen der Membervariablen von `b`, welche von `A` und `B`, stammen ausgegeben.
   (d) Interpretiere das Ergebnis in `Antworten.txt`.



9. (3P) Schreibe/adaptiere die Klasse `MyStringPrinter` aus dem Wintersemester, welche nun einen `MyString s` speichert (Komposition!) und dann ausgeben kann.
   (a) Konstruktoren von `MyStringPrinter` müssen Objekte vom Typ `MyString` ebenso akzeptieren wie String-Literale (`const char*`) oder Strings.
   (b) Schreibe eine `get()` Funktion, welche `s` als constant reference liefert.
   (c) Die Methode `print()` gibt `s` unformatiert in der Konsole aus.
   (d) Eine Methode `info()` gibt die Info aus, dass `MyStringPrinter` den String unformatiert ausgibt.
   (e) Schreibe drei verschiedene *Unterklassen*, welche die Methode `print` so anpassen, dass sie den String jeweils verschieden formatiert ausgeben: z.B. ein schöner Rahmen drumherum, zwischen den Buchstaben Leerzeichen, unterstrichen etc.
   (f) Die Methode `info` soll ebenfalls so angepasst werden, dass die Info ausgegeben wird, um welche spezielle Formatierung es sich jeweils handelt.
   (g) Benenne die Unterklassen aussagekräftig! (zB `UnderlineStringPrinter`).
   (h) Demonstriere das richtige Funktionieren der Konstruktoren und Methoden für die Basis- und Subklassen sauber und übersichtlich in `main`.
   (i) Polymorphismus: erweitere die Klassen (Schlüsselwort `virtual`), sodass fünf `StringPrinter`-Pointer unterschiedlichen dynamischen Typs in einem Array von `StringPrinter`-Pointern gespeichert werden.

   Beim Durchlauf durch das Array soll dann das Kommando `stringPrinters[i]->print();` bewirken, dass `Hallo` fünfmal unterschiedlich — dem *dynamischen* Typ entsprechend — ausgegeben wird. Ebenso mit `info()`.

   In `main()` muss das korrekte Funktionieren übersichtlich illustriert werden.
   (j) Trenne wiederum `.cpp` und `.h` Dateien.

(k) Bei welchen der folgenden Aufrufen wird die Funktion der Oberklasse, bei welchen der Unterklasse aufgerufen? Wo tritt ein Kompilierfehler auf? Warum? Wo ist Polymorphismus aktiv, wo nicht? Was ist jeweils der dynamische, was der statische Typ? (Bitte wieder in `Antworten.txt` geben.)

```
1   StringPrinter strPr("Servus");
2   strPr.print();
3
4   StringPrinter strPrPtr* = new UnderLineStringPrinter("Hallo");
5   strPrPtr->print();
6
7   UnderLineStringPrinter uStrPr("Hi!");
8   strPrPtr = &uStrPr;
9   strPrPtr->print();
10
11  StringPrinter& strPrRef= uStrPr;
12  strPrRef.print();
13
14  UnderLineStringPrinter* uStrPrPtr = new UnderLineStringPrinter("Hola!");
15  strPrPtr = uStrPrPtr;
16  strPrPtr->print();
17
18  uStrPrPtr = &strPr;
19  uStrPrPtr->print();
20
21  strPr = uStrPr;
22  strPr.print();
23
24  uStrPr = strPr;
25  uStrPr.print();
```

10. (5P) Erweitere das Programm mit der `Article`-Klasse um Polymorphismus und Komposition:

    (a) Schreibe eine Klasse `Store`, welche Pointer auf `Article`-Objekte in einem Array sowie deren Anzahl speichert.

    (b) `Store` muss es erlauben, einen `Article` neu hinzuzufügen, sowie einen `Article` mit einer gewissen Artikelnummer zurückzugeben.

    (c) Mache `print()` und `edit()` mithilfe von `virtual` polymorphismus-fähig. Stelle sicher, dass je nach dynamischem Typ die korrekten Funktionen aufgerufen werden [8].

    (d) `Store` soll auch bei *allen* Artikeln (lso auch *DVDs*, *Books* etc.) auf den Preis eine Steuer aufschlagen können (codeaddTaxes(float percent)), sämtliche Informationen über *alle* Artikel ausgeben (`printAll()`), und den Gesamtpreis *aller* gespeicherten Artikel berechnen können (`computeTotalPrice()`). Tipp: innerhalb von `printAll()` das Pointer-Array mit einer Schleife durchlaufen, und auf jeden Pointer `print()` aufrufen.

    (e) Füge Methoden `countBooks()`, `countDVDs()`,... hinzu, welche mithilfe eines *dynamic casts* die unterschiedlichen Artikeltypen zählen[9].

    (f) Die Methode `printStatistics()` soll unter Verwendung der `count`-Methoden den prozentuellen Anteil der unterschiedlichen Artikeltypen ausgeben.

    (g) Verwende `const` überall, wo angebracht.

    (h) Illustriere das korrekte Funktionieren übersichtlich in `main()`.

11. Studiere je nach Betriebssystem die beiden Memory Debugger `Valgrind` bzw. `DrMemory`: lies und arbeite die Anleitungen[10] durch, installiere die Software, kompiliere das fehlerhafte Programm `buggy_program.cpp` mit der Option `-g`, und wende den Memory Debugger auf die produzierte Executable `buggy_program.exe` an. Studiere die Ausgabe, und versuche damit die Bugs zu verstehen. (*Keine Abgabe erforderlich.*)

[8] Mit dem Aufruf `articles[i]->edit();` soll man also — je nach dynamischem Typ — die Attribute von `DVD`, `Book`, etc. verändern können. Vermeide bei den überschriebenen Methoden der Unterklassen Code-Verdoppelung, indem ggf. explizit die Methode der Oberklasse aufgerufen wird.

[9] *Nicht* mit einem type field in `Article` arbeiten!

[10] siehe `mem_debugging.zip` im Moodle

4. Übungsblatt

Abgabe der Beispiele bis 19.04.13 8:00 per `git`. Thema dieses Übungsblatts: Polymorphismus, operator overloading, Templates.

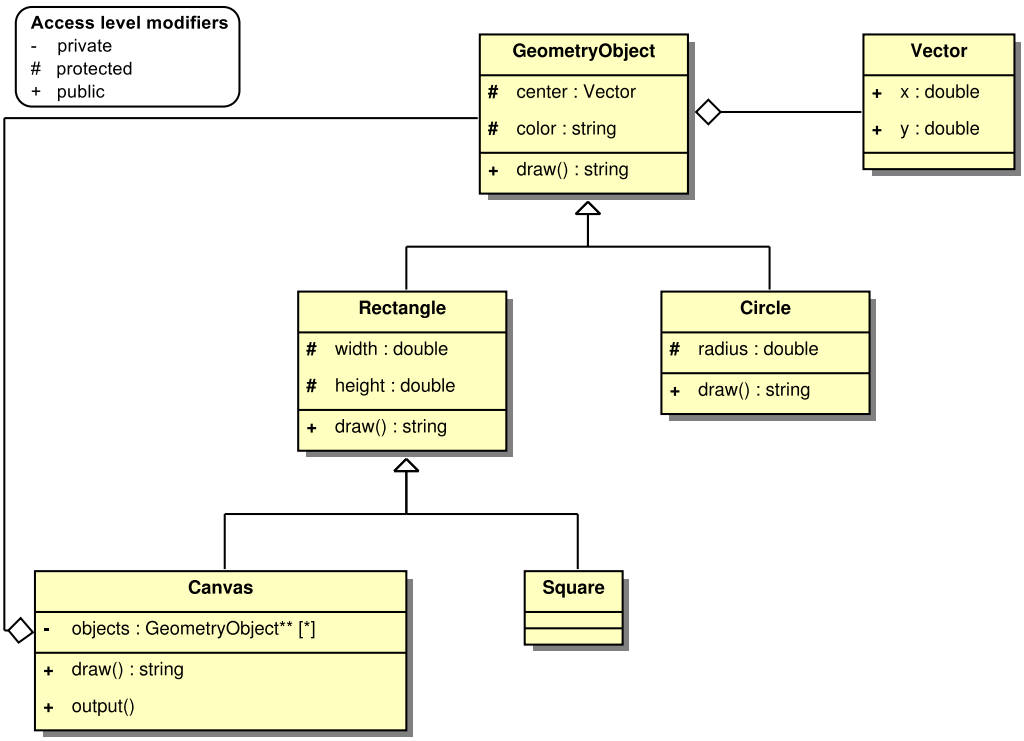
12. (6P) Schreibe ein Programm, das mithilfe der Software *ImageMagick*[11] [12] einfache Bilder bestehend aus geometrischen Objekten erzeugen kann.

    (a) Schreibe dafür zunächst eine Klasse `GeometryObject`, die eine pure virtual function `string draw()` hat.

    (b) Ein `GeometryObject` hat weiters eine Farbe (`color`), die als `string` gespeichert wird[13], sowie einen Mittelpunkt `center` vom Typ `Vector`.

    (c) Leite dann von der Klasse `GeometryObject` die Unterklassen `Rectangle`, `Square`, `Circle` und `Canvas` ab:

    - Ein `Rectangle` hat eine Breite (`width`) und Höhe (`height`).
    - Ein `Square` ist ein Spezialfall eines `Rectangle`s und sollte keinen Code von `Rectangle` duplizieren.
    - Ein `Circle` hat einen Radius (`radius`).
    - Die Klasse `Canvas` besitzt ein Array `objects` von Zeigern auf `GeometryObject`s.
    - Ein `Canvas` ist von einem `Rectangle` abgeleitet, und hat somit die vererbten Attribute Breite (`width`) und Höhe (`height`).
    - `Canvas` muss das Hinzufügen von Geometry Objekten erlauben: mit der Methode `addGeometryObject(GeometryObject *object)`, welche den übergebenen Pointer zu `objects` hinzufügt.



    (d) Die pure virtual function `GeometryObject::draw()` soll einen String zurückgeben und soll durch die unten angeführten Unterklassen überschrieben werden, sodass folgendes Resultat zurückgeliefert wird:

[11] `http://www.imagemagick.org/script/binary-releases.php`. Der Link zum Windows-Installer ist `http://www.imagemagick.org/script/binary-releases.php#windows`. Nach der Installation muss ggf. der Rechner neu gebootet werden, damit das benötigte Konsolen-Programm `convert` von der Kommandozeile aus verfügbar ist.

[12] Alternativ steht unter `http://media.zero997.com/convert.php` eine Online-Version von `convert` zur Verfügung

[13] Mögliche Farben sind `"red"`, `"green"`, `"blue"`, `"yellow"`,`"black"`, `"white"`, `"transparent"`.

- für `Rectangle`: `fill <color> rectangle <x0>,<y0> <x1>,<y1>`
- für `Circle`: `fill <color> circle <xc>,<yc> <xc + radius>,<yc>`
- für `Canvas`: die durch Leerzeichen getrennten Resultate von `draw()` aller `GeometryObjects`, welche sich in im Array `objects` befinden.

Dabei gilt:

- `<color>` ist der Name der Farbe
- `<x0>,<y0>` sind die Koordinaten der linken oberen Ecke
- `<x1>,<y1>` sind die Koordinaten der rechten unteren Ecke
- `<xc>,<yc>` sind die Koordinaten des Mittelpunktes

(e) Implementiere im `Canvas` zusätzlich eine Methode `output()`, die folgenden String ausgibt:

```
convert -size <width>x<height> xc:transparent -draw "<output von draw()
des Canvas>" output.gif
```

(Dieser String kann dann in der Konsole abgesetzt werden, um die Bilddatei `output.gif` zu produzieren.)

(f) Achte auf Bugfreiheit: keine memory leaks, keine unerlaubten Speicherzugriffe,...

(g) Produziere auf diese Weise ein paar verschiedene Output-Bilder und gib sie im `git` ab.

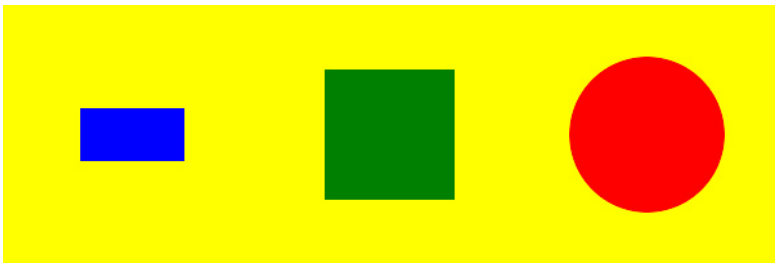
Ein Beispiel, wie die Klassen in `main` benützt werden könnten:

```
Canvas image(Vector(300,100), string("yellow"),600,200);

Rectangle rectangle( Vector(100,100), 80, 40, string("blue"));
Square square( Vector(300,100), 100, string("green"));
Circle circle( Vector(500,100), 60, string("red"));

image.addGeometryObject(&rectangle);
image.addGeometryObject(&square);
image.addGeometryObject(&circle);

image.output();
```

Der Code liefert folgenden (textuellen) Output:

```
convert -size 600x200 xc:transparent -draw "fill yellow rectangle 0,0 600,200
fill blue rectangle 60,80 140,120 fill green rectangle 250,50 350,150 fill red
circle 500,100 560,100" output.gif
```

Und dieser (textuelle) Output wiederum liefert, wenn er als Kommando in der Konsole abgesetzt wird, folgendes Bild `output.gif`:



13. (2P) Studiere die Klasse `Color` (siehe Moodle) die Verwendung und Funktionsweise der `static`-Methoden bzw. Attribute.

    Erweitere dann die Klasse um folgende Operatoren (welche als Methoden von `Color` implementiert werden sollen):

    (a) `Color Color::operator+(const Color& other) const`
    Dieser (binäre) Operator mischt zwei Farben nach der additiven Farbmischung mithilfe des arithmetischen Mittels. Das Mischen von `(r1, g1, b1, a1)` und `(r2, g2, b2, a2)` ergibt also die Farbe `( (r1 + r2)/2, (g1 + g2)/2, (b1 + b2)/2, (a1+a2)/2)`. Hier stehen die Kürzel r, g, b, a natürlich für Rot, Grün, Blau, Alpha.

    (b) `Color Color::operator~() const`
    Dieser (unäre) Operator gibt die invertierte Farbe zurück. Aus `(r, g, b, a)` wird also `(1-r, 1-g, 1-b, a)`. Es bleibt hier der Alpha-Wert gleich.

    (c) `bool Color::operator==(const Color& other) const`
    Der Vergleichsoperator soll zurückgeben, ob die Farben identisch sind.

    (d) `Color& Color::operator+=(const Color& other)`
    Kombination aus + und =.

    Hinweis: für das Überladen neuer Operatoren dürfen selbstverständlich bereits bestehende überladene Operatoren verwendet werden [14].

    Beispielhafte Demonstration der Funktionsweise der Klasse in `main` (das Testen des Programmes in dieser Art und Weise genügt bei dieser Aufgabe):

```
Color c1 = Color::fromRGB(0.2, 0.6, 0.9);
Color c2 = ~c1;         //c2 ist jetzt (0.8, 0.4, 0.1, 1.0)
Color c3 = Color::fromRGB(0.4, 0.7, 0.9);
Color c4 = c2 + c3;     //c4 ist jetzt (0.6, 0.55, 0.5, 1.0)
assert((~~c3) == c3);
assert((~c1) == c2);
assert(!(c1 == c2));
c1 += c2;               //c1 ist jetzt (0.5, 0.5, 0.5, 1.0)
```

14. (1P) Implementiere Template Funktionen `myswap` (Vertauschung zweier Parameter), `mymin` (Minimum), `mymax` (Maximum). Die Funktionen sollen mit generischen Datentypen funktionieren.

    Funktionsweise in `main`:

```
int a = 5; int b = -3;
myswap(a, b);
cout << a << " " << b; // gibt "-3 5" aus

string c = "Hallo"; string d = "Welt";
myswap(c, d);
cout << c << " " << d; // gibt "Welt Hallo" aus

float x = -3.1415; float y = 2.718;
myswap(x, y);
cout << x << " " << y; //gibt "2.718 -3.1415" aus

cout << mymin(5, -14) << " " << mymax(5, -14); //gibt "-14 5" aus
cout << mymin('z', 'a') << " " << mymax('z', 'a'); ; //gibt "a z" aus
cout << mymin(-5.0, 12.34) << " " << mymax(12.34, -5.0); //gibt "-5.0 12.34" aus
```

[14] Siehe evtl. auch `http://www.cs.caltech.edu/courses/cs11/material/cpp/donnie/cpp-ops.html`

15. (3P) Erweitere den Code der Template `Array`-Klasse (siehe Moodle) um folgende Methoden / Operatoren:
    (a) `operator==`: Test auf elementweise Übereinstimmung
    (b) `operator!=`: mithilfe von == implementieren
    (c) `unsigned count_elem(const T& element)`:

    gibt zurück wie oft das Element `element` vorkommt.
    (d) `reverse()`:

    Die Reihenfolge der Elemente *ohne* Verwendung eines Hilfsarrays umdrehen (also *in place*).
    (e) `int findSubArr(const MyArray& arr)`:

    gibt zurück, ob das übergebene Array vollständig enthalten ist. Die Reihenfolge der Elemente ist zu berücksichtigen, ebenso dürfen keine Lücken auftreten.

    Bsp: `[ 2, 3 ]` ist in `[ 1, 2, 3, 4, 5 ]` enthalten; `[ 2, 4 ]` und `[ 3, 2 ]` aber nicht.

    Der Rückgabewert $>= 0$ gibt den Startindex des enthaltenen Arrays an (im Beispiel wäre das 1). Ein negativer Rückgabewert bedeutet, dass `arr` nicht enthalten ist.
    (f) `getSubArr(unsigned fromIdx, unsigned length)`:

    gibt das Teilarray der Länge `length`, beginnend bei Index `fromIdx`, als neues Objekt vom Typ `Array<T>` zurück.
    (g) **Optional/Bonus**: (3P)
        i. `removeElement(const T& element)`:

        entfernt `alle` Vorkommen des Elements `element` im Array, und schließt die Lücken durch elementweises Nach-vorne-kopieren.

        Wichtig: Verhindere in jedem Fall einen Zugriff außerhalb der Arraygrenzen.

        Versuche die Anzahl an benötigten Kopiervorgängen gering zu halten. (Optimal wäre ein einziger Durchlauf mit max. `n` Kopiervorgängen...)
        ii. `allUnique()`: gibt zurück, ob das Array lauter verschiedene Elemente enthält.
        iii. `allEqual()`: gibt zurück, ob das Array lauter identische Elemente enthält.
        iv. `getEveryNthElement(unsigned n)`:

        liefert ein neu erzeugtes `Array`, welches jedes `n`-te Element ($n \geq 1$) des Originalarrays enthält (zB jedes 2-te, jedes 10-te, etc.).
        v. `countDistinctElements()`:

        zählt, wie viele verschiedene Elemente das Array enthält.
        vi. `getDistinctElements()`:

        liefert ein neu erzeugtes `Array`, das nur alle unterschiedlichen Elemente enthält.

    Stelle die Korrektheit der Operatoren / Methoden für unterschiedliche Template Parameter (zB `Array<int>` und `Array<string>`) ausgiebig mit `asserts` in Testmethoden sicher. Es dürfen unter keinen Umständen Speicherverletzungen passieren. Verhindere weiters memory leaks. Optional: benutze Exceptions bei ungültigen Parametern.

16. **Optional/Bonus** (2P). Experimentiere mit dem *Singleton* Design Pattern.
    (a) Studiere, wie das Keyword `static` benutzt wird, um von einer Klasse nur eine einzige Instanz zu erzeugen [15].
    (b) Erweitere den Code um einen Instanzenzugriffs-Zähler samt passender `getNumAccesses()`-Methode. Der Zähler wird jedesmal hochgezählt, wenn `single` nach außen gegeben wird. Demonstriere das richtige Funktionieren in `main()`.

[15] `http://www.codeproject.com/Articles/1921/Singleton-Pattern-its-implementation-with-C`

(c) Erweitere den Code um Ausgaben im Destruktor, und führe `delete`-Statements auf die Pointer in `main` aus. Was passiert? Demonstriere und kommentiere in `main()`.

(d) Erweitere den Code in `method()` um die Ausgabe des `this`-Pointers (also die Adresse der einen Instanz).

(e) Vereinfache den Code, indem du `instanceFlag` mit einer Überprüfung, ob `single` NULL ist, ersetzt.

(f) Versuche in `main()`, mithilfe des Copy-Constructors trotzdem eine weitere Instanz zu erzeugen, indem du in `main()` folgenden Code ausführst: `Singleton *sc3 = &Singleton(*sc1);`

Überprüfe mit `method()`, ob die Erzeugung einer weiteren Instanz tatsächlich geglückt ist.

5. Übungsblatt

Abgabe der Beispiele bis 26.04.13 8:00 per `git`. Thema dieses Übungsblatts: Einfach verkettete Listen mit `tail`-Pointer.

17. (5P) Benutze die Code-Samples der Klasse `SList` aus den VO-Slides (S.5-11), um eine **einfach** verkettete Liste **mit** `tail`-Pointer[16] (welche aber *nicht* doppelt verkettet ist) zu implementieren, welche Strings (`std::string`, statt wie in der VO `int`) in ihren Knoten abspeichert.

    Folgende Operationen sind von der Liste *fehlerfrei* zu unterstützen. (Achtung: `tail` muss bei den bereits vorgegebenen Operationen aus den VO-Slides mitberücksichtigt und korrekt angepasst werden!. Private Hilfsmethoden dürfen natürlich nach Bedarf ergänzt werden.)

    (a) `push_front` und `pop_front`: Einfügen und Löschen am Beginn der Liste.

    (b) `push_back` und `pop_back`: Einfügen und Löschen am Ende.

    (c) `findFirst`: liefert einen Pointer auf den ersten Knoten, der einen übergebenen String speichert (bzw. 0, falls keiner gefunden wird).

    (d) `getPrevNode`: liefert einen Pointer auf den Knoten, welcher sich *vor* dem (per Pointer) übergebenen Knoten befindet. (Bzw. 0, falls nicht existent).

    (e) `remove`: löscht den (per Pointer) übergebenen Knoten und liefert einen Pointer auf den unmittelbar *folgenden* Knoten (bzw. 0, falls nicht existent).

    (f) `removeAfter`: löscht den Knoten, welcher sich *nach* dem (per Pointer) übergebenen Knoten befindet.

    (g) `removeAll`: löscht alle Knoten, die den übergegebenen String speichern.

    (h) `clear`: löscht alle Knoten aus der Liste.

    (i) Ein Konstruktor, welcher einen String bekommt und eine Liste mit einem Knoten aufbaut.

    Achte auf:

    - Ausreichend `asserts` in separaten Testmethoden zum Testen der Korrektheit. Teste insbesondere auch das Funktionieren von „Spezialfällen“ wie leere Listen etc.
    - Getrennte Header- und Implementierungsdateien.
    - Vermeidung von unnötigen Listendurchläufen. (Manchmal ist ein kompletter Listendurchlauf bei einer einfach verketteten Liste aber unvermeidbar.)
    - Vermeidung von memory leaks.
    - Sauberen, schlanken Code, ordentliche Kommentare.

18. (1P) Erweitere obiges Programm um einen Copy-Konstruktor und einen überladenen Assignment-Operator (deep copy Semantik!), sowie Destrukor. (Samt Tests.)

19. (1P) Erweitere obiges Programm um die Operatoren == und != (samt Tests).

[16] siehe S.12

20. (1P) Erweitere obiges Programm um die Funktion `append` (samt Tests): hängt eine übergebene Liste an die aktuelle Liste an.

21. (1P) Erweitere obiges Programm um die Funktion `getReversed` (samt Tests): liefert eine neu erzeugte Liste mit umgedrehter Reihenfolge der Knoten.

22. (1P) **Bonus**: Operatoren + und +=, um einen einzelnen Knoten und eine ganze Liste anzuhängen. (Samt Tests.)

23. (2P) **Bonus**: Implementiere eine Methode `swap`, welche zwei Pointer auf Knoten bekommt, und diese Knoten in der Liste durch *Umhängen* vertauscht. (Wir wollen *nicht* die Daten in den Knoten vertauschen, sondern die Knoten selbst umhängen!) Achte auf Spezialfälle wie Gleichheit der Pointer, `tail` und `head` etc. (Samt Tests.)

24. (3P) **Bonus**: Implementiere das ***Problem des Josephus*** mithilfe einer *zirkulären* Liste (und dazu benötigten Methoden):

    `n` Personen (eine Person sei lediglich durch den `string` seines Namens gegeben) stehen im Kreis. Jetzt wird bis zu einer Zahl `m`, beginnend bei 1, durchgezählt. Die `m`-te Person wird nun „hingerichtet“, in dem Sinn, dass sie aus der Liste gelöscht wird. Bei ihrem Nachfolger beginnend wird wieder bis `m` gezählt, die Person wieder gelöscht usw. Das Programm soll für beliebige $n \geq 1$ und $m \geq 1$ den Namen der letzten überlebenden Person ausgeben.

    Demonstriere übersichtlich und sauber, dass das Programm korrekt arbeitet.

6. Übungsblatt

Abgabe der Beispiele bis 03.05.13 8:00 per `git`. Thema dieses Übungsblatts: Stacks, Queues, Heaps.

25. (2P) (Anzukreuzen, aber nicht abzugeben, jedoch zu präsentieren.)
    (a) Demonstriere manuell an einem arithmetischen Ausdrücken nicht zu geringer Komplexität die Algorithmen der VO (Postfixausdruck berechnen, Umrechnen von Infix zu Postfix, Syntaxbaum erstellen) und schreibe in jedem Schritt den Status des Stacks sowie die aktuelle Ausgabe auf.
    (b) Demonstriere manuell an einer Zahlenfolge nicht zu geringer Komplexität das Aufbauen eines Max-Heaps und stelle ihn in jedem Schritt graphisch als Baum dar. Lösche jeweils dreimal das größte Element wieder heraus, und führe die korrekten Umbauoperationen am Baum durch. Demonstriere an einem Beispiel den Zusammenhang zwischen Speicherung als Binärbaum und als Array.
26. (3P) Studiere und vervollständige die Implementierung von `upheap(...)`, `parentIdx(...)`, `leftChildIdx(...)`, `rightChildIdx(...)` eines Max-Heaps auf Basis eines Arrays (`maxheap.h`, siehe Moodle).

    Demonstriere das korrekte Funktionieren anhand einiger Beispiele (zB der vorigen Nummer) übersichtlich in `main.cpp`.
27. (2P) Implementiere eine Klasse `MyStack`, welche ein Array (`array.h`, siehe Moodle) einer zur compile-Zeit fixen Größe `capacity` kapselt (`capacity` soll beim Konstruktor mitgegeben werden). Die Funktionen `pop`, `push` und `top` müssen die Fälle „leerer" bzw. „voller" Stack berücksichtigen.
28. (2P) Benutze den selbstgeschriebenen Stack, ob zu erkennen, ob beliebige Ausdrücke korrekt geklammert sind. Es müssen vier unterschiedliche Klammertypen (rund, geschweift, eckig, spitz) erkannt werden. Die Position des Eingabestrings, an welcher ein etwaiger Fehler auftaucht, muss ausgegeben werden, sowie welche Klammer erwartet wurde. (Was innerhalb der Klammern steht ist für diesen Zweck unerheblich und muss nicht analysiert werden.)

    Bsp.: `{xyz}abc+-x[[[a[cb]k-lm]v](a<s<df>>)]` ist korrekt geklammert.
    `(([(x+(x-({x+y})))]oo)poi)` ist nicht korrekt geklammert, Fehler an Position 14, gefunden: `)`, erwartet: }.
29. (2P) Implementiere einen Konverter von vollständig geklammerten Postfix- zu Infixausdrücken auf Basis des selbstgeschriebenen Stacks.
    (a) Der Konverter bekommt als Input einen String des Postfixausdrucks (etwa `34+5*62:-`) und liefert einen String des vollständig geklammerten Infixausdrucks, etwa `(((3+4)*5)-(6:2))`.

        Es genügt die korrekte Behandlung von einziffrigen Zahlen.
    (b) Vorgehensweise: beim Parsen des Postfixausdrucks werden die Operanden am Stack gespeichert, bis ein Operator auftaucht. Die Operanden werden vom Stack genommen, der zugehörige Infixausdruck gebaut, und als String (inkl. Klammern) wieder am Stack abgelegt.
    (c) Teste ausgiebig die korrekte Funktionalität in `main.cpp`.
30. **Bonus:** (1P) (Nicht abzuschicken, jedoch zu präsentieren.) Studiere das Konzept und die Implementierung einer Prioritätswarteschlange, welche eine gegebene doppelt verkettete Liste kapselt (`priorityqueue.h`, siehe Moodle), siehe auch VO S.15.
31. **Bonus:** (2P) Der Konverter akzeptiert auch mehrstellige Zahlen. Als Input erhält er nicht mehr einen einzelnen String, sondern eine *Liste* von sogenannten *Tokens* des Postfixausdrucks: ein Token ist dabei einfach ein String, welcher entweder eine mehrstellige Zahl enthält, oder einen Operator.
32. **Bonus:** (1P) Ein `Tokenizer` nimmt einen Postfixausdruck, welcher Leerzeichen als Trennzeichen enthält, als Input, und liefert die oben erwähnte Liste von Tokens.
33. **Bonus:** (2P) Entwickle und implementiere einen Algorithmus, der einen Infix-Ausdruck (etwa `(((3+4)*(5+6))+8)`) in **Prefix**-Schreibweise ausgibt:

    hierbei wird der Operator *vor* die Operanden geschrieben (polnische Notation). Aus `(A+B)` wird also `+ A B`. Das angegebene Beispiel ergibt etwa `+*+34+568`.

Verwende anschließend das Programm, um einen Infix-Ausdruck als „Baum“ auszugeben, etwa in der Form

```
    +
      *
        +
          3
          4
        +
          5
          6
      8
```

34. **Bonus:** (2P) Vergleiche die Performance zweier Implementierungen einer Priority Queue: einmal auf Basis einer doppelt verketteten Liste , einmal auf Basis des arraybasierten Max-Heaps (siehe oben).

    Miss die Performance von (1) `enqueue` von einer großen Menge zufällig generierten Werten, und (2) anschließendem `dequeue` aller Elemente.

    Nutze zB die Funktionen `clock` in `<ctime>`.

    Ein Liniendiagramm (mit Calc, Excel, gnuplot, R, ...), welche die Performance-Entwicklung beider Implementiertungen bei stetigem Erhöhen der Datenmenge (zB in 1 000er oder 10 000 Schritten) graphisch darstellt, ist zu erstellen. Beachte, dass für aussagekräftige Messergebnisse die Datenmengen so gewählt werden sollen, dass die gemessenen Zeiten zB zwischen 1 s und 1 min variieren.

35. **Bonus**: (Nicht anzukreuzen, nicht abzuschicken.) Recherchiere, wie Priority Queues für das Gebiet der *Diskreten Simulation* bzw. *Diskreten Ereignissimulation* eingesetzt werden. Demonstriere das Ablaufen einer solchen Simulation manuell anhand eines einfachen Beispiels, indem bei jedem Schritt der Zustand der Priority Queue aufgezeichnet wird.

7. Übungsblatt

Abgabe der Beispiele bis 10.05.13 8:00 per `git`. Thema dieses Übungsblatts: *Rekursion* und *Binärbäume*.

Für alle Programme gilt: teste und demonstriere die Korrektheit gründlich!

36. Übungsaufgaben die nicht committed werden müssen, aber separat zu kreuzen und an der Tafel zu präsentieren sind:
    (a) (1P) Berechne manuell die `n`-te Fibonacci-Zahl rekursiv (siehe VO S.26, `fib_rec`), indem du aufzeichnest, welche Funktion von welcher Funktion rekursiv aufgerufen wird (die aufrufende Funktion ist mit der aufgerufenen Funktion durch einen Pfeil verbunden). Der so entstehende Binärbaum der Funktionsaufrufe soll Schritt für Schritt aufgebaut werden. Trage ebenso die jeweiligen Rückgabewerte ein.
    (b) (1P) Zeichne den binären Suchbaum von `int`s, der entsteht, wenn die Zahlen 34, 13, 56, 29, 12, -3, -11, 16, 20, 44, 39, 49, 70 der Reihe nach eingefügt werden.
    (c) (1P) Traversiere den entstandenen Baum in pre-/in- und postorder.
    (d) (1P) Wie sieht der Baum jeweils aus, nachdem der Reihe nach die Elemente 13, 44, 56 gelöscht werden? (Die Methode der VO S.13-16 verwenden.)
    (e) (1P) In welcher Reihenfolge müssen die Zahlen von 1 bis 15 in einen Baum eingefügt werden, sodass ein vollständiger binärer Suchbaum entsteht? (Es gibt mehrere Möglichkeiten.)
37. Schreibe ein Programm mit folgenden rekursiven Funktionen (jede Unternummer kann separat gekreuzt werden):
    (a) (1P) eine Funktion `void countdown(unsigned int n)`, die rekursiv (also ohne Schleife!!!) einen Countdown von n ausgibt, für $n = 4$ also: `4, 3, 2, 1, 0`
    (b) (1P) eine Funktion `unsigned int sum(unsigned int n)`, welche rekursiv die Summe der Zahlen von 1 bis n zurückliefert. `sum(3)` liefert also `6`, `sum(100)` 5050. (Die Summenformel von Gauss kann zur Verifizierung des Ergebnisses verwendet werden.)
    (c) (1P) eine Funktion `float power(float x, unsigned int n)`, die rekursiv Potenzrechnen kann, nach dem Schema:
        - $x^0 = 1$, und
        - $x^n = x \cdot x^{n-1}$ für $n > 0$

        `power(2.0, 3)` liefert also `8.0`, `power(-3.0, 3)` liefert `-27.0`, und `power(3.1415, 0)` liefert `1.0`.
    (d) (1P) eine Funktion `isPalindrome(string s)`: sie überprüft rekursiv, ob ein String ein Palindrom ist: also ein String, der von hinten nach vorne gelesen gleich ist wie von vorne nach hinten, zB `Radar`. Groß-/Kleinschreibung soll ignoriert werden.

        Hinweis: Ein String aus einem Buchstaben ist immer ein Palindrom. Ein String aus mehreren Buchstaben ist ein Palindrom, wenn erster und letzter Buchstabe gleich sind, und gleichzeitig der Zwischenstring ein Palindrom ist.
    (e) **Bonus:** (2P) schreibe `isPalindromeSentence`, welche auch Palindromsätze erkennt, zB `O Genie, der Herr ehre Dein Ego!` Satzzeichen, Leerzeichen und Groß-/Kleinschreibung sollen ignoriert werden.
    (f) **Bonus:** (2P) Recherchiere das Problem der *Türme von Hanoi* und implementiere den rekursiven Lösungs-Algorithmus von Randoff[17]. Versuche im Detail die Abfolge der Züge nachzuvollziehen[18], die durch den rekursiven Algorithmus geliefert werden.

[17] `http://de.wikipedia.org/wiki/T%C3%BCrme_von_Hanoi`
[18] `http://www.mathematik.ch/spiele/hanoi_mit_grafik/`

38. Erweitere die `Bintree`-Klasse (Code siehe Moodle) um folgende Funktionen (dürfen wieder separat gekreuzt werden):

    (a) (1P) `doesContainRec(Node* root, int value)`, welche statt einer Schleife wie in der VO nun rekursiv ermittelt, ob `value` im Baum mit Wurzel `root` enthalten ist.

    Ein (Teil)Baum enthält einen Knoten mit dem Wert `value`, wenn die Wurzel oder der linke/rechte Teilbaum einen Knoten mit `value` enthält. (Achtung: nicht bei jedem Aufruf beide Teilbäume, sondern *nur einen* durchsuchen.)

    (b) (1P) `count(Node* root)`, welche rekursiv alle Knoten eines Binärbaums zählt.

    Hinweis: `count(n)` ist die Summe von `count(n->left)` und `count(n->right)`, plus 1. Überlege weiters, was `count(n)` liefert, wenn `n` ein Null-Pointer ist.

    (c) (1P) `T sum(Node* root)`, welche rekursiv die Werte aller Knoten eines Binärbaums addiert und die Summe zurückgibt.

    (d) (1P) `height(Node* n)`, welche rekursiv die Höhe des Binärbaums mit Wurzel `n` bestimmt.

    Hinweis: die Höhe ist jeweils das Maximum der Höhen der beiden Unterbäume plus 1 (sofern einen nicht-leeren Unterbaum gibt). Die Höhe eines Baumes, dessen Wurzel keine Kinder besitzt, ist 0.

    (e) (1P) `isEqual(const Bintree<Key>& t)`, welche zwei Bäume auf Gleichheit prüft.

    (Zwei Bäume sind gleich, wenn alle Knoten den gleichen Wert aufweisen, *und* die Bäume dieselbe Struktur aufweisen.)

    (f) (1P) `getDepth(Node* n)`, welche rekursiv die Tiefe eines Knotens zurückliefert.

    Bsp.: VO S.7: Knoten 8 hat Tiefe 0, Knoten 10 hat Tiefe 2, Knoten 17 Tiefe 3.

    (g) (1P) `print_postorder(Node* root)`: Implementiere die Postorder-Traversierung[19] rekursiv.

    Das Prinzip ist analog zur Preorder-Traversierung (VO S.29): gib die Knoten des linken Teilbaums postorder aus, dann die des rechten Teilbaums; gib danach den Wert des aktuellen Knotens aus.

    Bsp: die Ausgabe der Postorder-Traversierung des Baums (VO S.27) wäre: 2, 6, 4, 15, 22, 19, 18, 14.

    (h) **Bonus:** (2P) `getDistance(Node* n1, Node* n2)`, welche den *Abstand* von zwei Knoten eines binären Suchbaums zurückliefert. Der Abstand ist definiert als die Länge des (eindeutigen) Kantenzuges von `n1` zu `n2`, wobei die Kantenrichtungen *nicht* berücksichtigt werden.

    ZB: der Abstand von Knoten 6 und 10 der VO-Slides auf S.7 wäre 4, der von Knoten 10 und 17 ist 3, der von Knoten 8 und 16 ist 2, der von Knoten 16 und 17 ist 1.

    (Hinweis: `getDistance(root, n);` muss die Tiefe von `n` liefern; `getDepth` darf verwendet werden.)

    (i) **Bonus:** (2P) `mirror(Node* root)`, welche rekursiv den binären Suchbaum spiegelt, also alle linken und rechten Unterbäume so vertauscht, dass eine Inorder-Traversierung eine *ab*steigende Sortierung liefern würde.

    (Ein Knoten `n` der vorher ein linkes Kind des Eltern-Knotens war, ist in der gespiegelten Version sein rechtes Kind.)

    (j) **Bonus:** (2P) Implementiere die Inorder-Traversierung *ohne* Rekursion.

    Hinweis: solange es ein linkes Kind des aktuellen Knotens gibt, kommt es auf den Stack. Wenn es kein linkes Kind mehr gibt, wird der aktuelle Knoten vom Stack genommen, ausgegeben, und sein rechtes Kind kommt aus den Stack usw.

    (k) **Bonus:** (2P) Implementiere die Präordertraversierung *ohne* Rekursion, dafür mithilfe eines Stacks. Hinweis: die Wurzel muss zuerst auf den Stack. Danach in einer Schleife: das oberste Element vom Stack nehmen, dessen Kinder in der richtigen Reihenfolge wieder auf den Stack geben, und innerhalb der Schleife das vom Stack genommene Element ausgeben. (Bis der Stack leer ist.)

    (l) **Bonus:** (2P) weise an einem implementierten Beispiel nach, dass unter Verwendung einer Queue anstatt eines Stacks eine Level-Order-Traversierung entsteht.

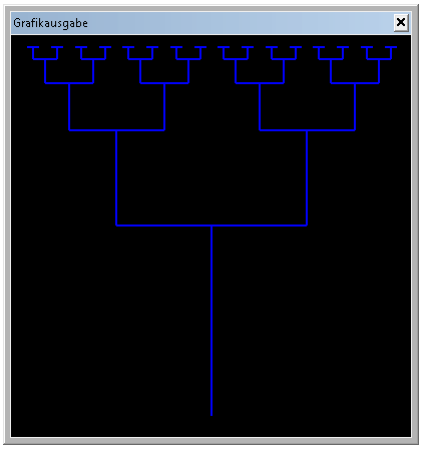
[19] Ein Demo-Applet ist unter `http://nova.umuc.edu/~jarc/idsv/lesson1.html` zu finden. Achtung: evtl. die Tilde manuell eingeben.

8. Übungsblatt

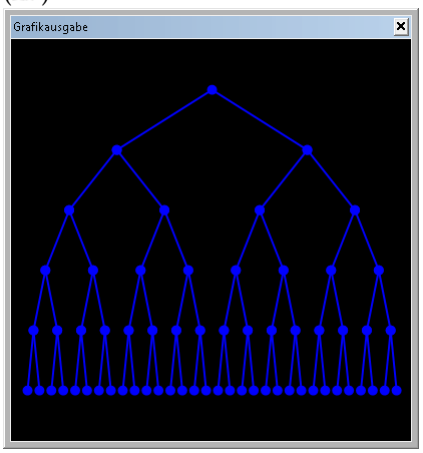
Abgabe der Beispiele bis 24.05.13 8:00 per `git`. Thema dieses Übungsblatts: *Rekursion* und *fraktale Geometrie*.

39. Programmiere *rekursiv* folgende geometrische Muster mit einer Technologie deiner Wahl (C++ mit einer Grafik-Library, Visual Studio und WinForms, HTML5/javascript/canvas,...). Bitte die Laptops zur Präsentation vorbereiten. (Es ist je nach Technologie nicht unbedingt erforderlich, dass die Programme im Computer-Labor kompilieren.)

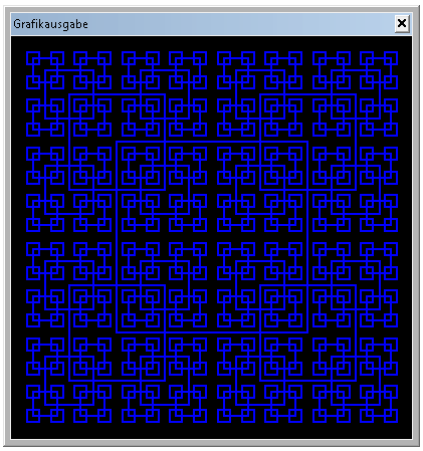
    (a) (2P) Hinweis: Zeichne zuerst das große T mit einem gewissen Startpunkt und Größe. Rufe danach rekursiv die Funktion zum Zeichnen der beiden nächstkleineren Ts auf, an den jeweils richtigen Startpunkten (Enden des Querbalkens), mit der richtigen Größe.



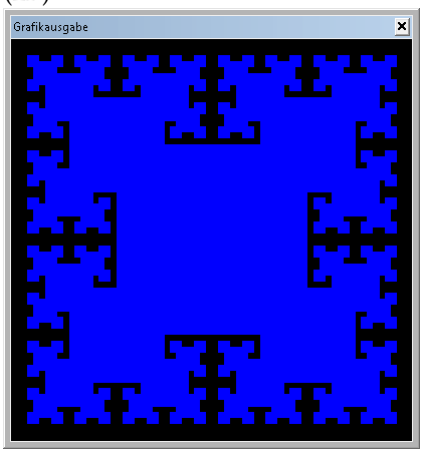
    (b) (2P)



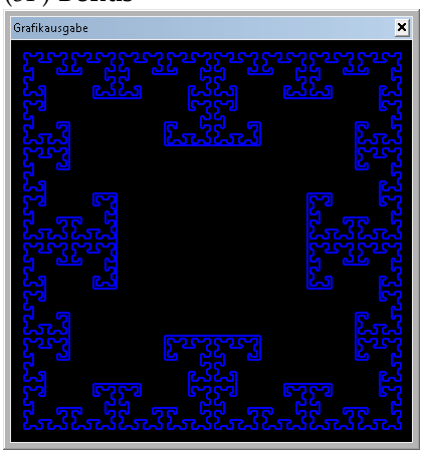
    (c) (2P)



    (d) (2P)



    (e) (5P) **Bonus**



9. Übungsblatt

Abgabe der Beispiele bis 07.06.13 8:00 per `git`. Thema dieses Übungsblatts: *Laufzeitanalyse* und *Komplexität*.

40. (1P) Wir untersuchen Algorithmen mit Laufzeiten von $\log n$, $\sqrt{n}$, n, n^2, 2^n. (Nicht abzuschicken.)
    (a) Analysiere jeweils mathematisch, wie sich die Laufzeit verändert, wenn n verdoppelt wird.
    (b) Analysiere jeweils mathematisch, wie n verändert werden muss, damit sich die Laufzeit verdoppelt.
41. (1P) Vergleiche die Laufzeitentwicklungen von $11n$, $n \log n$, und $n^2/1000$ mit einem gemeinsamen Plot (zB mithilfe einer Tabellenkalkulation) aller drei Funktionen von 0 bis n_{max} (in beliebigen, aber gleichmäßigen Schritten[20]).

    Wähle für den ersten Plot $n_{max} = 1.000$, für den zweiten $n_{max} = 10.000$, und für den dritten $n_{max} = 100.000$. Vergleiche und interpretiere! Verhält sich $n \log n$ eher linear oder eher quadratisch?

    (Plots bzw. Tabellen-Sheets abschicken.)
42. (1P) Implementiere die Binäre Suche (VO Slides S.22) **rekursiv**. Hinweis: Verwende als Parameter `int* number, int idxLeft, int idxRight, int k`, wobei `idxLeft` die aktuelle linke Grenze des zu durchsuchenden (Teil-)Arrays ist, und `idxRight` die rechte Grenze.
43. (4P) Wir weisen nach, dass die **Laufzeit der Binärsuche**[21] in $\Theta(\log n)$ liegt.
    (a) Die mathematische Begründung liegt darin, dass im jeden Schritt halbiert wird. Warum ist es trotzdem gültig, zu sagen, die Laufzeit liegt in $\Theta(\log n)$, wo doch aufgrund der Halbierung $\log_2 n$ verwendet werden müsste?
    (b) Weise die Eigenschaft experimentell nach, indem du sie mit der linearen Laufzeit der sequentiellen Suche (Durchsuchen von vorne nach hinten) vergleichst:
        i. Lass n gleichmäßig anwachsen (zB in 100er oder 1.000er Schritten; zB von 1.000 bis 30.000)[22].
        ii. Für jedes n, erzeuge ein **sortiertes** Array der Größe n.
        iii. Für jedes n, lass im Array nach einem zufälligen, aber sicher enthaltenen Element suchen. (Wie geht dies einfach? Einen Index zufällig[23], wählen, dann nach dem Element an diesem Index binär suchen lassen.)
        iv. Zähle die notwendigen **Arrayzugriffe**, bis das Element gefunden wurde. (Einmal für die Binärsuche, einmal für die sequentielle Suche.)
        v. Um den Zufall herauszumitteln, führe für jedes n eine gewisse fixe Anzahl m von Versuchen durch, und bilde den Mittelwert (jeweils für Binärsuche und sequentielle Suche).
        vi. Trage die Mittelwerte der Arrayzugriffe der beiden Suchverfahren für jedes n **übersichtlich** in eine Tabelle [24] ein.
        vii. Erstelle daraus (zB mithilfe einer Tabellenkalkulation) für beide Suchverfahren einen **übersichtlichen Plot**. Er muss eine schöne und **klar erkennbare logarithmische** (im Fall der Binärsuche) bzw. **lineare** (im Fall der sequentielle Suche) Laufzeitentwicklung bei steigendem n zeigen.

[20] Das heißt, eine gewöhnliche lineare Skala, nicht eine logarithmische wie in der VO verwenden.
[21] Unabhängig, ob die Implementierung rekursiv oder iterativ ist.
[22] Größere Arrays sind zwar sinnvoll, aber bei kleinem `RAND_MAX` etwas schwieriger zu behandeln. Siehe unten.
[23] Achtung: die jeweilige Größe von `RAND_MAX` beachten!!! Sind Arraygrößen, die `RAND_MAX` übersteigen, für den Plot nötig, ggf. eine Alternative finden, etwa die Funktion `random_shuffle` aus `<algorithm>`. (Aufruf: `random_shuffle(array, array+SIZE);` für ein Array der Größe `SIZE`.)
[24] Man kann zur Erleichterung der Ergebnisverwaltung die Ausgabe des Programms einfach mit dem >-Operator in eine Textdatei (zB CSV-Format) umleiten. (Bsp. für einen derartigen Konsolenaufruf: `myprog.exe > output.csv`)

44. (4P) Was ist die Laufzeitkomplexität für das Aufbauen eines **binären Suchbaums** mit n Knoten im **average** case und im **worst** case?
    (a) Arbeite mit der bereits bekannten `bintree.h`-Implementierung.
    (b) Lasse n wiederum gleichmäßig ansteigen.
    (c) Für jedes n, baue einen Binärbaum mit n Knoten auf. (Beginne mit einem leeren Binärbaum.) Für den Fall des average case, wähle die Werte **zufällig**. Für den Fall des worst case, wähle eine **sortierte** Reihenfolge.
    (d) Miss in beiden Fällen die **Zeit** in ms zum Aufbauen des gesamten Baums mit der Funktion `clock()` aus `<ctime>`.
    (e) Erstelle wie oben **übersichtliche Tabellen und Plots** mit den Laufzeiten für jedes n.
    (f) Aufgrund der Laufzeitplots, gib eine Vermutung für die Laufzeitkomplexitäten ab, und begründe sie auch mit mathematischen Argumenten.
    (g) Beachte, dass für aussagekräftige Messergebnisse die Datenmengen n so gewählt werden sollen, dass die gemessenen Zeiten zB zwischen 1 s und 1 min. variieren. (Sehr kleine gemessene Zeiten sind nicht aussagekräftig.)
45. (nicht anzukreuzen) Studiere `http://blog.codility.com/2011/03/solutions-for-task-equi.html`.
46. **Bonus:** (4P) Löse die Omega 2013 Challenge unter `http://codility.com/train/`

10. Übungsblatt

Abgabe der Beispiele bis 14.06.13 8:00 per `git`. Thema dieses Übungsblatts: *Sortieren* mit *Laufzeitanalyse* und *Komplexität*.

47. (4P) Führe zur Übung Selection Sort, Insertion Sort, Quick Sort und Merge Sort Schritt für Schritt manuell am Papier auf Arrays selbst gewählter Größe aus.

    Am Beginn ist es ratsam, das Array nicht zu groß zu wählen, dafür nach jeder Veränderung der Reihenfolge neu aufzuschreiben. (ist nicht zu *git*-ten)

48. (6P) **Optimiere Quick Sort** und führe **Laufzeitmessungen** durch.

    (a) Erweitere die Quick-Sort-Version der VL, indem...

    - ...das Pivot-Element als den *Median* des ersten, mittleren und letzten Elements des (Teil-)Arrays gewählt wird, *und*
    - *Insertion Sort* verwendet wird, wenn das (Teil-)Array eine Größe von höchstens 32 besitzt.

    (b) Vergleiche die **originale** Version A von Quicksort und die **optimierte** Version B anhand aussagekräftiger Laufzeittests [25].

    (c) Verwende für die Tests sowohl bereits **sortierte** Arrays (produziert den worst-case bei A) sowie **randomisierte** Arrays. Es darf mit der STL-`vector`-Klasse gearbeitet werden!

    (d) Wie am letzten Übungsblatt gilt: um aus einem Array der Zahlen 1 bis n ein zufällig gewürfeltes Array zu erzeugen bietet sich die Funktion `random_shuffle` aus `<algorithm>` an [26].

    (e) Lasse n schrittweise bis zu einer sinnvollen Grenze ansteigen, sodass die **Entwicklung der Laufzeit klar ersichtlich** wird [27].

    (f) Lasse am besten wiederum per Umleitungsoperator > die Ergebnisse in eine csv-Datei schreiben, welche mit einem Tabellenkalkulationsprogramm für den **Plot** weiterverarbeitet werden. Analysiere die Ergebnisse hinsichtlich Komplexität.

    (g) Achtung: damit die hohe Rekursionstiefe im worst-case (unoptimierter Quicksort bei sortiertem Array) keinen vorzeitigen Programmabbruch herbeiführt, das Programm zusätzlich mit den Kommandozeilenoptionen `-O3 -funroll-loops` kompilieren.

49. (2P) Bubblesort:

```
1   void bubbleSort(int arr[], unsigned n) {
2       bool swapped;
3       do {
4           swapped = false;
5           for (i=0; i< n-1; i++)
6           if (arr[i] > arr[i+1]) {
7               swap(arr[i], arr[i+1]);
8               swapped = true;
9            }
10           n = n-1;
11      } while (swapped);
12   }
```

    (a) Führe den Algorithmus manuell auf einem beliebigen `int`-Array durch, und überlege, warum der Algorithmus korrekt ist.

    (b) Finde durch *mathematische* Analyse die Anzahl der Vergleiche und Vertauschungen im best und worst case heraus.

    (Aufgabe nicht *git*-ten.)

[25] Verwende zur Zeitmessung wieder die Funktion `clock()` bzw. die Konstante `CLOCKS_PER_SEC` aus `<ctime>`.

[26] Sie wird bei einem STL-vector `v` wie folgt aufgerufen: `random_shuffle(v.begin(), v.end());`. Den Zufallszahlengenerator zuvor mit `srand(time(0))` initialisieren.

[27] Siehe die Demo-Plots in Moodle. Im worst-case des Algorithmus A können nur viel kleinere n behandelt werden, darum der separate Plot.)

50. (2P) Bonus: Erweitere Selection Sort und Insertion Sort.
    (a) Sowohl bei Selection Sort als auch Insertion Sort (beide Varianten) soll die Zahl C der Vergleiche und S Vertauschungen (bzw. Verschiebungen) mitprotokolliert werden.
    (b) Führe die Algorithmen für Arrays mit zB 10^3, 10^4 und 10^5 (sofern möglich) Zufallswerten durch und vergleiche mit den Formeln für C und S aus VL10 (S.20 bzw. S.23).
51. (2P) Bonus: Vergleiche den optimierten Quicksort-Algorithmus (B) anhand von Laufzeitmessungen mit dem vordefinierten Algorithmus `sort` der STL. Demonstriere mit Plots: Welche Variante ist schneller? Gibt es Komplexitätsunterschiede? Wie ist es bei sortierten, wie bei randomisierten Daten?

11. Übungsblatt

Abgabe der Beispiele bis 21.06.13 8:00 per `git`. Thema dieses Übungsblatts: *Hashtables* sowie freiwillig *Problemlösung mit rekursivem Backtracking*. Im gezippten Material auf Moodle finden sich Implementierungen von Hashtables:

- `hashtablesc.*` — verwendet separate chaining
- `hashtablelp.*` — verwendet open addressing mit linear probing.
- `hashtabledh.*` — verwendet open addressing mit double hashing.

52. (10P) Verständnisaufgaben[28]:
    (a) Wende die Divisionsmethode und linear probing an, um in eine Hashtable der Größe 12 nacheinander die key-value-Paare (7, A), (14, B), (22, C), (58, D), (71, A), (122, C), (238, D), (602, A), (1202, B) einzufügen. Wie sieht die Hashtable am Ende aus?
    (b) Erkläre die Löschproblematik bei linear probing bei naivem Rücksetzen des `occupied`-Arrays (*ohne* nachträglichem Neu-Einfügen, siehe VL12 S.11-12) am Beispiel des Löschens des Elements mit key 22, und dem anschließenden Suchen von key 238. Was passiert dabei?

        Lösche nun 22 mit der vorgeschlagenen Variante, die Elemente des Clusters *neu* einzufügen. Wie sieht die Hashtable nun aus?
    (c) Eine Hashfunktion für keys, welche als String der Länge L vorliegen, sei wie folgt gegeben: $h(k) = (\sum_{i=0}^{L-1} k[i]) \mod M$.

        Füge in eine Hashtable der Länge 10 die Strings `abc`, `bbb`, `cba`, `aad` ein (linear probing). Was ist der offensichtliche Nachteil dieser Hashfunktion?
    (d) Wende die Multiplikationsmethode für A = 0.61 und M = 13 an, und füge Elemente mit den keys 0.01, 0.31415, 4.44, 0.707, 13.19 und 21.12 ein.
    (e) Verifiziere die mod-Formel auf S.24 (VL11) anhand verschiedener Werte für a und b.
    (f) Wende die Hashfunktion auf S.11 für den key `hallo` an, und verwende dabei die angegebene mod-Formel, welche das Entstehen größerer Zwischenergebnisse verhindert.

        Welcher Hashwert ergibt sich?

        Wende nun eine geringfügig veränderte Hashfunktion auf `hallo` an, indem nun die Basis 256 mit 128 ersetzt wird.
    (g) Führe das Beispiel auf S.15 (VL12) zu double hashing durch, und verifiziere die behauptete probing sequence.
    (h) Führe das Beispiel auf S.15 (VL12) durch, diesmal für M=17 anstatt 16.

        Wie lautet die Probing Sequenz nun?
    (i) Warum darf $h_2(k)$ nicht 0 sein? Warum ist $h_2(k) = 1$ nicht sinnvoll? Warum darf nicht $h_2(k) = M$ sein?
    (j) Wende double hashing statt linear probing mit der zweiten Hashfunktion $h_2(k) = 1 + (k \mod (M - 2))$ für das Beispiel in (a) an.

[28] Nicht abzugeben, aber einzeln anzukreuzen, manuell durchzuführen und an der Tafel zu präsentieren. Jede Aufgabe ist 1P wert.

53. (4P) **Statistiken zu Hashtables:**

    (a) *Statistiken über separate chaining.*

    Erzeuge eine Hashtable der Größe 1.000 und befülle sie mit 10.000 zufälligen key-value-Paaren.

    Was sind die maximale und die durchschnittliche Länge der verketteten Listen der Hashtable?

    (b) *Statistiken über linear probing und double hashing.*

    Erzeuge jeweils eine Hashtable der Größe 10.000 und befülle sie mit jeweils n zufälligen key-value-Paaren, wobei n von 5.000 bis 9.500 in 500er Schritten ansteigt.

    Was sind die maximale und die durchschnittliche Länge der Cluster in der Hashtable für die verschiedenen Anzahlen?

    Speichere die Ergebnisse gut illustriert tabellarisch ab und erzeuge jeweils einen **Plot** über die Entwicklung der Clusterlängen.

    - Achte darauf, dass ein einzelner Cluster das Ende und den Anfang einer Hashtable umfassen kann.
    - Tipp: suche den ersten freien Bucket und fange von dort an, die Cluster-Längen zu erfassen.

54. (4P) Bonus: *Resize a hashtable.*

    Führe bei allen drei Implementierungen (separate chaining; open addressing: linear probing und double hashing) ein neues Attribut `count` ein, welches die Anzahl der gespeicherten Elemente in der Hashtable speichert. Adaptiere die Methoden `insert` und `remove` entsprechend.

    Ergänze die open addressing Implementierungen um eine Methode `resize(unsigned tablesize)`, welche die Größe der Hashtable entsprechend ändert.

    Achtung, `resize` sollte überprüfen, ob die Grösss e der Hashtable ausreicht, um alle Elemente zu speichern (im Falle einer Verkleinerung).

    (Tipp: geschickt implementiert delegiert die Methode `resize` in beiden Fällen die wesentliche Arbeit an `insert`. Die Implementierung für beide Varianten unterscheidet sich daher kaum.)

    Adaptiere die Methoden `insert` und `remove` derart, dass die Hashtable ihre Größe verdoppelt, wenn der Befüllungsgrad über 50% steigt bzw. ihre Größe halbiert, wenn der Befüllungsgrad unter 12.5% fällt.

55. (4P) Bonus: Löse das 8-Damenproblem mithilfe *rekursiven Backtrackings*. Zähle alle Lösungen, und speichere jeweils die Konsolenausgabe der Lösungs-Spielbrettkonfiguration ab (zeichne ein `X` in die Felder, wo eine Dame steht).

    Beispielausgabe des Programmendes:

```
Found solution nr: 92
. . X . . . . .
. . . . . X . .
. . . X . . . .
. X . . . . . .
. . . . . . . X
. . . . X . . .
. . . . . . X .
X . . . . . . .

There are 92 solutions.
```

56. (4P) Bonus: Löse das Springerproblem (Springer startet bei Feld A1, der Springer muss nicht zu A1 zurückkehren) mithilfe *rekursiven Backtrackings*. Finde mindestens die ersten 20 Lösungen, und speichere jeweils den Konsolenoutput, indem du die Felder in der Reihenfolge des Besuchs des Springers durchnummerierst (Feld A1 besitzt immer die Nummer 1).

    Beispielausgabe:

```
Found solution nr: 1
1  38  59  36  43  48  57  52
60  35  2  49  58  51  44  47
39  32  37  42  3  46  53  56
```

```
34  61  40  27  50  55  4  45
31  10  33  62  41  26  23  54
18  63  28  11  24  21  14  5
9  30  19  16  7  12  25  22
64  17  8  29  20  15  6  13
```

57. (4P) Bonus: Löse das Münzwechselproblem. Gegeben ist eine vom Benutzer beliebig vorgegebene Anzahl an Münzen mit verschiedenen Werten. Auszugeben sind alle Möglichkeiten, wie ein vom Benutzer einzugebener Geldbetrag mit diesen Münzen dargestellt werden kann.

    Beispiel:

```
Input number of different coins: 3
You entered: 3.
Input coin 1: 1
Input coin 2: 5
Input coin 3: 10
Give amount of money: 21
You entered: 21.
Currently used coins: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Currently used coins: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 5
Currently used coins: 1 1 1 1 1 1 1 1 1 1 1 5 5
Currently used coins: 1 1 1 1 1 1 1 1 1 1 1 10
Currently used coins: 1 1 1 1 1 1 5 5 5
Currently used coins: 1 1 1 1 1 1 5 10
Currently used coins: 1 5 5 5 5
Currently used coins: 1 5 5 10
Currently used coins: 1 10 10
```

Trainingsmaterial

Zum Thema Graphen, sowie diverse Programmieraufgaben. ZB als Klausurvorbereitung, als Schlechtwetterprogramm, für schlaflose Nächte etc.

58. Führe die rekursive und nicht-rekursive Variante von DFS, sowie BSF auf einen zusammenhängenden, ungerichteten Graphen mit 10 Knoten und 15 Kanten aus.

    Wir gehen davon aus, dass adjazente Knoten immer in aufsteigend sortierter Reihenfolge besucht werden (und so zB in der Adjazenzliste gespeichert sind).

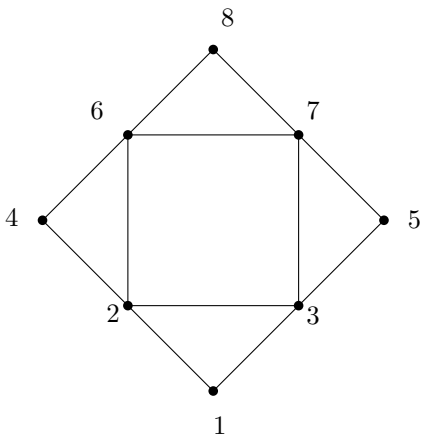
    Gib nach jeder Änderung den aktuellen Status der verwendeten Datenstruktur (zB `visit[]`, `todo`,...) in geeigneter Form an.

59. DFS (BFS) geben nur dann Knoten auf den Stack (in die Queue), sofern diese *noch nicht* besucht wurden (`if (!visit[curr])...`). Führe die beiden Algorithmen *ohne* diese `if`-Bedingung durch und analysiere das Ergebnis. Warum (bzw. warum nicht) ändert sich die Reihenfolge der besuchten Knoten? Vergleiche weiters die maximal erreichte Länge des Stacks/Queue mit der Variante von (a).

60. Gib die Reihenfolge der besuchten Knoten für den Graphen aus (a) an, mit dem Unterschied dass nun adjazente Knoten in *absteigend* sortierter Reihenfolge besucht werden.

61. Ein Student der Algorithmen und Datenstrukturen hat die Idee, bei DFS (BSF) nur dann Knoten auf den Stack (die Queue) zu geben, sofern diese noch nicht bereits am Stack (in der Queue) gespeichert sind. Funktionieren die Algorithmen immer noch? Ändert sich die Reihenfolge der besuchten Knoten? Ist es bei DFS und BFS unterschiedlich? Begründe und gib ein passendes Beispiel an.

62. Führe den Algorithmus zur Ermittlung einer Eulertour auf folgenden Graphen aus. Protokolliere den Status der verwendeten Datenstrukturen, insbesondere `todo` und `tour`. Knoten werden in aufsteigend sortierter Reihenfolge besucht.



63. Zeichne jeweils einen gerichteten und ungerichteten Graphen mit 7 Knoten und gib seine Ecken- und Kantenmenge formal korrekt an. Gib die Adjazenzmatrix und -liste(n) an.

64. Gib ein Beispiel für einen Teilgraphen, welcher keine Zusammenhangskomponente eines Graphen ist.

65. Slides S.7: kann bei einer Wanderung $k > n$ sein? Wie ist es beim Weg? Wie beim Pfad?

66. Gib formal korrekt Wege, Wanderungen und Pfade an. Finde eine Wanderung, die kein Weg ist. Finde einen Weg, der kein Pfad ist.

67. Zeichne einen DAG, der kein Wurzelbaum ist.

68. Zeichne einen Graph mit 5 Knoten, wo $|E|$ in $\Theta(|V|^2)$ ist.

69. *Spanning Trees*

    Implementiere den Algorithmus zur Berechnung des spannenden Baumes auf Basis von DFS.

    (a) Adaptiere den Code der VO (`dfs_visit_adjlist_nonrec`)

    (b) Die Adjazenzlisten sollen Knoten in aufsteigender Reihenfolge speichern.

(c) Die Funktion soll das Knoten-Array `vector<int> parents` liefern. Der Wert von `parents[i]` gibt den (Index des) Elternknoten(s) von Knoten `i` an. (Falls `tree[3]` also `5` ist, so ist Knoten 5 der Elternknoten von Knoten 3.) Für den Graphen auf Seite 25 (links) würde `parents` so aussehen: `[-1, -1, 1, 1, 3, 2, 2, 6, 7, 5, 9]`[29]
-1 bedeutet, dass es keinen parent-Knoten gibt.

(d) Beim Ablegen des zu besuchenden Knotens v am Stack ist es vorteilhaft, mitzuspeichern, von welchem Knoten aus man zu v gelangt.

70. **Weitere allgemeine Programmieraufgaben:** *Römische Zahlen in der Normalform*

Schreibe ein Programm, das eine natürliche Zahl zwischen 1 und 3999 in eine römische Zahl umwandelt, welche die Zeichen `I` (1), `V` (5), `X` (10), `L` (50), `C` (100), `D` (500) sowie `M` (1000) benutzt.

Die Subtraktionsmethode *in der Normalform* muss angewendet werden, um das Aufeinanderfolgen vier identischer Zeichen zu vermeiden (siehe wikipedia), so ist z.B. die Zahl 99 `XCIX` (aber *nicht* `IC`!!!), und 3999 ist `MMMCMXCIX`.

Die Ausgangszahl soll als einziges Argument dem Programm per command line argument (`argc`, `argv`) als Parameter übergeben werden, und die Ausgabe besteht ausschließlich aus der römischen Zahl.

71. *Permutationen*

Liste zu einem gegebenen String der Länge n (zB `abc` für n=3) alle $n!$ Permutationen auf. In diesem Beispiel: `abc, acb, bac, bca, cab, cba`.

72. *Teilmengen*

Liste zu einer gegebenen Menge M der Größe n (zB $n = 3$, $M = \{1, 2, 3\}$ alle 2^n Teilmengen auf. In diesem Beispiel: $\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$.

73. *Brute-Force Angriff auf ein Passwort*

Liste alle Möglichkeiten für ein Passwort der (Maximal-)Länge n auf, wobei das Passwort aus einer gewissen Menge von Zeichen (zB alphanumerisch) aufgebaut sein kann.

Kleine Variante: liste alle Möglichkeiten für das Passwort auf, wobei jedes Zeichen nur ein einziges Mal vorkommen darf.

74. *KI für Tic Tac Toe*

Verwende den *Minimax*-Algorithmus (evtl. auch: $\alpha\beta$-Suche), um eine optimale Computerstrategie für Tic Tac Toe zu finden.

Bei Tic Tac Toe können auch einfache Heuristiken zu einem guten Ergebnis führen: zB. folgende Regeln

(a) Wenn der Gegner im nächsten Zug gewinnen kann, verhindere seinen Sieg.

(b) Ansonsten: setze auf dasjenige Feld, das noch an der größtmöglichen Anzahl an potentiellen Gewinnsituationen teilnehmen kann. (Gibt es mehrere Felder mit der gleichen maximalen Anzahl an potentiellen Gewinnsituationen, so wähle einfach das erstbeste.)

Eine potentielle Gewinnsitution ist dabei eine Dreierreihe (Zeile, Spalte oder Diagonale), deren Felder entweder ausschließlich mit eigenen Steinen besetzt sind oder noch frei.

Finde weiters (manuell) eine Zugfolge, mit der man gegen diese Strategie gewinnen kann, wenn man den ersten Zug tätigen darf.

75. *Dreidimensionales Vier-Gewinnt*

Gegeben ist ein 4x4x4 Spielfeld/-würfel. Man muss sich dabei vier 4x4 Spielfelder übereinander gestapelt vorstellen. Zwei Spieler spielen abwechselnd, indem sie auf irgendeiner der vier Ebene ein X bzw. ein O setzen. Gewonnen hat derjenige Spieler, der als erster irgendeine Reihe voll hat, also 4 Steine in einer Spalte/Zeile/Diagonale, wobei jede mögliche Flächen- bzw. Raumdiagonale zählt. (Auch 4 Steine direkt übereinander gelten als eine volle Reihe). Ein Unentschieden muss ebenso erkannt werden. Bei Gewinn muss das Spiel beendet werden und der Gewinner ausgegeben werden.

Das Spiel muss jede Form einer ungültigen Eingabe abweisen, und eine übersichtliche Ausgabe nach jedem Zug tätigen, etwa:

[29] Hinweis: da wir in der VO bei Knoten 1 beginnen, darf das Array der Einfachkeit halber Größe $|V| + 1$ besitzen.

```
Ebene 1        Ebene 2         Ebene 3         Ebene 4
---------      ---------       ---------       ---------
|x|.|o|.|      |.|x|.|.|       |.|.|x|.|       |.|.|o|x|
|x|o|.|.|      |.|o|.|.|       |o|.|x|.|       |.|.|.|x|
|o|x|.|.|      |.|.|.|.|       |o|.|.|.|       |.|.|.|.|
|.|.|.|.|      |.|.|.|.|       |.|.|o|o|       |.|.|x|.|
---------      ----------      ---------       ---------
```

In diesem Fall hätte X gewonnen, weil es eine Diagonale besitzt (erste Zeile jeder Ebene).

76. *Project Euler*

Eine lange Liste von eigentlich mathematischen Problemen mit ansteigendem Schwierigkeitsgrad, die aber programmiertechnisch gelöt werden können, sind hier aufgelistet: `http://projecteuler.net` Die Lösung kann online verifiziert werden.

77. **Pirates of the Urstone.**

Captain *Jack Sparrow* und Konsorten sind gerade auf *Treasure Island* eingefallen, und wollen nun die verborgenen Schätze der Insel erbeuten.

Leider gibt es Hindernisse, die den Piraten ein paar Umwege bescheren. Glücklicherweise haben sie aber die geheime Schatzkarte vom fiesen *Captain Barbossa* erbeutet, auf der sowohl die Schätze als auch die Hindernisse eingezeichnet sind, sowie der einzige mögliche Anlegeplatz für die *Black Pearl*, von wo aus sie den Beutezug starten.

Hilf den Piraten nun, die Schätze möglichst schnell aufzusammeln, indem du mit Breitensuche die kürzesten hindernisfreien Wege vom Anlegeplatz zu den Schätzen findest, und die Schätze nach aufsteigender Entfernung vom Anlegeplatz aus sortiert ausgibst.

Details:

(a) Schatzkarten sind in Moodle verfügbar und sind als Textfile codiert. Eine Schatzkarte sieht beispielsweise wie folgt aus:

```
XXXXXXXXXX
X   S    X       X
X                 a   X
XXXXXXXXXXX   XXXXXXXX
        X
    b   X

XXXXXXXXXXXXXXXXX
```

Die Startposition ist durch das Zeichen 'S' gegeben. Alle Felder bis auf Hindernisse 'X' sind begehbar. Von einem Feld kann man in alle vier Himmelsrichtungen gehen, nicht jedoch auf Hindernisse oder über die Karten hinaus.

Ein Leerzeichen ist hindernisfreies Land, alle anderen Zeichen ('a', 'b', ...) zeigen einen Schatz an. Für das obige Beispiel soll das Programm also zuerst den Schatz 'a' und dann 'b' ausgeben, da von 'S' aus 'a' den geringeren Abstand hat.

(b) Der Graph muss nicht explizit abgespeichert werden, da durch die aktuelle Position und die Informationen in der Karte klar ist, welcher Knoten (Feld) mit welchem Knoten verbunden ist.

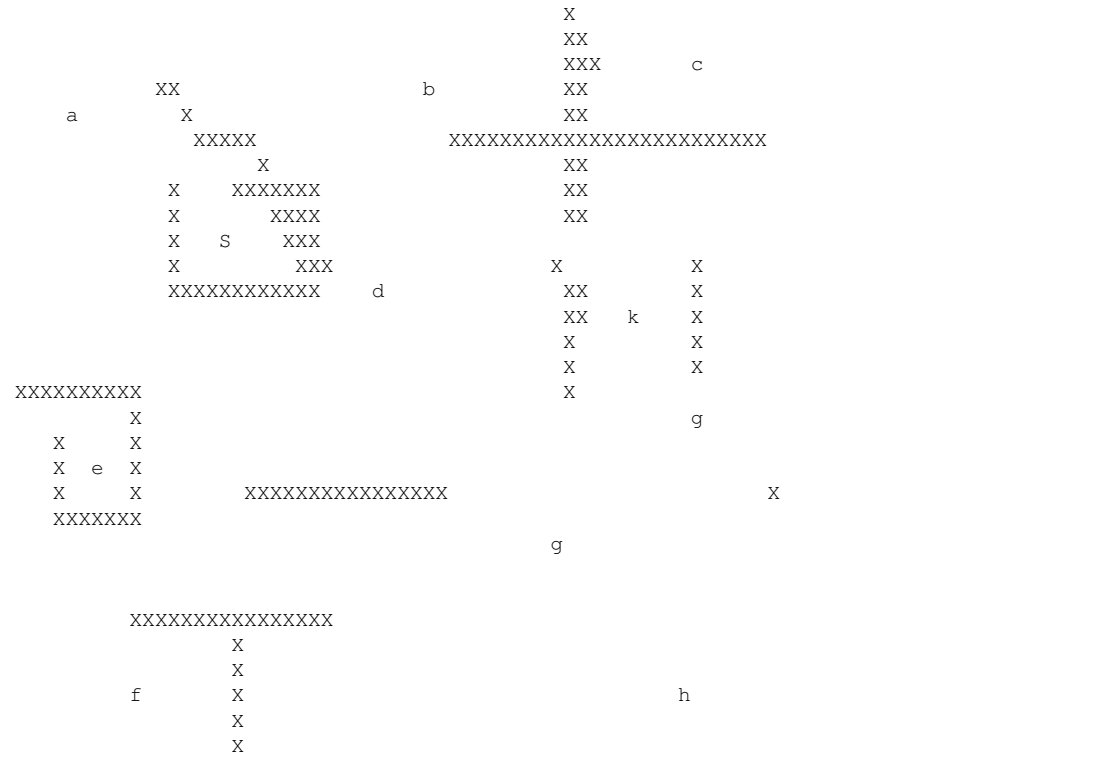
Es reicht daher die Karte zB als Matrix von `char vector<vector<char>>` abzuspeichern und für die Breitensuche bei einem Feld dessen vier Nachbarn zu betrachten.

(c) Verwende eine Hilfsmatrix `vector<vector<bool>> visited`, welche für jedes Feld abspeichert, ob jenes schon besucht wurde.

(d) Die Breitensuche liefert die Schätze in aufsteigender Distanz vom Startpunkt, unter Berücksichtigung der Hindernisse. Die Ausgabe des Programms sollen die Schätze in eben dieser Reihenfolge, inklusive ihrer Koordinaten, sein.

(e) Erleichtere den Piraten die Jagd nach den Schätzen, indem du zusätzlich zu jedem Schatz den Weg vom Startpunkt aus als Abfolge von Marschkommandos nach Norden `N`, Süden `S`, Westen `W` bzw. Osten `E` angibst. Ein möglicher Weg wäre zB für den Schatz `b` im Beispiel:
`SEEEEEEESSSSWWWWWWWN`.

Anbei eine Ausgabe für die Schatzkarte `map1.txt`.




```
start position: (16, 9)
Found: a @ (4, 4)
  path: WWWNNNWWWWWWWWNN
Found: d @ (28, 11)
  path: WWWNNNWWSSSSSSEEEEEEEEEEEEEEEN
Found: f @ (9, 27)
  path: WWWNNNWWWSSSSSSSSSSSSSSSWWSSSSESS
Found: b @ (32, 3)
  path: WWWNNNWWWNNNNEEEEEEEEEEEEEEEEEEES
Found: e @ (6, 18)
  path: WWWNNNWWWSSSSSSSSSSSSSSSWWWWWWWWNNNNNEEEESS
Found: g @ (42, 21)
  path: WWWNNNWWSSSSSSEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEESSSSSSSSS
Found: k @ (48, 12)
  path: WWWNNNWWSSSSSSEEEEEEEEEEEEEEEEEEEEEEEEEEEEEENNNEEEEEEESS
Found: g @ (53, 16)
  path: WWWNNNWWSSSSSSEEEEEEEEEEEEEEEEEEEEEEEEEEEEEESSSSEEEEEEEEEE
Found: h @ (52, 27)
  path: WWWNNNWWSSSSSSEEEEEEEEEEEEEEEEEEEEEEEEEEEEEESSSSEEEEEEEEESSSSSSSSSS
Found: c @ (53, 2)
  path: WWWNNNWWSSSSSSEEEEEEEEEEEEEEEEEEEEEEEEEEEEEENNNEEEEEEEEEEEEEEEEENNNNNWWWWWWNN

\end{enumerate}
```