Prüfung Algorithmen und Datenstrukturen I

Datum	:	23.01.2017, 08:30 Uhr
Bearbeitungszeit	:	90 Minuten
Prüfer	:	Prof. Dr. Oliver Braun
Hilfmittel	:	Keine
Erreichbare Punkte	:	90

Name: _____

Vorname: _____

Matrikelnummer: _____ Studiengruppe: _____

Hörsaal: _____ Platz Nr.: _____

Unterschrift: _____

Bitte kontrollieren Sie, ob Sie eine vollständige Angabe mit 5 Aufgaben auf 8 Seiten erhalten haben.

Aufgabe	1	2	3	4	5	Summe
max. Punkte	30	15	15	15	15	90

Anmerkungen:

- Nutzen Sie einen dokumentenechten Stift für alles was bewertet werden soll. Auch bei Skizzen ist die Verwendung eines **Bleistifts nicht** zulässig.
- Schreiben Sie die Lösungen in die dafür vorgesehenen Kästchen bzw. direkt zur Aufgabe. Sollte Ihnen der Platz dabei nicht reichen, benutzen Sie die Rückseite **und vermerken Sie das bei der entsprechenden Aufgabe!**

Aufgabe 1 (30 Punkte)

Eine Deque (Double Ended Queue) ist eine Datenstruktur die ähnlich einem Stack bzw. einer Queue ist. Der wesentliche Unterschied ist, dass an beiden Enden angefügt und entfernt werden kann.

Gegeben sei folgende C++-Klasse für eine Deque:

```
class Deque {
private:
    struct Elem {
        const int content;
        Elem *next = nullptr;
        Elem(const int c, Elem *n) : content(c), next(n) { }
    };
    Elem *front = nullptr, *back = nullptr;
    void balance();
public:
    bool empty() { return front == nullptr && back == nullptr; }
    // vorne anfügen
    void push(const int);
    // vorne entfernen
    int pop();
    // hinten anfügen
    void put(const int);
    // hinten entfernen
    int get();
    // vorne lesen ohne entfernen
    int first();
    // hinten lesen ohne entfernen
    int last();
    // wenn eine Liste leer ist, die andere aufteilen
    friend int length(Elem *);
    friend Elem *reverse(Elem *);
};
```

Die Besonderheit an der Deque-Implementierung ist Folgendes:

Eine Deque wird durch zwei einfach verkettete Listen dargestellt, wobei die zweite Liste umgedreht ist. Dadurch ist das Einfügen und Entfernen an beiden Enden sehr schnell, da die Liste nicht durchlaufen werden muss. Sobald auf eine der beiden Teillisten zugegriffen wird obwohl diese leer ist, muss die andere Liste wieder einigermaßen gleichmäßig aufgeteilt werden, damit anschließende Zugriffe wieder schnell sind. Das Aufteilen übernimmt die Methode `balance()`, die nur in diesem Fall von den entsprechenden Methoden aufgerufen wird.

Zur Verdeutlichung ein Beispiel (die beiden Listen werden hier im Beispiel als Tupel dargestellt):

```
auto deque = new Deque();
// ([], [])
```

```
deque->push(1)
// ([1], [])
deque->put(2)
// ([1], [2])
deque->push(3)
// ([3, 1], [2])
deque->put(4)
// ([3, 1], [4, 2]) entspricht der double ended queue 3->1->2->4
deque->pop() // 3
// ([1],[4,2])
deque->pop() // 1
// ([],[4,2])
deque->pop() // 2
// erst wird durch balance umgebaut in  ([2],[4]) und dann die 2 zurück gegeben
```

Die beiden Hilfsfunktionen `length()` und `reverse()` werden nur in `balance()` genutzt.

Wird auf eine leere Deque mit `pop()`, `get()`, `first()` oder `last()` zugegriffen, werfen Sie eine Exception, z.B. so:

```
if (empty())
    throw "cannot pop from empty deque";
```

(a) Geben Sie Implementierungen für `push` und `put` an: (6)

(b) Geben Sie eine Implementierung für `pop` an: (12)

(c) Geben Sie eine Implementierung für `reverse` an: (12)

Aufgabe 2 (15 Punkte)

Gegeben sei folgende Liste, die in einem Array abgelegt ist:

```
12, 3, 56, 18, 32, 19, 1, 21
```

Die Liste soll mit dem Standard-Quicksort-Verfahren (keine Variante) in-situ sortiert werden. Als Pivot-Element soll immer das letzte Element der zu sortierenden (Teil-)Liste genommen werden.

Geben Sie alle verschiedenen Speicherzustände, die jeweils vor bzw. am Ende eines Rekursionsschrittes vorliegen, an und machen Sie deutlich was beim Übergang von einem zum anderen Zustand passiert ist. Führen Sie von einer zur nächsten Zeile jeweils **alle** zu diesem Moment anstehenden rekursiven Aufrufe aus. Beispiel: Von Zustand 1 nach 2 werden die Teillisten berechnet und das Pivotelement wandert an die richtige Stelle. Von Zustand 2 nach 3 wird das für **beide** Teillisten parallel gemacht, …

Unterstreichen Sie die Teillisten, die Argumente eines rekursiven Aufrufs sind, und markieren Sie die Werte sobald sie ihre endgültige Position erreicht haben.

Aufgabe 3 (15 Punkte)

(a) Erstellen Sie einen Treap aus folgenden Werten, wobei die erste Komponente des Tupels jeweils der *Schlüssel* und die zweite Komponente die *Priorität* ist: (4,7), (8,6), (12,3), (13,12), (15,9), (17,11) (10)

(b) Erläutern Sie wie das Element (16,1) schrittweise in den Treap eingefügt wird? Geben sie außerdem den Treap nach dem abgeschlossenen Einfügen an. (5)

Aufgabe 4 (15 Punkte)

Gegeben sei folgende C++-Klasse für einen binären Suchbaum:

```
class BinTree {
private:
    int val;
    BinTree *left = nullptr, *right = nullptr;
public:
    bool search(const int) const;
    vector<int> *preorder() const;
};
```

(a) Geben Sie eine Implementierung für `search` an: (7)

(b) Geben Sie eine Implementierung für `preorder` (Hauptreihenfolge) an: (8)

Anmerkung: Sie können an einen Vektor `*v1` den Inhalt von `*v2` folgendermaßen anhängen:

```
v1->insert(v1->end(), v2->begin(), v2->end())
```

Aufgabe 5 (15 Punkte)

Gegeben sei der AVL-Baum der nur aus einer Wurzel mit dem Schlüssel 18 besteht.

Fügen Sie in den AVL-Baum nacheinander die Werte 12, 13, 22, 24 und 20 ein und geben Sie alle **Zwischenergebnisse** als AVL-Bäume inklusive der Balancefaktoren an, d.h. den AVL-Baum mit 18 und 12, den AVL-Baum mit 18, 12 und 13, …

Anmerkung: Sie brauchen keine Blätter zeichnen, es reichen die inneren Knoten.