Übungsblatt 5

Jan Hendrik Dithmar (Matrikelnummer 2031259)
Andreas Steinel (Matrikelnummer 2029984)
Manuel Gorius (Matrikelnummer 2030845)
Robert Vollmann (Matrikelnummer 2030097)
Übungsgruppe Di, 9-11 Uhr
Andreas Augustin

Aufgabe 1

Sei m die Anzahl von Operationen. Lege $2^n + 1$ Werte per `pushBack` auf das Array, mit $2^n + 1 \approx \frac{m}{2}$. Die interne Größe des Arrays ist dann gleich $2^n + 1$. Entfernt man nun wechselweise mit `popBack` ein Element und legt mit `pushBack` wieder eines auf das Array, so wird das Array bei jeder Operation auf die Hälfte verkleinert bzw. vergrößert, sodass jedesmal 2^n Elemente kopiert werden müssen. Die Laufzeit ist ungefähr $\frac{1}{4}4^{log_2 m}$. Mit $4^x = 2^x \cdot 2^x$ gilt:

$$\frac{1}{4}4^{log_2 m} = \frac{1}{4} \cdot 2^{log_2 m} \cdot 2^{log_2 m} = \frac{1}{4} \cdot m \cdot m = \frac{1}{4} \cdot m^2 \in \theta(m^2)$$

□

Aufgabe 2

(a)

worst-case-Laufzeit = $\lceil log_2 m \rceil$

(b)

- betrachte $m = 2^k - 1, \quad k \in \mathbb{N}$, sodass k Binärziffern benötigt werden
- sei $c = \sum_{i=0}^{k-1} d_i \cdot 2^i$
- ist $d_0 = 0$, so muss für das Inkrement nur 1 Bit verändert werden; dies stellt die Hälfte aller Fälle dar
- ist $d_0 = 1$ und $d_1 = 0$, so müssen zwei Bits verändert werden; dies stellt ein Viertel aller Fälle dar
- ...
- für $m = 2^k - 1$ Inkrementopertionen sind also

$$\sum_{i=1}^{k} \frac{2^k}{2^i} = 2^k - 1$$

  Bit-Änderungen notwendig; berechne die durchschnittliche Anzahl von Bit-Änderungen pro Inkrement-Operation

$$\frac{\sum_{i=1}^{k} \frac{2^k}{2^i}}{m} = \frac{2^k - 1}{2^k - 1} = 1 \in \mathcal{O}(1)$$

- ist m nicht von der Form $2^k - 1$, so erhöht sich die durchschnittliche Anzahl von Bit-Änderungen pro Inkrement-Operation, sie bleibt aber dennoch $\in \mathcal{O}(1)$. □

(c)

Betrachte $m = 2^k, \quad k \in \mathbb{N}$, sodass k Binärziffern benötigt werden. Führe $\frac{m}{2}$ Inkrementoperationen durch. Anschließend ist $c = \frac{m}{2} = 2^{k-1}$; die amortisierte Laufzeit beträgt $\frac{m}{2}$. Führe nun wechselweise je 2^{k-2} Dekrement- und Inkrementoperationen durch, beginnend mit einer Dekrementoperation. Da dann jeweils $log_2 m = k$ Bit-Änderungen vorgenommen werden müssen, beträgt die Laufzeit $\frac{m}{2} \cdot log_2 m$. In der Summe erhalten wir nun für diese m Operationen eine Laufzeit von $\frac{m}{2}(1 + log_2 m) \in \theta(m \cdot log_2 m)$. □

Aufgabe 3

```
#include "iostream"
#include "assert.h"

using namespace std;

template<class T>
class SList {
private:

  class Item {
    T _data;
    Item *_next;
  public:
    Item (const T &d) {
      _data = d;
      _next = (Item*)0;
    }

    Item (const T &d, Item *n) {
      _data = d;
      _next = n;
    }

    static void free_list (Item *i) {
      if (i != (Item*)0) {
        free_list (i->_next);
        delete i;
      }
    }

  public:

    T data () {
      return _data;
    }

    Item *next () {
      return _next;
    }

    void set_data (const T &d) {
      _data = d;
    }

    void set_next (Item *next) {
      _next = next;
    }
  };

private: // Data
  Item *_list;
```

```
  Item *_last;

protected:
  SList (Item *l) {
    _list = l;
  }

public:

  SList () {
    _list = (Item*)0;
    _last = (Item*)0;
  }

  SList (const T&d) {
    _list = new Item (d);
    _last = _list;
  }

  SList (const T&d, const SList<T> &l) {
    _list = new Item (d, l._list);
    _last = _list;
  }

  SList (const SList<T> &l) {
    _list = l._list;
    _last = _list;
  }

  bool is_empty () {
    return _list == (Item*)0;
  }

  T head () {
    return _list->data ();
  }

  SList<T> tail () {
    return SList(_list->next ());
  }

  void free_list () {
    Item::free_list (_list);
  }

  void push_front (const T &d) {
    _list = new Item (d, _list);
    if (_last == (Item*)0) {
      _last = _list;
    }
  }

  void push_back (const T &d) {
    Item *l = new Item (d);
```

```
    if (_last == (Item*)0) {
      _list = _last = l;
    } else {
      _last->set_next (l);
      _last = l;
    }
  }

  void pop_front () {
    assert (_list != (Item*)0);
    _list = _list->next;
    if (_list == (Item*)0) {
      _last = (Item*)0;
    }
  }

  void pop_frontD () {
    assert (_list != (Item*)0);
    Item *h = _list;
    _list = _list->next ();
    delete h;
    if (_list == (Item*)0) {
      _last = (Item*)0;
    }
  }
};

template<class T>
class FIFO {
private:
  SList<T> _list;
public:
  FIFO () : _list() {
  }

  ~FIFO () {
    _list.free_list ();
  }

public:
  const T first () {
    return _list.head ();
  }

  bool is_empty () {
    return _list.is_empty ();
  }

  void pushBack (const T &d) {
    _list.push_back (d);
  }

  void popFront () {
    _list.pop_frontD ();
```

```
  }
};

int main (int argc, char **argv) {
  FIFO<int> fifo;

  for (int i = 0; i < 40; i++) {
    fifo.pushBack (i);
  }

  for (int i = 0; i < 40; i++, fifo.popFront ()) {
    cout << i << ": " << fifo.first () << endl;
  }

  return 0;
}
```