Algorithm Engineering
Zusammenfassung Klausuraufgaben

17. Oktober 2018

1 Aufgabe

Geben Sie eine Spezifikation für den Datentyp `int_stack` (Keller von ganzen Zahlen) an und schreiben Sie eine entsprechende Header-Datei `int_stack.h` in C++. Geben Sie zwei Implementierungen für `int_stack` in C++ an:

int_stack.h

```
1  class int_stack{
2  private:
3      int* A;      //Feld 1
4      int size;    //Länge von A
5      int t;
6
7  public:
8
9      //Default-Konstruktor
10     int_stack();
11
12     //Konstruktor
13     int_stack(int size);     //Erzeugt einen Stack mit max. Größe size
14
15     //Destruktor
16     ~int_stack();
17     //Löscht den Stack
18
19     void push(int x);
20     //Fügt x als letztes Element an die Folge an
21
22     int top() const;
23     //Liefert das letzte Element. Precondition: Stack nicht leer
24
25     int pop();
26     //Liefert das letzte Element der Folge und gibt es zurück
27
28     bool empty() const;
29     //true, wenn Stack leer ist, sonst false
30 };
```

a) Eine Array-Datenstruktur

stack_array.cpp

```
#include "int_stack.h"

int_stack::int_stack(int n) {
    size = n;   //Länge des Feldes
    A = new int[size];
    t = -1; //Leeres Array
}

int_stack::int_stack() {
    int_stack(2);
}

int_stack::~int_stack() {
    delete[] A;
}

int_stack::push(int x) {
    if (t == size-1){
        //Stack ist voll
        int* B = new int[2*size];
        size = size*2;
        for (int i = 0; i <= t; i++) {
            B[i] = A[i];
        }
        delete[] A;
        A = B;  //Verschiebe Pointer
    }
    A[++t] = x; //Die eigentliche Push-Operation
}

int_stack::pop() {
    if (t == -1){
        EXCEPTION("Empty Stack");
    }
    return A[t--];
}

int_stack::top() {
    if (t == -1){
        EXCEPTION("Empty Stack");
    }
    return A[t];
}

int_stack::empty() {
    return (t == -1);
}
```

b) Eine dynamische Listendatenstruktur

stack_list.cpp

```
class int_stack {
```

```
2
3  public:
4      class stack_element{     //Listenelement
5          int value;
6          stack_element* next;
7          stack_element(int x, stack_element* p){
8              value = x;
9              next = p;
10         }
11     };
12
13 private:
14     stack_element* first;
15
16 public:
17
18     int_stack() {
19         first = NULL;
20     }
21
22     ~int_stack() {
23         //Aufgabe 1
24         while (!empty()){
25             pop();
26         }
27         //Aufgabe 4
28         delete first;
29     }
30
31     int pop() {
32         if (first == NULL) {return first;}
33         stack_element* p = first;
34         first = first->next;
35         return p;
36     }
37
38     void push(stack_element* p) {
39         p->next = first;
40         first = p;
41     }
42
43     int_stack top() {return first;}
44
45     bool empty() {return (first == NULL);}
46 };
```

1.1 Alternative Aufgabe:

Realisieren Sie den parametrisierten Datentyp `stack<T>` (Keller von elementen vom Typ T) durch ein Klassentemplate. Die maximale Größe des Stack soll im Konstruktor angegeben werden.

1.2 Aufgabe zur Antwort:

Verwenden Sie den *Stack* aus Aufgabe X und den Graphdatentyp `graph` aus der Vorlesung zur Implementierung der *Tiefensuche*(DFS). Schreiben Sie hierfür eine Funktion:

`list<node> dfs(graph& G, node s)`

die alle von s erreichbaren Knoten G als Liste zurückgibt.

dfs.cpp

```
list<node> dfs(graph& G, node s){
    list<node> result;
    stack<node> S;
    node n;
    edge e;
    for_each(n,G){
        G.set(n,0); //unbesucht setzen
    }
    G.set(s,1);
    S.push(s);
    while (!S.empty()){
        s = S.pop();
        result.append(s);
        for_each(e,s){  //alle out edges
            n# = G.target(e);
            if(G.get(n) == 0){
                S.push(n);
                G.set(n,1); //besucht setzen
            }
        }
    }
    return result;
}
```

1.3 Weitere Aufgabe:(Point & Student List)

Implementieren Sie generische einfach verkettete Listen durch Vererbung. Hierzu müssen zwei Basisklassen für die Listenelemente und die Liste selbst definiert werden, von denen dann jeweils die Objekte und die eigentliche Liste abgeleitet werden. Demonstrieren Sie die Verwendung am Beispiel einer Klasse `point` für Punkte in der Ebene, die in einer Liste `point_list` abgespeichert werden sollen.

Implementieren Sie generische einfach verkettete Listen durch Vererbung. Hierzu müssen zwei Basisklassen definiert werden, einmal für die Listenelemente und eine für die Liste selbst. Von diesen sollen dann jeweils die Objekte und die eigentliche Liste abgeleitet werden. Demonstrieren Sie die Verwendung am Beispiel einer Klasse `student` für Punkte in der Ebene, die in einer Liste `student_list` abgespeichert werden sollen.

list_element.cpp

```
class list_element{
    list_element* next;
    list_element(list_element* p){
        next = p;
    }
};
```

List Klasse gleiche wie stack_list.cpp (Stack mit Listenstruktur)

point_list.cpp

```
stack_list point_list;
point* p = new point(x,y);
point_list->push(p);
point* q = (point*) point_list->top();
point* q = (point*) point_list->pop();

1.Leite point von list_element ab: class point: public slist_elem{...};
2.list kann nun für point verwendet werden
```

2 Aufgabe

Spezifizieren Sie den Datentyp `int_queue` (Schlange von ganzen Zahlen) durch eine entsprechend Header-Datei `int_queue.h`. Geben Sie eine array-basierte Implementierung an (`int_queue.cpp`).

queue.h

```
template <class T>

class queue{

private:
    T* A;
    int size;
    int t;

public:

    //Default-Konstruktor erzeugt leere Queue
    queue();

    //Konstruktor erzeugt leere Queue der Größe x
    queue(int size);

    //Destruktor, löscht die Queue
    ~queue();

    void append(T* x);
    //Fügt x am Ende der Queue an

    T pop();
    //Precondition: Queue nicht leer

    bool empty() const;
    //true, wenn Queue leer ist, sonst false

    int length();
    //Gibt die Länge der Queue zurück
};
```

queue.cpp

```
template <class T>
class queue{
    #include "queue.h"

public:
    queue::queue(int t){
        size = x;
        A = new T[size];
        t = -1
    }

    queue::queue(){
        queue(10);
    }

    queue::~queue(){
        delete[] A;
    }

    void queue::append(T* x){
        if (t == size-1){
            size = 2*size;
            int i;
            for (int i = 0; i <= t; i++) {
                B[i] = A[i];
            }
            delete[] A;
            A = B;
        }
        A[++t] = x;
    }

    T queue::pop(){
        if (t == -1){
            EXCEPTION("Leere Queue");
        } else { //Warum hier else und bei Stack nicht???
            int i:
            T x = A[0];
            for (i = 0; i < t; i++) {
                A[i] = A[i+1];
            }
            t--;
            return x;
        }
    }

    bool queue::empty(){return (t==1);}

    int queue::length(){return t;}

};
```

2.1 Alternative Aufgabe:

Realisieren Sie den parametrisierten Datentyp `queue<T>` (Schlange von Elementen vom Typ T) durch ein Klassentemplate. (Die maximale Länge der Queue soll im Konstruktor angegeben werden)

2.2 Aufgabe zur Antwort:

Verwenden Sie die Schlange aus Aufgabe X und den Graphdatentyp `graph` aus der Vorlesung zur Implementierung der *Breitensuche*(BFS). Schreiben Sie hierfür eine Funktion:

`list<node> BFS(const graph& G, node s)`

die alle von s erreichbaren Knoten G als Liste zurückgibt.

bfs.cpp

```
1  list<node> bfs(graph& G, node s){
2      list<node> result;
3      queue<node> Q;
4      node n;
5      edge e;
6      for_each(n,G){
7          G.set(n,0); //unbesucht setzen
8      }
9      G.set(s,1);
10     Q.append(s);
11     while (!Q.empty()){
12         s = Q,pop();
13         result.append(s);
14         for_each(e,s){  //alle out edges
15             n# = G.target(e);
16             if(G.get(n) == 0){
17                 Q.append(n);
18                 G.set(n,1); //besucht setzen
19             }
20         }
21     }
22     return result;
23 }
```

3 Aufgabe

Überlegen Sie sich eine Spezifikation für den Typ Wörterbuch (`dictionary`), also eine Header-Datei `dictionary.h` zum Speichern von Paaren aus Strings und ganzen Zahlen. Verwenden Sie den Datentyp, um das sogenannte `word count` Problem zu lösen. Dabei ist eine Folge von Strings gegeben und es soll für jeden String seine Häufigkeit in der Eingabefolge berechnet und ausgegeben werden.

dictionary.h

```
1  class dictionary{
2      //Instanz vom Typ dictionary <String, int> ist eine partielle
           Ableitung Abb d:String->int, mit dom(d) ist Definitionsbereich
3
4  public:
5      //Konstruktor, erstellt leeres Wörterbuch
6      dictionary();
```

```
7
8       //Destruktor, löscht Wörterbuch
9       ~dictionary();
10
11      void insert(String k, int i);
12
13      void delete(String k);
14
15      bool empty();    //true, falls dom(d)=leere Menge, sonst false
16
17      int lookup(String k);
18
19      bool defined(String k); //true, falls k /in dom(d), sonst false
20  };
```

wordcount.cpp

```
1   void wordcount() {
2       dictionary<String, int> D;
3       String s;
4       for (s:Eingabe) {
5           if (!D.defined(s)){
6               D.insert(s,1);
7           } else {
8               D.insert(s,D.lookup(s)+1);
9           }
10      }
11      for (s:D){
12          cout<<s<<":"<<D.lookup(s)<<"/n";
13      }
14  }
```

4 Aufgabe

Schreiben Sie ein Funktionstemplate `MERGE(a, b, c, d, e)`, das 5 Iteratoren als Argumente nimmt und die aufsteigend sortierten Folgen die durch die Iteratoren **a** und **b** bzw. **c** und **d** gegeben sind zu einer aufsteigenden Folge zusammen mischt und diese beginnend mit der Position, die durch den Iterator **e** definiert ist, ausgibt. Nehmen Sie hierzu an, dass der <-Operator, für die betreffenden Werte definiert ist.

merge.cpp

```
1   template <class Iterator>
2   void MERGE(Iterator a, Iterator b, Iterator c, Iterator d, Iterator e){
3       while (a != b || c != d){
4           *e++ = (c == d || (*a < *c && a != b)) ? *a++ : *c++;
5       }
6   }
```

5 Aufgabe

Schreiben Sie ein Funktionstemplate `palindrom`, das zwei Iteratoren in einem beliebigen Container als Argumente nimmt und testet, ob die Folge der Elemente im entsprechenden intervall ein palindrom ist, d.h. vorwärts wie rückwärts gelesen gleich ist.

palindrome.cpp

```
1 template <class T>
2 bool palindrom (Iterator a, Iterator b){
3     --b;     //zeigt sonst auf "\o"
4     while (a < b && a* == b*){
5         ++a;
6         --b;
7     }
8     return a >= b;
9 }
```