Informatik für Mathematiker und Physiker  HS15

Solutions 13


Solution 1

i) For \( N = 7 \) there are 3 arrays, and for \( N = 33 \) there are 6 arrays.

**Remark:** In general, the number of arrays is equal to the number of binary digits of the number of elements on the stack. This is since the first array has length 1, and each new array has twice the length of the previous one. Thus the length of the \( k \)-th array is \( 2^{k-1} \) (i.e. by induction). And therefore with \( n \) such arrays we can store at most \( \sum_{k=1}^{n} 2^{k-1} = \sum_{i=0}^{n-1} 2^i \) elements. But the given sum simply corresponds to the decimal value of the binary number 11...1 (i.e. \( n \) binary digits, all of which are 1).

ii) [1]: 0
    [2]: node
    [3]: node->next
    [4]: tmp

**Note:** Here we use a pointer `node` which we shift over all the `Nodes` in the `DoublingStack` and delete them. For this we have to use `delete` as the `Nodes` were created using `new`. We have to be careful though to not dereference a pointer to an object that is already deleted. This is why a `tmp`-pointer is used to memorize the current node while we can already shift our `node` pointer to the next `Node` (while the current `Node` still hasn’t been deleted). Then we can use the `tmp`-pointer to delete the `Node`.

iii) [5]: 0
    [6]: top->prev
    [7]: top->size
    [8]: top->next
    [9]: top->next = 0
    [10]: top->values[--pos]

**Note:** In the if-statement we remove the current top `Node` if it is empty. To do this we first readjust `top` to point to the `Node` before. Then we adjust `pos` to indicate that this `Node` is fully occupied. In the delete-statement we then remove the empty `Node`. But
there is still a pointer in the new top-node which is pointing to the deleted Node and which we thus have to set to 0. Finally, in gap [10] we have to remove the value on top of the stack and return it.

Solution 2

This is very similar to the stack, except that we maintain an additional pointer to the last node.

The full header file:

```cpp
// Informatik − Serie 13 − Aufgabe 2
// Programm: queue.h
// Autor: ... (Gruppe ...)

#include <iostream>
#include <cassert>

namespace ifmp
{

    // linked list node
    struct list_node {
        int key;
        list_node* next;
        list_node (int k, list_node* n = 0) : key (k), next (n) {}
    };

    class queue{
    public:
        // default constructor:
        // POST: *this is an empty queue
        queue () : first_node (0), last_node (0) {}

        // destructor
        ~queue () {
            clear (first_node);
        }

        // copy constructor
        queue (const queue& s) : first_node (0), last_node (0) {
            copy (s.first_node, first_node);
        }
    };

    // Solution 2
    // This is very similar to the stack, except that we maintain an additional pointer to the last node.

    // stack
    class stack{
    public:
        // default constructor:
        // POST: *this is an empty stack
        stack () : top (0) {}

        // destructor
        ~stack () {
            clear (top);
        }

        // copy constructor
        stack (const stack& s) : top (0) {
            copy (s.top, top);
        }
    };

    typedef stack stack_t;
    typedef queue queue_t;
}
```
// assignment operator
queue& operator=(const queue& s)
{
    if (first_node != s.first_node) { // avoid self-assignment
        clear(first_node);
        first_node = 0;
        last_node = 0;
        copy(s.first_node, first_node);
    }
    return *this;
}

// POST: key is added as last element
void push_back(int key)
{
    if (first_node == 0)
        first_node = last_node = new list_node(key);
    else {
        last_node->next = new list_node(key);
        last_node = last_node->next;
    }
}

// POST: returns whether *this is empty
bool empty() const
{
    return first_node == 0;
}

// PRE: !empty()
// POST: first element of *this is returned
int front() const
{
    assert(!empty());
    return first_node->key;
}

// PRE: !empty()
// POST: last element of *this is returned
int back() const
{
    assert(!empty());
    return last_node->key;
}

// PRE: !empty()
// POST: first element of *this is removed
void pop_front()
{
    assert (!empty());
    if (first_node == last_node)
    {
        last_node = 0;
        list_node* p = first_node;
        first_node = first_node->next;
        delete p;
    }
}

// POST: *this is written to o
void print (std::ostream& o) const
{
    const list_node* p = first_node;
    while (p != 0) {
        o << p->key << " ";
        p = p->next;
    }
}

private:
list_node* first_node;
list_node* last_node;

// PRE: to == 0, last == 0
// POST: nodes after from are copied to nodes after to in *this
//       if at least one node was copied, last_node holds the
//       address of the last node in *this
void copy (const list_node* from, list_node*& to)
{
    assert (to == 0);
    if (from != 0) {
        to = new list_node (from->key);
        last_node = to;
        copy (from->next, to->next);
    }
}

// POST: nodes after from are deleted
void clear (list_node* from)
{
    if (from != 0) {
        clear (from->next);
        delete (from);
    }
};
Solution 3

The full header file:

```cpp
namespace ifmp {
    class vector {
        public:
            typedef int* iterator;
            typedef const int* const_iterator;

            // default constructor
            // POST: *this is an empty vector
            vector () : begin_(0), end_(0), end_of_memory_(0) {}

            // constructor for size
            // POST: *this is a vector of size s
            vector (const unsigned int s) {
                begin_ = new int[s];
                end_ = end_of_memory_ = begin_ + s;
            }

            // constructor for size and init value
            // POST: *this is a vector of size s, elements initialized
            // with val
            vector (const unsigned int s, const int& val) {
                begin_ = new int[s];
                end_ = end_of_memory_ = begin_ + s;
                for (iterator it = begin_; it != end_; ++it) *it = val;
            }
```

```
// copy constructor
vector (const vector& v) {
    begin_ = new int[v.size()];
    end_ = end_of_memory_ = copy(v.begin_, v.end_, begin_);
}

// assignment operator
vector& operator= (const vector& v) {
    if (begin_ != v.begin_) { // avoid self-assignment
        if (capacity() < v.size()) increase_capacity(v.size());
        end_ = copy(v.begin_, v.end_, begin_);
    }
    return *this;
}

// destructor
vector () {
    if (begin_ != 0) delete[] begin_;
}

// POST: *this was emptied
void clear () {
    end_ = begin_;
}

// POST: returns the size of the vector
unsigned int size () const {
    return end_ - begin_;
}

// POST: returns the capacity of the vector
unsigned int capacity () const {
    return end_of_memory_ - begin_;
}

// POST: element with value val is appended to *this, // capacity of *this was increased if necessary
void push_back (const int& val) {
    if (end_ == end_of_memory_) increase_capacity();
    *(end_++) = val;
}

// PRE: *this contains at least one element // POST: last element of *this was removed
void pop_back () {
    assert(begin_ < end_);
    --end_;
}
// PRE: *this contains at least one element
// POST: returns the value of the first element in *this
int front() const {
    assert(begin_ < end_);
    return *begin_;}

// PRE: *this contains at least one element
// POST: returns the value of the last element in *this
int back() const {
    assert(begin_ < end_);
    return *(end_ - 1);}

// POST: returns an iterator pointing to the first element
iterator begin() {
    return begin_;}

// POST: returns an iterator pointing to the first element where
// modifying the range is not allowed.
const_iterator begin() const {
    return begin_;}

// POST: returns a past-the-end iterator
iterator end() {
    return end_;}

// POST: returns a past-the-end iterator where modifying the
// range is not allowed.
const_iterator end() const {
    return end_;}

// PRE: n < size()
// POST: returns a reference to the n-th element
int& operator[](const unsigned int n) {
    assert(n < size());
    return *(begin_ + n);}

// PRE: n < size()
// POST: returns a const reference to the n-th element
const int& operator[](const unsigned int n) const {

assert(n < size());
return *(begin_ + n);
}

// POST: *this is written to o
void print (std::ostream& o) const {
    const int* p = begin_;  
    while (p < end_)
        o << *(p++) << " ";
}

private:
    iterator begin_;  
    iterator end_;  
    iterator end_of_memory_;  

    // PRE: [begin, end) is a valid range, [to, to + end−begin)
    // points to allocated memory and is not overlapping
    // POST: elements in [begin, end) are copied to the range
    // [to, to + end−begin), returns an iterator pointing to
    // the end of the destination range
    iterator copy (iterator begin, iterator end, iterator to) {
        for(iterator it = begin; it != end; ++it)
            *(to++) = *it;
        return to;
    }

    // POST: *this has twice as much capacity as before or at least 1
    void increase_capacity () {
        if (size() == 0) increase_capacity (1);  
        else increase_capacity(2 * size());
    }

    // PRE: c >= size()
    // POST: *this has capacity at least c
    void increase_capacity (const unsigned int c) {
        assert(c >= size());
        iterator temp = new int[c];
        if (begin_ == 0)  
            end_ = temp;
        else {
            end_ = copy(begin_, end_, temp);
            delete[] begin_;
        }
        begin_ = temp;
    end_of_memory_ = temp + c;
    }

    } // end namespace

    // POST: v is written to o
    std::ostream& operator<<(std::ostream& o, const vector& v) {
        v.print (o);
        return o;
    }
} // end namespace