Tom Jenkins

Ms. Sanders

Programming Languages

27 October 2016

Flexible Division of the Array for Stack Implementation

What we need to do is implement three stacks using the arrays in C++. We have already implemented the stack using arrays with the fixed division, and now let's try to do it in a more interesting way. This approach implies the flexible allocation of the space to stack blocks. When one of the stacks is too big to fit in the original space, we increase the amount of resources needed and move items if necessary.

In addition, we can create an array so that the last stack will start at the end of the array, and end at the beginning. It will be some kind of "looped" array.

Here is the implementation of the algorithm:

```
public class data{
    public int begin;
    public int ptr;
    public int length = 0;
    public int capacitance;
```

```
public data(int _begin, int _capacitance){
    begin = _begin;
    ptr = _begin -1;
    capacitance = _capacitance;
}

public boolean isInStach(int ind, int real_lenght){
    if(begin + capacitance <= real_lenght) { // real size of the stack
        if(begin <= ind && ind <= begin + capacitance) {
            return true;
        } else {
            return false;
        }
    } else {    // stack is cut off in the beginning of the array
        int moved_ind = ind + real_lenght;
        if (begin <= moved_ind &&
        moved_ind <= begin + capacitance){
            return true;
        } else {
```

```
            return false;
        }
    }
}

public class Question A {
    static int stacksAmount = 3;
    static int defaultLength = 4;
    static int real_lenght = defaultLength * stacksAmount;
    static StackData [] stacks = {new StackData(0, defaultLength),
    new StackData(defaultLength, defaultLength),
    new StackData(defaultLength * 2, defaultLength)};
static int [] buf = new int [real_lenght];

public static void main(String [] args) throw Exception {
    push(0,1);
    push(1,2);
    push(2,3);
```

```
    int q = pop(0);
}

public static int nextEl(int ind) {
    if (ind + 1 == real_lenght) return 0;
    else return ind + 1;
}

public static int prevEl(int ind) {
if (ind ==0) return real_lenght - 1;
    else return ind - 1;
}

public static void move(int stackNum) {
    StackData stack = stacks[stackNum];
    if (stack.length >= stack.capacitance) {
        int nextStack = (stackNum + 1) % number_of_stacks;
        move(nextStack); // move the stack
        stack.capacitance++;
```

```
}

//move elements backwards
for (int i = (stack.begin + stack.capacitance -1) %  real_lenght;
   stack.isInStach(i, real_lenght);
   i=prevEl(i)) {
      buf[i] = buf[prevEl(i)];
   }

   buf[stack.begin] = 0;
   stack.begin = nextEl(stack.begin); //move the beginning of the stack
   stack.ptr = nextEl(stack.ptr); // move the pointer
   stack.capacitance--; // set the original size
}

/* enlarge the stack, move the other stacks */
public static void enlarge(int stackNum) {
   move((stackNum + 1) % number_of_stacks);
   stacks[stackNum].capacitance++;
```

```
}

public static void push(int stackNum, int value)
throws Exception {
   StackData stack = stacks[stackNum];
   //check the space
   if (stack.length >= stack.capacitance) {
      if (numberOfElements() >= real_lenght) { // Totally full
         throw new Exception("Not enough space");
      } else {    // move the stack
         enlarge(stackNum);
      }
   }
   /* find the index of the top element of the
   array +1 and increase the pointer of the stack */
   stack.length++;
   stack.ptr = nextEl(stack.ptr);
   buf[stack.ptr] = value;
}
```

```
public static int pop(int stackNum) throws Exception{
   StackData stack = stacks[stackNum];
   if (stack.length == 0) {
      throw new Exception("You are trying to use the empty stack");
   }
   int value = buf[stack.ptr];
   buf[stack.ptr] = 0;
   stack.ptr = prevEl(stack.ptr);
   stack.length--;
   return value;
}

public static int peek(int stackNum) {
   StackData stack = stacks[stackNum];
   return buf[stack.ptr];
}

public static boolean isEmpty(int stackNum) {
   StackData stack = stacks[stackNum];
```

```
        return stack.length == 0;
    }
}
```

This is more complicated algorithm than the one with the fixed division, but it is more optimal and gives more opportunities to work with stacks.

Thanks for your attention!