Basic Data Structures

Important note

Java has implemented many data structures, such as stack, queue, lists, maps and many others. Usually we use ready-made structures, we do not need to reinvent it. Follow some implementations (after all, the most simplified) just for better understanding the principles, not for practical use.

Overview

There are many important and interesting data structures. Their description is mostly beyond the scope of this paper. However, we introduce some other structures in other places, as described below.

- Some data structures are described herein. They are:
- Dynamic array
- Queue
- Circular queue (circular buffer)
- Stack

Other data structures are explained in other places. These are the following structures:

- Binary heap - see chapter topic collection in Java2,
- Binomial heap - see chapter topic collection in Java2,
- Disjoint-set - see chapter topic collection in Java2,
- D-regular heap - see chapter topic collection in Java2,
- Graf - see Graphs
- Hash table - see Search,
- Set - see chapter topic collection in Java2,
- Multisets - see chapter topic collection in Java2,
- Linked list - see Lists

- Tree - see Trees.

Dynamic Resizing of Array

Dynamic resizing of an array is no data structure. But it is the technology often used when working with data structures, so I will explain it right at the beginning.

Dynamic resizing is available in Java. Many other programming languages (eg. Pascal) does not allow it.

Dynamic array allows you to add any number of elements, thus eliminating the main disadvantage of the classical array - fixed size.

Dynamic array stores its elements into the fixed length internal array; when the capacity of the array is exhausted, it will allocate a new larger array (typically 2x larger because of amortization), all elements of the original array are copied and old field deallocated.

In a similar way the container maintains, if it is too empty (vacancy ratio exceeds a certain limit) - smaller array is allocated and again the values are copied to it.

Typical operations

Inserting

Inserting: At first glance, the procedure for inserting an element is inefficient, because at the moment there is no place *array* must copy all the data. Asymptotic complexity of insertion element is therefore O (n).

If the program runs long enough, this complexity is amortized. This means that the cost of performing this operation is decomposed in time.

Thanks to amortized complexity of the structure dynamic arrays are very fast, although asymptotic complexity of its operations it would not say.

On the other hand, its use is not appropriate in *real-time* systems, which must guarantee a certain response time.

Deleting

Deleting: The complexity of the operation of deleting an element depends on the specific implementation. If deleted element only swap with the last element and a space at the end of the container is released (or declared as unoccupied) then and asymptotic complexity is O (1), but there is a shift of elements. If removal is done so that all other elements moves one place, then will be asymptotic complexity of O (n), but the order of elements stays unchanged.

Reading

Reading: Reading the element at index *i* will be in time O (1) because the array allows random access.

Stack

Significance

Stack is one of the basic data structures, which is e.g. used for temporary storage during the calculation.

Stack stores data in a way which is called **LIFO** - last in, first out. The last inserted element is outputed as the first, next as second, and so on.

Note: The datatype queue, FIFO - first in, first out, works the opposite way.

The stack is mainly used in computer science for saving the state of algorithms and programs. It is used in a depth-first search and implicitly in all recursive algorithms.

Virtual machines for Java and Lisp are built on stack architecture.

Basic Operations

- Abstract data type **stack** is specified by the following operations:
- push - inserts an element on top of the stack
- pop - removing the top tray
- top - query on top of the stack
- isEmpty - asking for an empty container (or size - a question on stack size)

For handling the stored data items is introduced a „stack pointer“, which indicates the relative address of the last added items, so-called **top of the stack**.

In most programming languages stack may be relatively easily implemented using an array, or a linear list.

What identifies a data structure as a stack is not an implementation is, but the interface - user can only add (*push*) or remove (*pop*) values from the array or linear list, along with a few auxiliary functions.

Implementation using an Array

Implementation using array focuses on creating a array where the zero index (*array [0]*) is the bottom of the stack.

The program stores in an auxiliary variable the index of current top of the stack - this index varies depending on how the stack data content increases or decreases.

Example

The following is a sample of how it was to create a stack using an array. The field is dynamic in the sense that if at the beginning we had not correctly guessed the required capacity of the stack, later we can create a new, larger array. In Java this works, unlike many other languages.

Let's look at a few attractions that are included in the example. The object-oriented approach is respected: the stack is formed as a completely separate class *StackArray*. The main program creates an instance of *StackArray* (named *demo*) and then repeatedly calls its methods *push()* and *pop()*. Here is the detail:

```
public static void main(String[] args) {

    StackArray<String> demo =
       new StackArray<>();

    demo.push("Amsterdam");
    demo.push("Bonn");
    demo.push("Cairo");
    demo.push("Detroit");
    demo.push("El Dorado");

    System.out.println(demo.pop());
    System.out.println(demo.pop());
    System.out.println(demo.pop());
    System.out.println(demo.pop());
    System.out.println(demo.pop());
}
```

At first glance you notice strange things: the name of the type "*StackArray*" is the term `<String>`. So far we have never met with the name of a type that would be closed into the pointy brackets `< >`. These are so-called generic types. We will discuss in greater detail in another subject.

For ease of understanding may help us the following comparisons. When we work with methods (functions), so we pass parameters in parentheses. When you define a function, we name the parameters with which we perform calculations.

These are formal parameters. Then, when we call the function, formal parameters are substituted by the specific values, actual parameters.

Importantly, the formal and actual parameters are **variables** in methods. As an example, we can imagine the function `int add (int i, int j) {...}` which adds two integers. Similarly, we could have a function `double add(double i, double j){…}` for variables of type *double* and another function `String add(String i, String j){…}` for "addition" values of type *String*.

In Java, we solved it by using function overloading.

But there is another way. Like we had a formal and actual parameters, we can also introduce formal and real types. These types will enter into pointy brackets. Our *add* function we could write as `<T> add (<T> i, <T> j) {...}`. Here *<T>* is a placeholder for some type. Later, when you use the function, it will bereplaced by any actual type, like *int*, *double*, or *String*.

Back to the stack. In file *StackArray.java* symbol `<E>` marked type of data that will be stored in the stack. Furthermore *DEFAULT_CAPACITY* is an integer constant, which indicates the initial length of the array. Array is formed of length 10 at the beginning and, whenever necessary, the method *doubleSize()* doubles its length. Note that the field is defined as an array of elements of type Object. That's okay, because the type Object is the default ancestor for each class. But just as well we could have defined array using formal type <E>, we'd save you the need to cast (for example, line 51 is `E = e (E) elements[-- size]`).

The function *push()* is used to initialize the stack and to add new value to its top.

It is responsible for the change in value of variable *size* and inserting a new element into the array `elements [size ++] = e;` . The function also verifies whether the stack is not full; If the array is filled, and that control would not be here, would lead to error.

Function *pop()* removes one element from the stack and decrements the number of elements.

It then verifies if the stack is not empty - if we did not, an attempt to remove elements from an empty stack would led to error.

Sources are in this file.

Implementation using Linked List

Stack of course can be implemented using a linked list, as will be explained later. It is not extremely advantageous.

Here is a sample of key parts of the code. It is based on an internal class Node, which acts as a data container. Contains useful data (*value*), a reference to another instance of the same class (*next*), and possibly also constructor *Node ()*.

```
    private class Node {
        private String value;
        private Node next;
        private Node(String value) {
            this.value = value;
        }
    }
```

Stack itself has a pointer *first* to the first element of the stack, integer number of elements *size* and a constructor, which resets the *size*.

```
public class StackList {
    private Node first;
    private int size;
    public StackList() {
        this.size = 0;
    }
```

The function can be illustrated using the example of the method *push*, which adds a new element to the top of the stack.

```
    public void push(String i) {
        Node n = new Node(i);
        Node currFirst = first;
        first = n;
        n.next = currFirst;
        size++;
    }
```

The method creates a new node *Node* and right in the constructor populates its value. Using an auxiliary variable *currFirst* connects this element into the list and finally increments the number of elements.

Sources are in this file.

Queue

Significance

The queue is one of the basic data types used to store data in such a way that the element that has been stored as the first, will also picked-up first.

This principle is called **FIFO** - first in, fist out.

Note: The opposite approach - LIFO - last in, first out – uses stack.

A special case of queue is the so-called **priority queue**. Priority queue in which the elements may anticipate a higher priority at the output of those with lower priority.

- The queue has a wide application in computer science, these are some examples:
- Operator pipe - communication between processes in operating systems
- Circular buffer - buffer for data flows

- Sort priority using heap - heapsort

Typical operations

Here are some operations that a typical Queue implements:

- *addLast* (enqueue) - Inserts an element into the queue.
- *deleteFirst* (poll, dequeue) - Gets and removes the first element (head) of the queue.
- *getFirst* (peek) - get the first element of the queue.
- *isEmpty* - Ask the emptiness of the queue.
- *size* - Returns the number of elements contained

If implemented by an array, the queue itself has very limited use, because the queue is growing steadily and never diminish.

Therefore rather uses circular queue (circular buffer), wich does not suffer from this defect.

However, implementation of the queue using the list is often used because it is simple and fast. It is very similar to the implementation of the tray. The full source code is here.

Circular Queue

As storage space for the queue in the simplest case, we can use an ordinary array. When running, data items are placed one after the other, ie. item 1, item 2, item 3 and so on.

Data is read back sequentially as well.

Queue with this solution has a limited duration (if we define array at the beginning of the program in the form of eg. `int [1..1000]`, such an array can have a maximum of 1000 places for storing data in the queue).

By writting and reading data, the head of queue gradually shifts to end of the array. There is stopped.

A simple solution would be, after each reading to shift all the items in the array to move them to the beginning. But this is not the best, especially for large amounts of data.

That is why it is used a **circular queue** (*circular buffer*). Circular queue, when it comes to the entry end of the array, writes again into the free space at the beginning.

Preferably use the **modulo** operator.

We must never forget the importance of checking the queue. If the space from which we read is empty (or into which we write is full, contains unread data), underflow or overflow of queue occures.

- We introduce two variables:
- **K** will be the index of the first field where there are not written data (ie. K will point after the end of data),
- **Z** will be the index of the field, where the data that we have not read (ie. Z will point at the beginning of data)
- <ENWe introduce two variables:

- **K** is the index of the field where there are no written data (ie. will point at the end of the data)
- **Z** is the index of the field, where are data that we have not read yet (ie. at the beginning of the data).

In writing, we will proceed as follows:

- first check if new end K will not catch the beginning Z,
- where points K, data should be entered,
- increment K to the next index.

And the queue is ready for next write.

When reading it is read from the location pointed to by Z and after reading Z moves to the next position. Underflow must be checked (ie. moved Z must not equal K).

Implementation by an Array

Circular queue can be implemented using array.

The source code is included in this file. Here are some key details:

```
public class CircularQueueArray<TYPE> {

    private int size;
    private final Object[] array;
    private int pointer; //first free index
```

```
/**
 * Constructor
 * @param length initial size of array
 */
public CircularQueueArray(int length) {
    this.array = new Object[length];
    this.size = 0;
    pointer = 0;
}

/**
 * Append item
 * @param i item
 */
```

```
    public void addLast(TYPE i) {
        if (this.size == array.length) {
            throw new
IllegalStateException("Buffer full");
        }
        array[pointer] = i;
        pointer = modulo((pointer + 1),
array.length);
        size++;
    }
```

Again, is used formal type <*TYPE*> and again used a dynamic extension of the array, as we had previously seen.

More important data structures

Content