DATA STRUCTURES AND ALGORITHMS

LECTURE 14

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 13...

- AVL Trees
- Huffman encoding

Today

- Parenthesis matching
- Perfect hashing
- Cuckoo hashing
- Linked Hash Table
- Conclusions
- Exam info

Delimiter matching

- Given a sequence of round brackets (parentheses), (square) brackets and curly brackets, verify if the brackets are opened and closed correctly.
- For example:
  - The sequence ()([][][(())]) - is correct
  - The sequence [()()()()] - is correct
  - The sequence [()]) - is not correct (one extra closed round bracket at the end)
  - The sequence [(]) - is not correct (brackets closed in wrong order)
  - The sequence {[[]] () - is not correct (curly bracket is not closed)

Bracket matching - Solution Idea

- Stacks are suitable for this problem, because the bracket that was opened last should be the first to be closed. This matches the LIFO property of the stack.
- The main idea of the solution:
  - Start parsing the sequence, element-by-element
  - If we encounter an open bracket, we push it to a stack
  - If we encounter a closed bracket, we pop the last open bracket from the stack and check if they match
  - If they don't match, the sequence is not correct
  - If they match, we continue
  - If the stack is empty when we finished parsing the sequence, it was correct

Bracket matching - Extension

- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch

Bracket matching - Extension

- How can we extend the previous idea so that in case of an error we will also signal the position where the problem occurs?
- Remember, we have 3 types of errors:
  - Open brackets that are never closed
  - Closed brackets that were not opened
  - Mismatch
- Keep count of the current position in the sequence, and push to the stack $< delimiter, position >$ pairs.

Perfect hashing

Perfect hashing

- Assume that we know all the keys in advance and we use *separate chaining* for collision resolution $\Rightarrow$ the more lists we make, the shorter the lists will be (reduced number of collisions) $\Rightarrow$ if we could make a large number of list, each would have one element only (no collision).
- How large should we make the hash table to make sure that there are no collisions?
- If $M = N^2$, it can be shown that the table is collision free with probability at least $1/2$.
- Start building the hash table. If you detect a collision, just choose a new hash function and start over (expected number of trials is at most 2).

Perfect hashing

- Having a table of size N^2 is impractical.
- Solution instead:
  - Use a hash table of size N (*primary* hash table).
  - Instead of using linked list for collision resolution (as in separate chaining) each element of the hash table is another hash table (*secondary hash table*)
  - Make the secondary hash table of size ${n_j}^2$, where n_j is the number of elements from this hash table.
  - Each secondary hash table will be constructed with a different hash function, and will be reconstructed until it is collision free.
- This is called **perfect hashing**.
- It can be shown that the total space needed for the secondary hash tables is expected to be at most $2N$ (if it is larger, just pick a different hash function).

Perfect hashing

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.

Perfect hashing

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.
- Let p be a prime number, larger than the largest possible key.
- The universal hash function family $\mathcal{H}$ can be defined as:

$$\mathcal{H} = \{H_{a,b}(x) = ((a * x + b) \,\%\, p) \,\%\, m)$$

$$\text{where } 1 \leq a \leq p-1, 0 \leq b \leq p-1$$

- a and b are chosen randomly when the hash function is initialized.

Perfect hashing - example

- Insert into a hash table with perfect hashing the values 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39
- Since we are inserting $N = 13$ elements, we will take $m = 13$.

Perfect hashing - example

- p has to be a prime number larger than the maximum key $\Rightarrow$ 151
- The hash function will be:

$$H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m$$

- where a will be 3 and b will be 2 (chosen randomly).

Value	76	12	109	43	22	18	55	81	91	27	13	16	39
H(Value)	1	12	1	1	3	4	3	3	7	5	2	11	2

Perfect hashing - example

- Collisions:
  - position 1 - 76, 109, 43
  - position 2 - 13, 39
  - position 3 - 22, 55, 81
  - position 4 - 18
  - position 5 - 27
  - position 7 - 91
  - position 11 - 16
  - position 12 - 12
- Sum of the sizes of the secondary hash tables: $9 + 4 + 9 + 1 + 1 + 1 + 1 + 1 = 27$

Perfect hashing - example

- For the positions where we have no collision (only one element hashed to it) we will have a secondary hash table with only one element and hash function $h(x) = 0$
- For the positions where we have two elements, we will have a secondary hash table with 4 positions and different hash functions, taken from the same universe, with different random values for a and b.
- For example for position 2, we can define $a = 4$ and $b = 11$ and we will have:
  $h(13) = 3$
  $h(39) = 0$

Perfect hashing - example

- Assume that for the secondary hash table from position 1 we will choose $a = 14$ and $b = 1$.
- Positions for the elements will be:
  $h(76) = ((14 * 76 + 1)\%151)\%9 = 8$
  $h(109) = ((14 * 109 + 1)\%151)\%9 = 8$
  $h(43) = ((14 * 43 + 1)\%151)\%9 = 6$
- In perfect hashing we should not have collisions, so we will simply chose another hash function: another random values for a and b. Choosing for example $a = 2$ and $b = 13$, we will have h(76) = 5, h(109) = 8, h(43) = 0.

Perfect hashing

- When perfect hashing is used and we search for an element we will have to check at most 2 positions (position in the primary and in the secondary table).
- This means that the worst case performance of the table is $\Theta(1)$.
- But in order to use perfect hashing, we need to have static keys: once the table is built, no new elements can be added.

Dynamic Perfect Hashing

- Traditionally, perfect hashing is said to work in a static environment (you need to know all the keys in advance).
- It is easy to see why: you can build a table to be collision free, by picking new hash functions. But if you allow new additions you might get a collision after you have built the table.
- However, dynamic perfect hashing was also introduced in 1994.
- It obviously implies a lot of rebuilding when a new element is added (the *small* hash table is rebuilt more often, but the primary hash table is also rebuilt after any M operations)

Cuckoo hashing

Cuckoo hashing

- In cuckoo hashing we have two hash tables of the same size, each of them more than half empty and each hash table has its hash function (so we have two different hash functions).
- For each element to be added we can compute two positions: one from the first hash table and one from the second. In case of cuckoo hashing, it is guaranteed that an element will be on one of these positions.
- Search is simple, because we only have to look at these two positions.
- Delete is simple, because we only have to look at these two positions and set to empty the one where we find the element.

- When we want to insert a new element we will compute its position in the first hash table. If the position is empty, we will place the element there.
- If the position in the first hash table is not empty, we will kick out the element that is currently there, and place the new element into the first hash table.
- The element that was kicked off, will be placed at its position in the second hash table. If that position is occupied, we will kick off the element from there and place it into its position in the first hash table.
- We repeat the above process until we will get an empty position for an element.
- If we get back to the same location with the same key we have a cycle and we cannot add this element $\Rightarrow$ resize, rehash

Cuckoo hashing - example

- Assume that we have two hash tables, with $m = 11$ positions and the following hash functions:
  - h1(k) = k % 11
  - h2(k) = (k div 11) % 11

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example

- Insert key 20

Cuckoo hashing - example

- Insert key 20
  - h1(20) = 9 - empty position, element added in the first table
- Insert key 50

Cuckoo hashing - example

- Insert key 20
  - h1(20) = 9 - empty position, element added in the first table
- Insert key 50
  - h1(50) = 6 - empty position, element added in the first table

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			20	

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example

- Insert key 53

Cuckoo hashing - example

- Insert key 53
  - h1(53) = 9 - occupied
  - 53 goes in the first hash table, and it sends 20 in the second to position h2(20) = 1

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20									

Cuckoo hashing - example

- Insert key 75

Cuckoo hashing - example

- Insert key 75
  - h1(75) = 9 - occupied
  - 75 goes in the first hash table, and it sends 53 in the second to position h2(53) = 4

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53						

Cuckoo hashing example

- Insert key 100

Cuckoo hashing example

- Insert key 100
  - h1(100) = 1 - empty position
- Insert key 67

Cuckoo hashing example

- Insert key 100
  - h1(100) = 1 - empty position
- Insert key 67
  - h1(67) = 1 - occupied
  - 67 goes in the first hash table, and it sends 100 in the second to position h2(100) = 9

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53					100	

Cuckoo hashing example

- Insert key 105

Cuckoo hashing example

- Insert key 105
  - h1(105) = 6 - occupied
  - 105 goes in the first hash table, and it sends 50 in the second to position h2(50) = 4
  - 50 goes in the second hash table, and it sends 53 to the first one, to position h1(53) = 9
  - 53 goes in the first hash table, and it sends 75 to the second one, to position h2(75) = 6

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			50		75			100	

Cuckoo hashing example

- Insert key 3

Cuckoo hashing example

- Insert key 3
  - h1(3) = 3 - empty position
- Insert key 36

Cuckoo hashing example

- Insert key 3
  - h1(3) = 3 - empty position
- Insert key 36
  - h1(36) = 3 - occupied
  - 36 goes in the first hash table, and it sends 3 in the second to position h2(3) = 0

Position	0	1	2	3	4	5	6	7	8	9	10
T		67		36			105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20			50		75			100	

Cuckoo hashing example

- Insert key 39

Cuckoo hashing example

- Insert key 39
  - h1(39) = 6 - occupied
  - 39 goes in the first hash table and it sends 105 in the second to position h2(105) = 9
  - 105 goes to the second hash table and it sends 100 in the first to position h1(100) = 1
  - 100 goes in the first hash table and it sends 67 in the second to position h2(67) = 6
  - 67 goes in the second hash table and it sends 75 in the first to position h1(75) = 9
  - 75 goes in the first hash table and it sends 53 in the second to position h2(53) = 4
  - 53 goes in the second hash table and it sends 50 in the first to position h1(50) = 6
  - 50 goes in the first hash table and it sends 39 in the second to position h2(39) = 3

Cuckoo hashing example

Position	0	1	2	3	4	5	6	7	8	9	10
T		100		36			50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20		39	53		67			105	

Cuckoo hashing

- It can happen that we cannot insert a key because we get in a cycle. In these situation we have to increase the size of the tables and rehash the elements.
- While in some situation insert moves a lot of elements, it can be shown that if the load factor of the tables is below 0.5, the probability of a cycles is low and it is very unlikely that more than $O(log_2 n)$ elements will be moved.

Cuckoo hashing

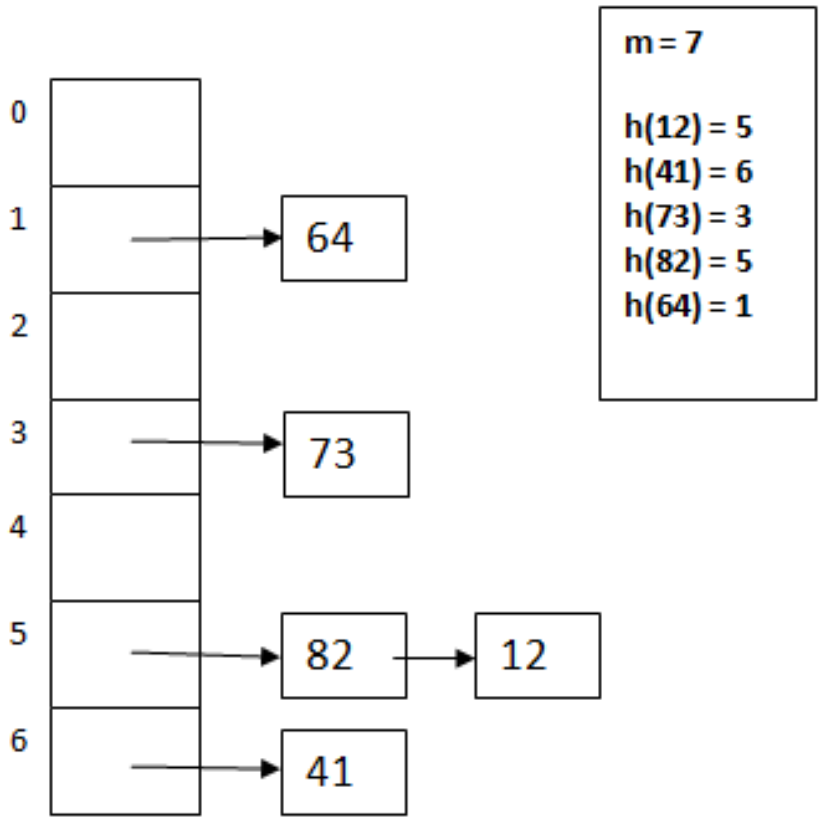
- If we use two tables and each position from a table holds one element at most, the tables have to have load factor below 0.5 to work well.
- If we use three tables, the tables can have load factor of 0.91 and for 4 tables we have 0.97

Linked Hash Table

Linked Hash Table

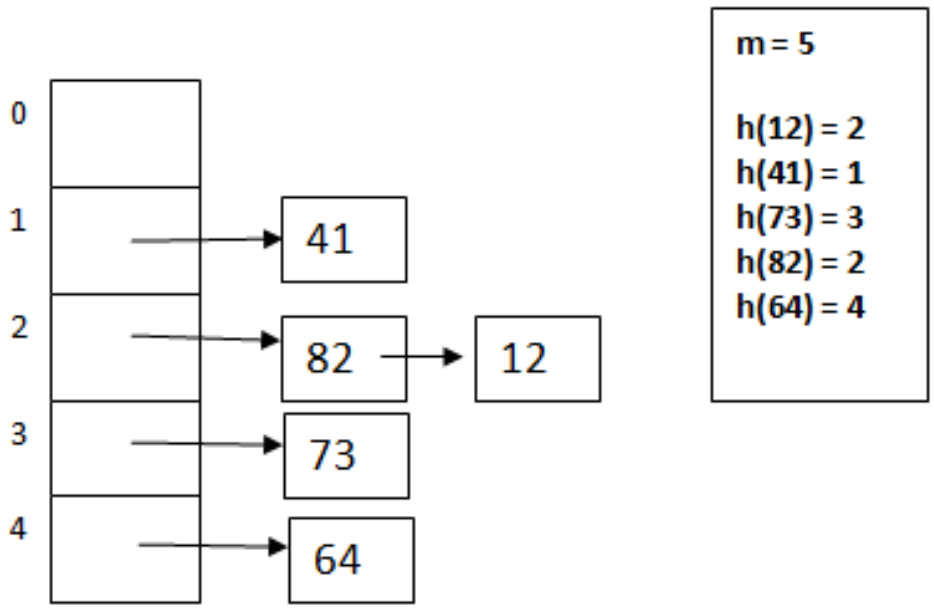
- Assume we build a hash table using separate chaining as a collision resolution method.
- We have discussed how an iterator can be defined for such a hash table.
- When iterating through the elements of a hash table, the order in which the elements are visited is *undefined*
- For example:
  - Assume an initially empty hash table (we do not know its implementation)
  - Insert one-by-one the following elements: 12, 41, 73, 82, 64
  - Use an iterator to display the content of the hash table
  - In what order will the elements be displayed?

Linked Hash Table



- Iteration order: 64, 73, 82, 12, 41

Linked Hash Table



- Iteration order: 41, 82, 12, 73, 64

Linked Hash Table

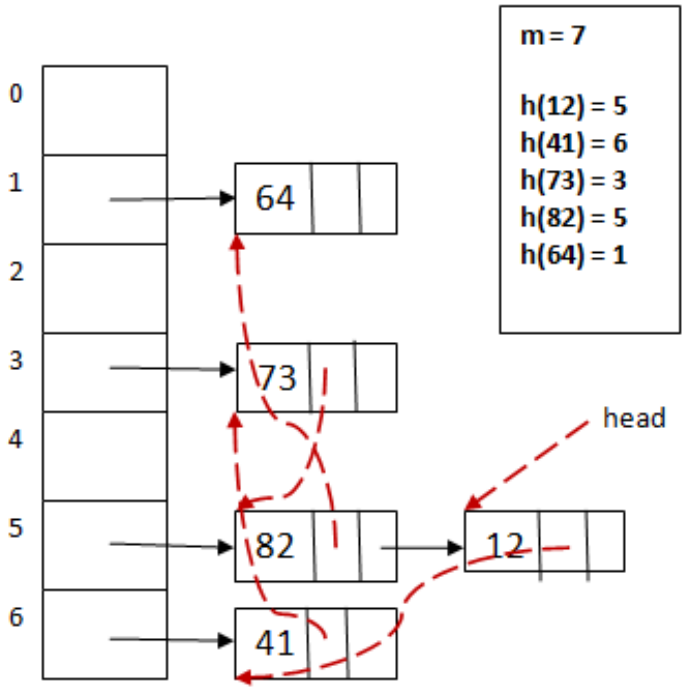
- A *linked hash table* is a data structure which has a *predictable* iteration order. This order is the order in which elements were inserted.
- So if we insert the elements 12, 41, 73, 82, 64 (in this order) in a linked hash table and iterate over the hash table, the iteration order is guaranteed to be: 12, 41, 73, 82, 64.
- How could we implement a linked hash table which provides this iteration order?

Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.

- Since it is still a hash table, we want to have, on average, $\Theta(1)$ for insert, remove and search, these are done in the same way as before, the *extra* linked list is used only for iteration.

Linked Hash Table



- Red arrows show how the elements are linked in insertion order, starting from a *head* - the first element that was inserted, 12.

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

- The only operation that cannot be efficiently implemented if we have a singly linked list is the *remove* operation. When we remove an element from a singly linked list we need the element before it, but finding this in our linked hash table takes $O(n)$ time.

Linked Hash Table - Implementation

- What structures do we need to implement a Linked Hash Table?

Node:
info: TKey
nextH: ↑ Node *//pointer to next node from the collision*
nextL: ↑ Node *//pointer to next node from the insertion-order list*
prevL: ↑ Node *//pointer to prev node from the insertion-order list*

LinkedHT:
m:Integer
T:(↑ Node)[]
h:TFunction
head: ↑ Node
tail: ↑ Node

Linked Hash Table - Insert

- How can we implement the *insert* operation?

subalgorithm insert(lht, k) **is:**
//pre: lht is a LinkedHT, k is a key
//post: k is added into lht
    allocate(newNode)
    [newNode].info ← k
    @set all pointers of newNode to NIL
    pos ← lht.h(k)
    //first insert newNode into the hash table
    if lht.T[pos] = NIL **then**
        lht.T[pos] ← newNode
    else
        [newNode].nextH ← lht.T[pos]
        lht.T[pos] ← newNode
    end-if
//continued on the next slide...

Linked Hash Table - Insert

```
    //now insert newNode to the end of the insertion-order list
    if lht.head = NIL then
        lht.head ← newNode
        lht.tail ← newNode
    else
        [newNode].prevL ← lht.tail
        [lht.tail].nextL ← newNode
        lht.tail ← newNode
    end-if
end-subalgorithm
```

Linked Hash Table - Remove

- How can we implement the *remove* operation?

```
subalgorithm remove(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k was removed from lht
    pos ← lht.h(k)
    current ← lht.T[pos]
    nodeToBeRemoved ← NIL
    //first search for k in the collision list and remove it if found
    if current ≠ NIL and [current].info = k then
        nodeToBeRemoved ← current
        lht.T[pos] ← [current].nextH
    else
        prevNode ← NIL
        while current ≠ NIL and [current].info ≠ k execute
            prevNode ← current
            current ← [current].nextH
        end-while
//continued on the next slide...
```

```
        if current ≠ NIL then
            nodeToBeRemoved ← current
            [prevNode].nextH ← [current].nextH
        else
            @k is not in lht
        end-if
    end-if
//if k was in lht then nodeToBeRemoved is the address of the node containing
//it and the node was already removed from the collision list - we need to
//remove it from the insertion-order list as well
    if nodeToBeRemoved ≠ NIL then
        if nodeToBeRemoved = lht.head then
            if nodeToBeRemoved = lht.tail then
                lht.head ← NIL
                lht.tail ← NIL
            else
                lht.head ← [lht.head].nextL
                [lht.head].prev ← NIL
            end-if
//continued on the next slide...
```

```
        else if nodeToBeRemoved = lht.tail then
            lht.tail ← [lht.tail].prev
            [lht.tail].next ← NIL
        else
            [[nodeToBeRemoved].next].prev ← [nodeToBeRemoved].prev
            [[nodeToBeRemoved].prev].next ← [nodeToBeRemoved].next
        end-if
        deallocate(nodeToBeRemoved)
    end-if
end-subalgorithm
```

Conclusions

- During the semester we have talked about the most important containers (ADT) and their main properties and operations
  - Bag, Set, Map, Multimap, List, Stack, Queue and their sorted versions
- We have also talked about the most important data structures that can be used to implement these containers
  - Dynamic array, Linked lists, Binary heap, Hash table, Binary Search Tree

Conclusions

- You should be able to identify the most suitable container for solving a given problem:

Conclusions

- You should be able to identify the most suitable container for solving a given problem:
- Example: *You have a type Student which has a name and a city. Write a function which takes as input a list of students and prints for each city all the students that are from that city. Each city should be printed only once and in any order.*
- How would you solve the problem? What container would you use?

Conclusions

- When you use containers existing in different programming languages, you should have an idea of how they are implemented and what is the complexity of their operations:

Conclusions

- When you use containers existing in different programming languages, you should have an idea of how they are implemented and what is the complexity of their operations:
- Consider the following algorithm (written in Python):

```
def testContainer(container, l):
 '''
container is a container with integer numbers
l is a list with integer numbers
'''
    count = 0
    for elem in l:
        if elem in container:
            count += 1
    return count
```

- The above function counts how many elements from the list *l* can be found in the container. What is the complexity of *testContainer*?

Conclusions

- Consider the following problem: *We want to model the content of a wallet, by using a list of integer numbers, in which every value denotes a bill. For example, a list with values [5, 1, 50, 1, 5] means that we have 62 RON in our wallet.*
  Obviously, we are not allowed to have any numbers in our list, only numbers corresponding to actual bills (we cannot have a value of 8 in the list, because there is no 8 RON bill).
  We need to implement a functionality to pay a given amount of sum and to receive rest of necessary.
  There are many optimal algorithms for this, but we go for a very simple (and non-optimal): keep removing bills of the wallet until the sum of removed bills is greater than or equal to the sum you want to pay.
  If we need to receive a rest, we will receive it in 1 RON bills.

Conclusions

- For example, if the wallet contains the values [5, 1, 50, 1, 5] and we need to pay 43 RON, we might remove the first 3 bills (a total of 56) and receive the 13 RON rest in 13 bills of 1.

Conclusions

- For example, if the wallet contains the values [5, 1, 50, 1, 5] and we need to pay 43 RON, we might remove the first 3 bills (a total of 56) and receive the 13 RON rest in 13 bills of 1.
- This is an implementation provided by a student. What is wrong with it?

```
public void spendMoney(ArrayList<Integer> wallet, Integer amount) {
    Integer spent = 0;
    while (spent < amount) {
        Integer bill = wallet.remove(0); //removes element from position 0
        spent += bill;
    }
    Integer rest = spent - amount;
    while (rest > 0) {
        wallet.add(0, 1);
        rest--;
    }
}
```