Computer Science 3A - CSC3A10/CSC03A3

Lecture 2: Arrays, Linked Lists and Recursion

Academy of Computer Science and Software Engineering
University of Johannesburg

- Binary Recursion
- Multiple Recursion
- Exercises

Arrays

Arrays

Storing in an array

- Object to store (such as GameEntry)
- High score class
- Insertion
- Removal

Arrays II

Sorting an array

- Insertion sort
- Algorithm (CF 3.5)

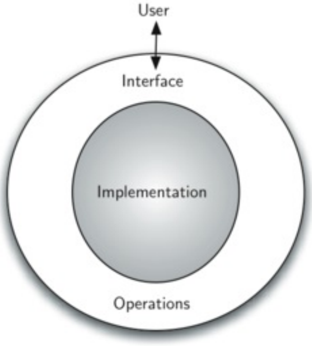
Java.util

- equals(A,B)
- fill(A,x)
- sort(A)
- toString(A)

Arrays Implemented

- Pseudo-Random Number (java.util.Random and seed)
- Cryptography (Caesar Cipher)
- 2D Arrays (Matrix and Positional games, such as Tic-Tac-Toe)

Basic ADT's



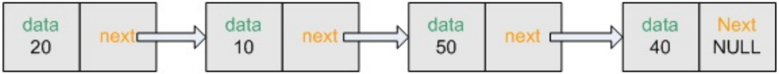
Postion ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Just one method:
  - object element(): returns the element stored at the position

List ADT

- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size()
  - isEmpty()
- Accessor methods:
  - first()
  - last()
  - prev(p)
  - next(p)
- Update methods:
  - replace(p, e)
  - insertBefore(p, e),
  - insertAfter(p, e),
  - insertFirst(e)
  - insertLast(e)
  - remove(p)

Singly Linked Lists



Linked list

Singly Linked List Properties

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

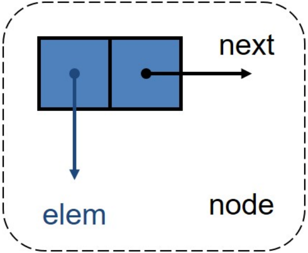


Figure: An individual node in a Singly Linked List

Singly Linked List II

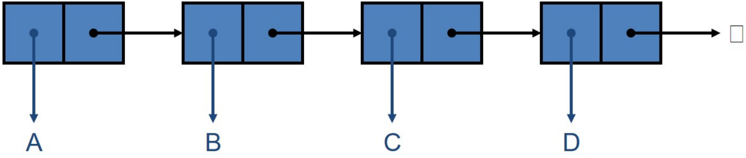


Figure: Singly Linked List with multiple nodes

Singly Linked List Node Class I

```
public class Node {
    // Instance variables:
    private Object element;
    private Node next;
    /** Creates a node with null references to its element and next
        node. */
    public Node()    {
        this(null, null);
    }
    /** Creates a node with the given element and next node. */
    public Node(Object e, Node n) {
        element = e;
        next = n;
    }
    // Accessor methods:
    public Object getElement() {
```

Singly Linked List Node Class II

```
        return element;
    }
    public Node getNext() {
        return next;
    }
    // Modifier methods:
    public void setElement(Object newElem) {
        element = newElem;
    }
    public void setNext(Node newNext) {
        next = newNext;
    }
}
```

Inserting at the Head

1. Allocate new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node

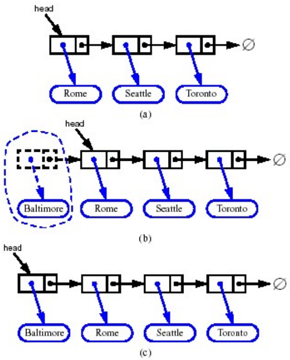


Figure: Insertion at head

Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

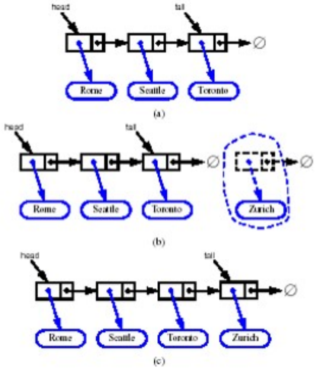


Figure: Insertion at tail

Removal at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

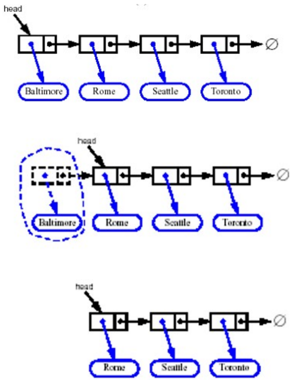


Figure: Removal at head

Removal at the Tail

1. Removing at the tail of a singly linked list is not efficient!
2. There is no constant-time way to update the tail to point to the previous node

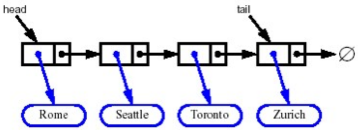


Figure: Removal at tail

Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time

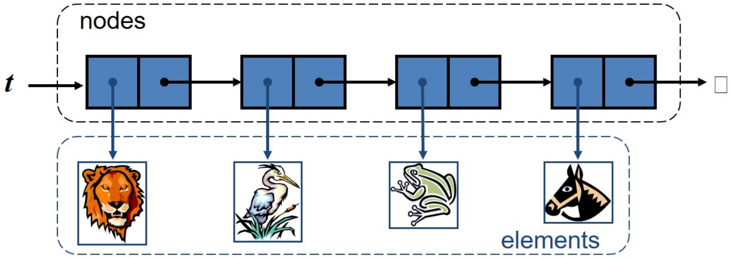


Figure: A visualization of a stack implemented using a SLL

Queue with a Singly Linked List

- We can implement a queue with a singly linked list
- The front element is stored at the first node
- The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time

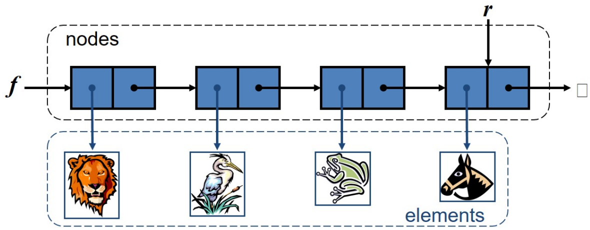


Figure: A visualization of a queue implemented using a SLL

Singly Linked List Implementation

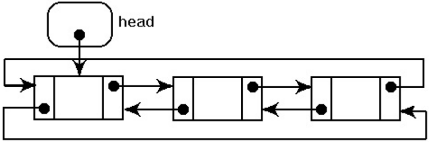
- Implementation (Node class - CF 3.14 and SLinkedList Class - CF 3.15)
- Insertion (Head and Tail - CF 3.15, *but what about the middle?*
- Removal (Head - CF 3.15, *but what about the tail and middle?*)

Singly Linked List Performance

In the implementation of the List ADT by means of a singly linked list

- The space used by a list with n elements is $O(n)$
- The space used by each position of the list is $O(1)$
- **Most** of the operations of the List ADT run in $O(1)$ time (except access methods)
- Operation element() of the Position ADT runs in $O(1)$ time

Circular Linked Lists



Doubly Linked Circular list

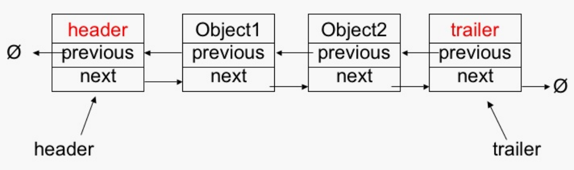
Circular Linked Lists

- Always a next pointer
- No null to signify end of list.
- No head, no tail
- Special node known as the cursor

Circular Linked Lists II

- Some methods
  - add
  - remove
  - advance [the cursor]
- Sorting (definitely easier on a doubly linked list!)
  - insertion sort (CF 3.5*)
  - Java (CF 3.6*)

Doubly Linked Lists



Doubly Linked List Properties

- Singly Linked Lists: No way of going (quickly) to predecessor node
- Doubly Linked List Nodes have both Next and Previous references (CF 3.17)
- They also contain Sentinel Nodes namely the header and trailer, which are empty! (Fig 3.19)

Doubly Linked List Properties II

- Now removing the tail?
- Adding at the head (in fact, add after the header)
- Insertion in the middle? (Easy: for any node v we call insertAfter(v, z) - z being the node to insert)
- Removal in the middle? (CF 3.18)

Doubly Linked Structure

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

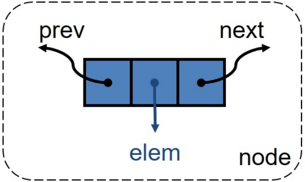


Figure: An individual node in a Doubly Linked List

Doubly Linked List II

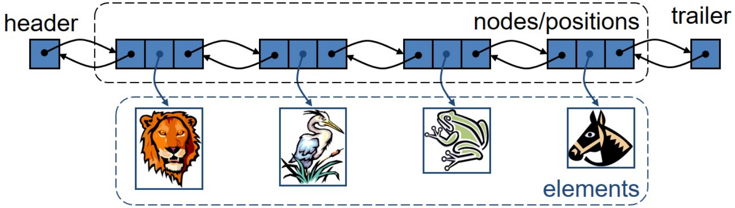


Figure: Doubly Linked List with multiple nodes

Doubly Linked List Insertion

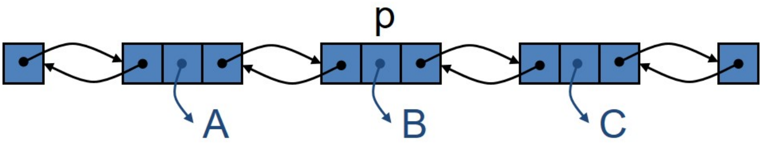


Figure: Doubly Linked List before insertion

Doubly Linked List Insertion II

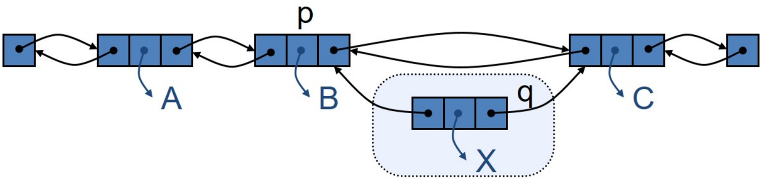


Figure: Doubly Linked List during insertion

Doubly Linked List Insertion III

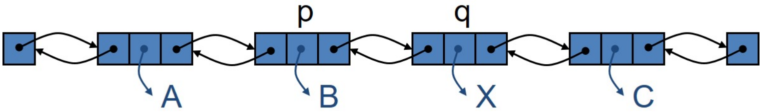


Figure: Doubly Linked List after insertion

Doubly Linked List Insertion IV

```
Algorithm insertAfter(p,e):
  Create a new node v
  v.setElement(e)
  v.setPrev(p)  {link v to its predecessor}
  v.setNext(p.getNext())  {link v to its successor}
  (p.getNext()).setPrev(v)  {link p's old successor to v}
  p.setNext(v)  {link p to its new successor, v}
  return v  {the position for the element e}
```

insertAfter algorithm

Doubly Linked List Removal

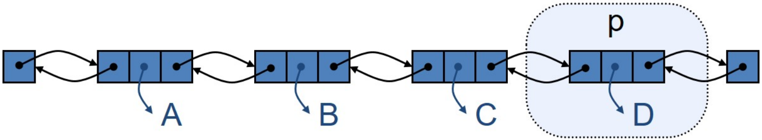


Figure: Doubly Linked List before removal

Doubly Linked List Removal II

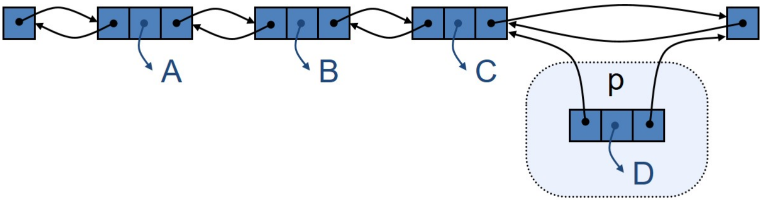


Figure: Doubly Linked List during removal

Doubly Linked List Removal III

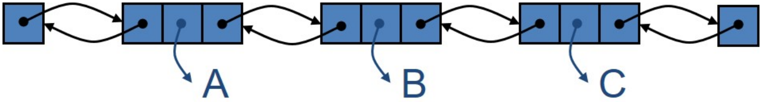


Figure: Doubly Linked List after removal

Doubly Linked List Removal IV

```
Algorithm remove(p):
  t = p.element {a temporary variable to hold the return value}
  (p.getPrev()).setNext(p.getNext())  {linking out p}
  (p.getNext()).setPrev(p.getPrev())
  p.setPrev(null) {invalidating the position p}
  p.setNext(null)
  return t
```

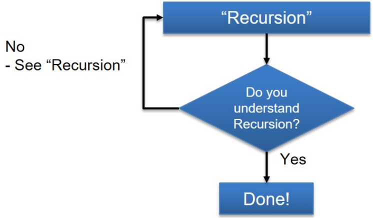
remove algorithm

Doubly Linked List Performance

In the implementation of the List ADT by means of a doubly linked list

- The space used by a list with n elements is $O(n)$
- The space used by each position of the list is $O(1)$
- **All** the operations of the List ADT run in $O(1)$ time
- Operation element() of the Position ADT runs in $O(1)$ time

Recursion


"Recursion"
No
- See "Recursion"
Do you understand Recursion?
Yes
Done!

Recursion Properties

- When a method calls itself
- Sometimes the solution has repetition

Recursion Examples

Classic recursive examples include:

- Euclid's algorithm
- The factorial function
- The Fibonacci sequence
- The Ackermann function
- Towers of Hanoi
- Others

Every recursive function can be written in an iterative manner!

Recursion Examples II

Euclid's Algorithm (Greatest Common Divisor):

$$\text{For } a, b \geq 0, gcd(a, b) = \begin{cases} a & \text{if } b = 0, \\ gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

```java
public static int gcd ( int a , int b )
{
    if ( b == 0 )
        return a;
    else if ( a >= b && b > 0)
        return gcd ( b , a % b );
    else return gcd ( b , a );
}
```

Euclid's Java function

Recursion Examples III

The factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

$$f(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot f(n-1) & \text{else} \end{cases}$$

```java
public static int recursiveFactorial(int n) {
  // basis case
  if (n == 0) return 1;
  // recursive case
  else return n * recursiveFactorial(n- 1);
}
```

Factorial Java function

Content of a Recursive Method

Base case(s):

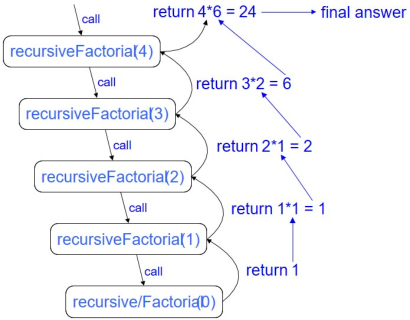
- Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
- Every possible chain of recursive calls must eventually reach a base case.

Recursive calls:

- Calls to the current method.
- Each recursive call should be defined so that it makes progress towards a base case.

Visualizing recursion

- Recursion trace
- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value



Types of Recursion

- Linear recursion
- Tail recursion
- Binary recursion
- Multiple recursion

Linear Recursion

Test for base cases

- Begin by testing for a set of base cases (there should be at least one).
- Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.

Recur once:

- Perform a single recursive call. (This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.)
- Define each possible recursive call so that it makes progress towards a base case.

Linear Recursion II

```
Algorithm LinearSum(A, n):
Input: A integer array A and an integer n = 1, such that A has at least
     n elements
Output: The sum of the first n integers in A
if n = 1 then
  return A[0]
else
  return LinearSum(A, n − 1) + A[n − 1]
```

Linear Summation Algorithm

Linear Recursion III

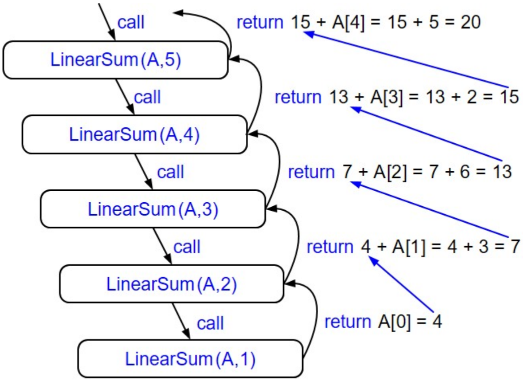


Figure: Recursive Linear Summation Visualized

Linear Recursion IV

```
Algorithm ReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and
    ending at j
if i < j then
  Swap A[i] and A[j]
  ReverseArray(A, i + 1, j − 1)
return
```

Reverse Algorithm

Linear Recursion V

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).

Tail Recursion II

For example:

```
Algorithm IterativeReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and
    ending at j

  while i <  j do
    Swap A[i] and A[j]
    i  = i + 1
    j  = j − 1
  return
```

Reverse Algorithm

Binary Recursion

Problem: add all the numbers in an integer array A:

```
Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
  if n = 1 then
    return A[i]
  return BinarySum(A, i, n/ 2) + BinarySum(A, i + n/ 2, n/ 2)
```

Binary Sum

Binary Recursion II

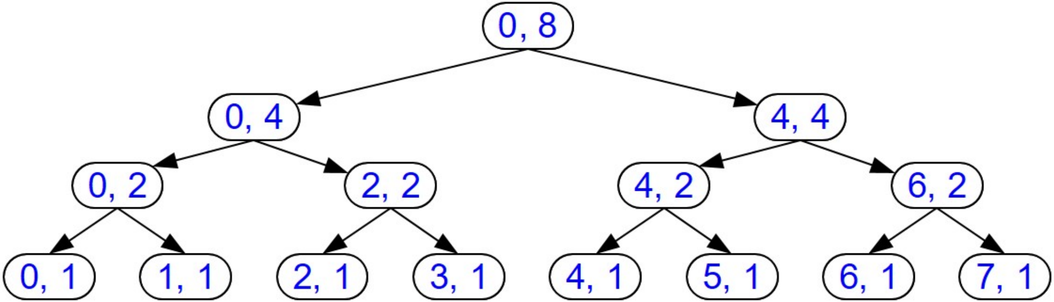


Figure: Example trace for binary sum

Binary Recursion III

The Fibonacci Algorithm via Binary recursion

$$f(n) = \begin{cases} F_0 = 1 & (or\ F_0 = 0) \\ F_1 = 1 & \\ F_i = F_{i-1} + F_{i-2} & for\ i > 1 \end{cases}$$

Binary Recursion IV

As a recursive algorithm (first attempt):

```
Algorithm BinaryFib(k):
Input: Nonnegative integer k
Output: The kth Fibonacci number Fk
  if k = 0 || k = 1 then
    return k
  else
    return BinaryFib(k − 1) + BinaryFib(k − 2)
```

Binary Fibonacci

Binary Recursion V

Let n_k denote number of recursive calls made by *BinaryFib(k)*. Then

```
n0 = 1 (sometimes there is a n00 = 0)
n1 = 1
n2 = n1 + n0 = 2
n3 = n2 + n1 = 3
n4 = n3 + n2 = 2 + 3 = 5
n5 = n4 + n3 = 3 + 5 = 8
n6 = n5 + n4 = 5 + 8 = 13
n7 = n6 + n5 = 8 + 13 = 21
n8 = n7 + n6 = 13 + 21 = 34.
```

Note that the value at least doubles for every other value of n_k. That is, $n_k > 2^{k/2}$. It is exponential!

Binary Recursion VI

Use linear recursion instead:

```
Algorithm LinearFibonacci(k):
Input: A nonnegative integer k
Output: Pair of Fibonacci numbers (Fk, Fk−1)
  if k = 1 then
    return (k, 0)
  else
    (i, j) = LinearFibonacci(k − 1)
    return (i +j, i)
```

Linear Fibonacci

Runs in O(k) time!

Multiple Recursion

Motivating example: summation puzzles

```
pot + pan = bib
dog + cat = pig
boy + girl = baby
```

Multiple recursion: makes potentially many recursive calls (not just one or two).

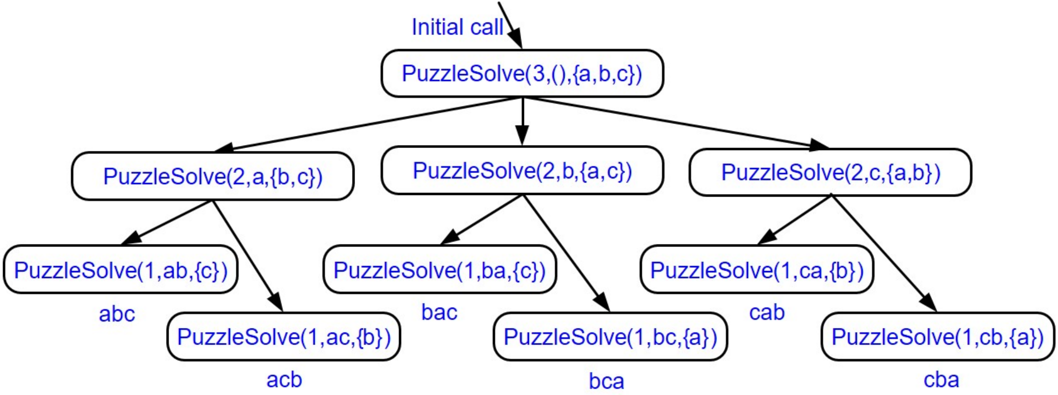
Multiple Recursion II

```
Algorithm PuzzleSolve(k,S,U):
Input: An integer k, sequence S, and set U (the elements used to test)
Output:  An enumeration of all k-length extensions to S using elements
    in U
without repetitions

for all e in U do
  Remove e from U   {e is now being used}
  Add e to the end of S
  if k = 1 then
    Test whether S is a configuration that solves the puzzle
    if S solves the puzzle then
      return ''Solution found: '' S
  else
    PuzzleSolve(k - 1, S,U)
  Add e back to U   {e is now unused}
  Remove e from the end of S
```

Multiple Recursion II

Example trace for PuzzleSolve(3,S,U) where S=() and U={a,b,c}



Reinforcement exercises: (U.S. version in parenthesis)

- R-3.1
- R-3.4
- R-3.7

Creativity exercises:

- C-3.17
- C-3.20
- C-3.23
- C-3.24
- C-3.25
- C-3.32
- C-3.33