Dynamic Arrays

Dov Kruger

Department of Electrical and Computer Engineering
Rutgers University

January 23, 2024

©Dov Kruger 2023

2.1 Introduction

A list is the single most used data structure, and the most common design pattern in the world

Most computer programs contain thousands or hundreds of thousands of lists
And the average size? Between 0 and 1.

Yes, that's right. *Most lists are either empty or have one element*

A **Dynamic array** is an object with a **single block of memory** to hold a list of values that can grow and shrink

- With the right design, it can be very efficient
- A common mistake yields disastrously bad performance

©Dov Kruger 2023

List Operations

What are the principle operations in creating a list?

- Adding an element to the list?
- Finding an element in a list?
- Replace?

It turns out that not all the above operations are fundamental

You have to drill down

©Dov Kruger 2023

Fundamental Operations for Lists

- Create a list
- Add a value to the end of the list
- Add a value to the start of the list
- Add a value at a specific position
- Get the size of the list
- Remove an element from the end of the list
- Remove an element from the start of the list
- Remove an element at a specific position
- Get the value at a particular position
- Set the value at a particular position

©Dov Kruger 2023

Fundamental Operations in Java

```
List a = new List(); // create an empty list object
a.addEnd(5);
a.addStart(1);
a.insertAfter(0, 3); // insert value 3 after position 0
int sz = a.size();    // number of elements in the list

a.removeEnd();
a.removeStart();
a.remove(index);

int v = a.get(index)
a.set(index, v)
```

©Dov Kruger 2023

Fundamental Operations in C++

C++ is similar to Java but objects are declared using different syntax

```
DynamicArray a; // create an empty list
a.addEnd(5); // add 5 to the end of the list
```

Be aware: by default, it's easy to crash in C++ if you do not understand copy semantics

©Dov Kruger 2023

Memory Management in C++ and Copy Semantics

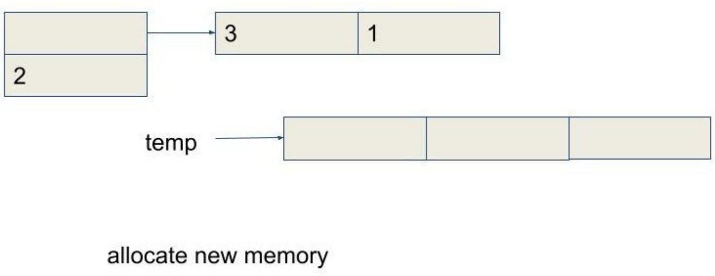
In C++ by default certain constructors are written. Not understanding how they work can result in crashing. If you write in C++ make sure you understand!

```
class DynamicArray {
public:
  DynamicArray(); // default constructor
  ~DynamicArray(); // destructor
  DynamicArray(const DynamicArray& orig); // copy constructor
  DynamicArray& operator =(const DynamicArray& orig);

  //optional: move constructor
  DynamicArray(DynamicArray&& orig);
};
```

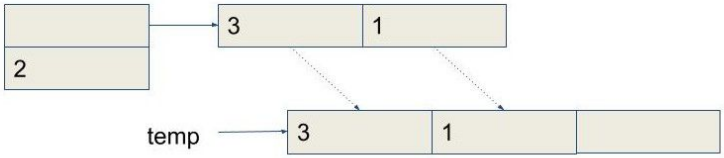
Operations: addEnd

What does it mean to add a number to a list?



©Dov Kruger 2023

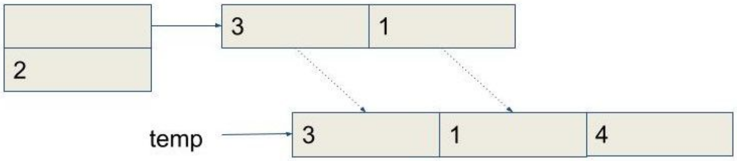
Operations: addEnd



copy in the current values

©Dov Kruger 2023

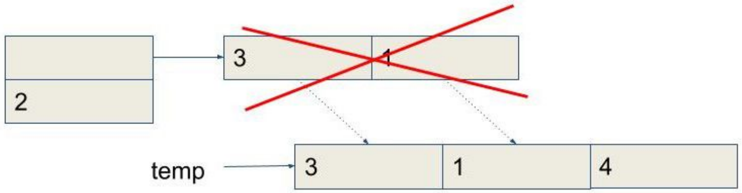
Operations: addEnd



add the new value

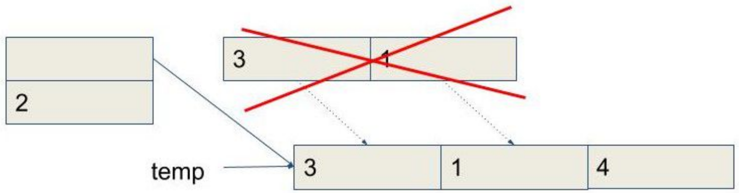
©Dov Kruger 2023

Operations: addEnd



delete old memory (in Java, this happens automatically)

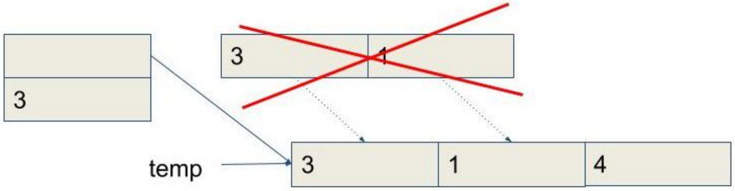
Operations: addEnd



update pointer to new memory

©Dov Kruger 2023

Operations: addEnd

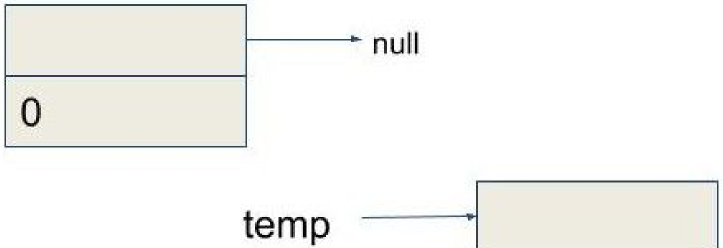


update size of list

©Dov Kruger 2023

Is There a Special Case for the First time?

The first time, the existing pointer is null, but the size=0, nothing is copied



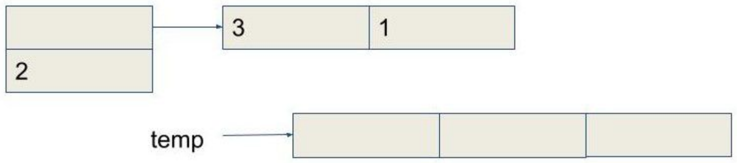
Implementation in C++

Analyze this code and determine the complexity of addEnd

```
void addEnd(int v) {      //O(?)
  const int* temp = data; //keep track of old memory
  data = new int[size+1]; //allocate new block one bigger
  for (int i = 0; i < size; i++)
    data[i] = old[i]; // copy old data to new
  data[size] = v; // add in new value
  delete [] old;
  size++; // add one to size of list
}
```

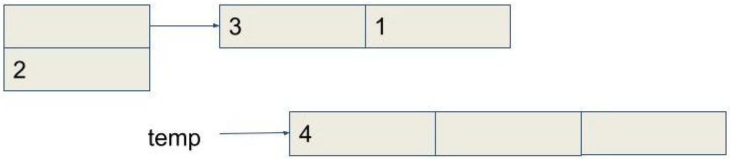
©Dov Kruger 2023

Operations: addStart(4)



allocate new memory

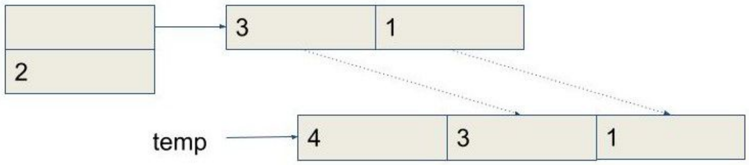
Operations: addStart(4)



set first value

©Dov Kruger 2023

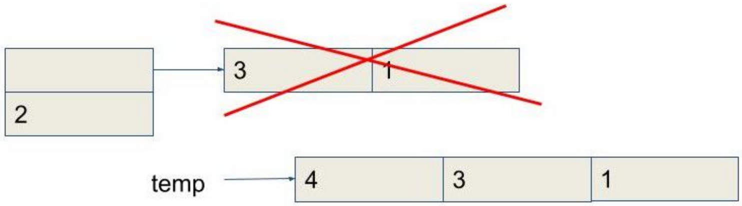
Operations: addStart(4)



copy values

©Dov Kruger 2023

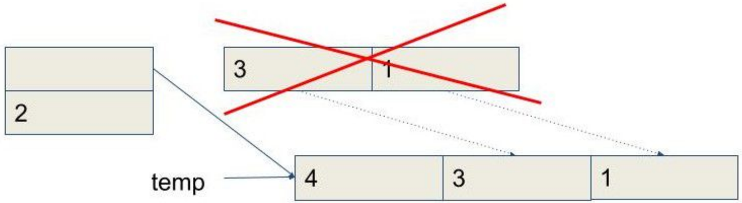
Operations: addStart(4)



delete memory (C++ only)

©Dov Kruger 2023

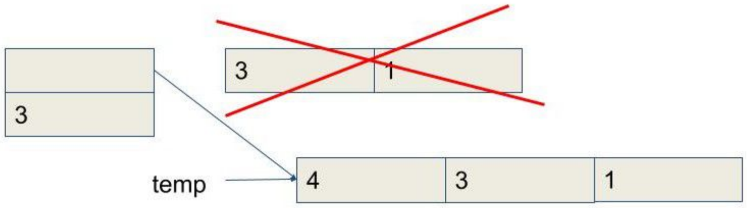
Operations: addStart(4)



point to new memory

©Dov Kruger 2023

Operations: addStart(4)



update size

©Dov Kruger 2023

Complexity of Adding Elements

First time: 1

Second time: allocate new memory, copy the existing (1) and add $1 = 2$

Third time: copy existing (2) and add $1 = 3$

$$1 + 2 + 3 + \ \dots \ = n(n + 1)/2 = O(n^2)$$

©Dov Kruger 2023

Implementation Complexity in C++ and Java

Java

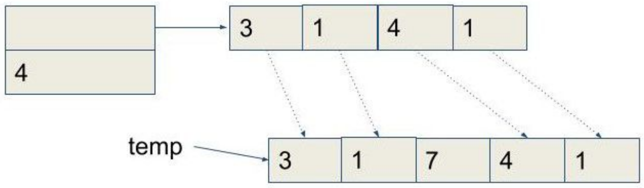
- Allocation copies zeros into every byte and is therefore O(n)
- Not necessary to explicitly delete the memory
- The garbage collector deallocates
- It still takes time, which must be accounted for in your algorithm

C++

- Allocation writes nothing. Memory is random. Therefore O(1)
- Presumably you have to correctly initialize, so O(n) but Java can be **twice as slow**
- You must manually delete the memory or you leak (task grows until you crash)
- If you want to know how to catch leaks, see videos on valgrind and -fsanitize

©Dov Kruger 2023

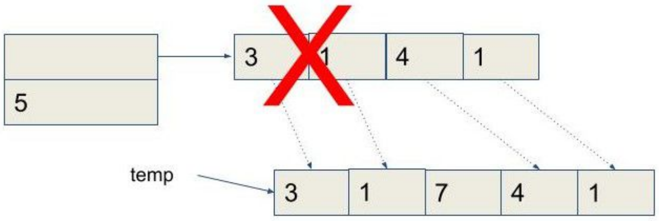
Operations: insert(2, 7)



allocate, copy values before, add the 7, copy values after

©Dov Kruger 2023

Operations: insert(2, 7). contd.



increase size, delete old memory, point to new memory

Operations: removeEnd

©Dov Kruger 2023

Operations: removeStart

©Dov Kruger 2023

Operations: remove()

©Dov Kruger 2023

Get size of list

Getting the size of the list is O(1)

The size of the DynArray is stored inside

©Dov Kruger 2023

Get/Set an Element

©Dov Kruger 2023

Complexity Problems

The DynamicArray as shown is poor because

- Growing or shrinking a list by n elements will take $O(n^2)$ time
- Total RAM copied will also be $O(n^2)$ although it can be recycled
- Very, very slow

Needed: Build a new, better DynamicArray which can grow more efficiently

- Preallocate more space than needed
- addEnd is $O(1)$ until space is all full
- Growing the list will require $O(n)$ time, where n is the list size
- What is the total cost of this?

©Dov Kruger 2023

Growing an Array

For DynamicArray, cost is
$1 + 2 + 3 + \ ... \ + n = n(n + 1)/2 = O(n^2)$

- Suppose instead of growing every time we grow every 2 times? $1 + 2 + 1 + 4 + 1 + 5 + \ ... \ n =$ still $O(n^2)$ though a lower constant factor

The way to fundamentally improve: double the size each time

This means grow will happen infrequently:
$1 + 2 + 4 + 8 + 16 + \ ... \ + n/2 + n$
Recall: sum of the powers of 2 = O(2n) = O(n)

Q: How many times will the list grow to reach n=1 million (10^6) elements?

©Dov Kruger 2023

5.2 Practical Problems Where Knowledge is Crucial

What you have learned about dynamic arrays will help you understand common performance problems

In Java, String is immutable (cannot be changed)

Therefore the only way to grow a string is to create a new one.

This is exactly like the first BadDynamicArray example we analyzed

```
public class MyCode {
  public static void main(String[] args) {
    String s = "";
    final int n = 1000000; // what is wrong with this code?
    for (int i = 0; i < n; i++)
      s += i; //"1", "12", "123", "1234", ...
    System.out.println(s);
  }
```

©Dov Kruger 2023

Java: StringBuilder to the Rescue

The problem with String on the previous slide is that it cannot grow. Each time

- A new String object must be created
- The old one must be copied
- StringBuilder solves this problem

```
StringBuilder b = new StringBuilder();
final int n = 1000000;
for (int i = 0; i < n; i++)
  b.append(i);
```

Knowing that the StringBuilder must grow, speed the code further:

```
StringBuilder b = new StringBuilder(n * 6);
```

Practical Example: MATLAB

MATLAB also can easily have the same problem as Java

This will never complete

```
a = [];
for i=1:1000000
   a = [a i];
end
```

©Dov Kruger 2023

Example: Dynamic Arrays Using C++ Library

In the C++ Library, vector

```
#include <vector>
int main() {
  vector<int> a;
  const int n = 1024;
  a.reserve(n); // preallocate space for efficiency
  for (int i = 0; i < n; i++)
    a.push_back(i);
}
```

©Dov Kruger 2023

Example: Dynamic Arrays Using Java Library

```
import java.util.*;

public class TestDynamicArray {
  public static void main(String[] args) {
    ArrayList<Integer> mylist = new ArrayList<>();
  }
}
```

©Dov Kruger 2023

Example: Dynamic Arrays Using C++ Library

```
#include <vector>

int main() {
   vector<int> mylist;
   for (int i = 0; i < n; i++)
      mylist.push_back(i);
}
```

©Dov Kruger 2023