csci 210: Data Structures

Lists and Iterators

Summary

- Topics
  - Java
    - Vector, ArrayList, Stack, LinkedList, Collections
    - extendable arrays
      - analysis
  - Iterators

- Reading:
  - Collections: LC chapter 3

ArrayLists and Vectors

- classes provided by Java
  - Java.util.ArrayList
  - Java.util.Vector

- practically identical

- provide support for “smart” arrays
  - allow variable size of array
  - support useful methods
    - get(i)
    - set(i,e)
    - add(i,e)
    - remove(i)
    - add(e)
    - size()
    - isEmpty()
- Exercise: implementation
- Notation
  - N is the maximum capacity of the array
  - n is the current size

Performance

- Performance
  - get(i) : O(1)
  - set(i,e): O(1)
  - add(i,e): O(n)
  - remove(i): O(n)
  - size(): O(1)
  - isEmpty(): O(1)
  - add(e): O(1) unless overflow

- ArrayLists and Vectors also grow the array
  - whenever add(e) occurs and the array is full, the array is re-allocated of double size
  - let's say N is the current max capacity of the array A
  - allocate B[] of size 2N
  - copy A[i] into B[i] for all i
  - [free the space of A: note: this does not happen in Java, the garbage collector will find out that the space of A is not in use anymore and will free it]
  - A = B
  - add e to A as usual

Analysis of extendable arrays

- Question: How long does add(e) take?
  - O(1) if the array does not grow
  - O(n) if the arrays grows (need to copy all elements of A to B)

- Suppose you start with an empty array of size 1, and you add n elements. How long will this take?
  - O(n^2) ?
  - 1 + 2 + 3 + 4 + ... +n ?

- Lemma:
  - A sequence of n add() operations on an initially empty array that grows by doubling takes O(n) time total.

- Intuition:
  - some add() need to relocate and are slow, but many are O(1)
  - reallocations are not that frequent
  - once the array is reallocated, it is half empty so the next bunch of add() are O(1)

Analysis of extendable arrays

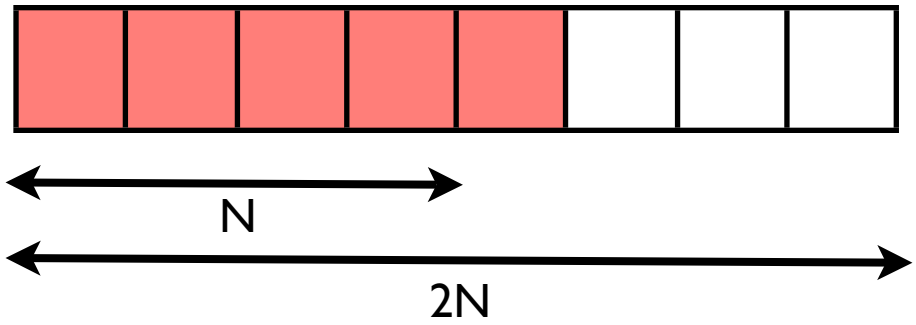
- assume initial capacity of A is 1 and A is empty
- add(e)

		max capacity	cost of copy	cost of add
1	add()	1	-	O(1)
2	add()	2	1	O(1)
3	add()	4	2	O(1)
4	add()		-	O(1)
5	add()	8	4	O(1)
6	add()		-	O(1)
7	add()		-	O(1)
8	add()		-	O(1)
9	add()	16	8	O(1)
...	add()		-	O(1)
17	add()	32	16	O(1)

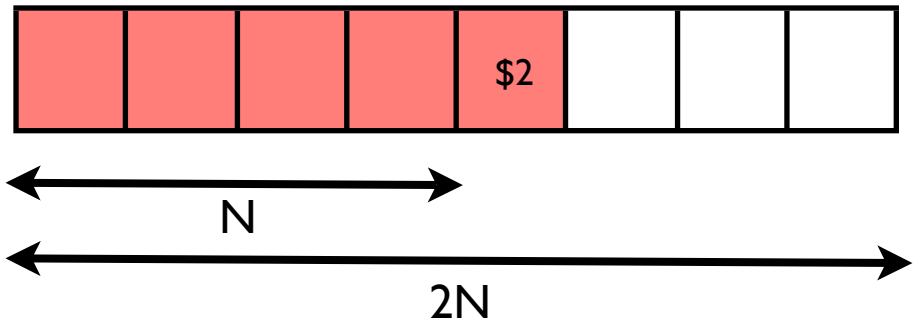
Analysis of extendable arrays

- Imagine you charge each add() $3
  - you use $1 to pay for the actual add()
  - you leave $2 as credit on the element
- We shall prove that the doubling can be paid for by credits accumulated in between doublings.

- Imagine you just doubled the array

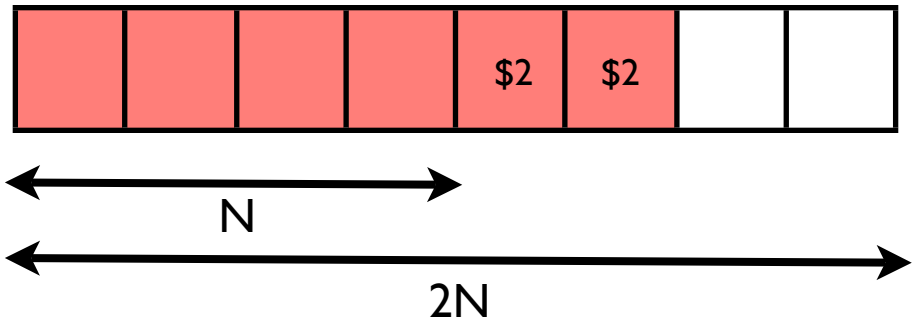


- and you charged this last add() that caused the doubling $3, so you have $2 left

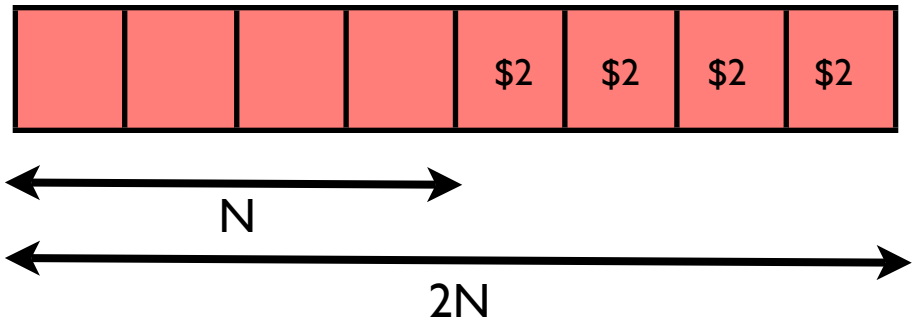


Analysis of extendable arrays

- the next add() : no overflow, O(1)



- ...



- the array gets full again after N add()
  - total credit accumulated: N x $2 = 2N
  - cost of copying the array: 2N

Iterators

- An iterator abstracts the process of scanning through a collection of elements one at a time
- An iterator is a class with the following interface
  - `boolean hasNext()`
    - return true if there are elements left in the iterator
  - `Type  next()`
    - return the next element in the iterator

Iterators in Java

- Java.util.Iterator interface
- Classes that implement collections of elements also support the following method()
  - `iterator()`
    - return an iterator of the elements in the collection

- Example

```
ArrayList<Type> a;
//Vector<Type> a;
//Stack<Type> a;
//LinkedList<Type> a;

Iterator<Type> it = a.iterator();
while (it.hasNext()) {
    Type e = it.next();
    //process e
    //...
}
//or
for (Iterator<Type> it = a.iterator(); it.hasNext();) {
    Type e = it.next();
    //...
}
```

Iterators in Java

- a Java specific for loop that uses iterators (under the hood)

```
Vector<Type> v;
for (Type x: v) {
    //x is the current element  in v and the loop iterates
    //through all elements of v
    System.out.print("the current element is " + x);
}
```

List iterators

- The preferred way to access a Java.util.LinkedList is through an iterator

	lastIndexOf(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this elem
ListIterator	**listIterator**(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
Object	**remove**(int index)

listIterator

```
public ListIterator listIterator(int index)
```

Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. Obeys the general contract of `List.listIterator(int)`.

The list-iterator is *fail-fast*: if the list is structurally modified at any time after the Iterator is created, in any way except through the list-iterator's own `remove` or `add` methods, the list-iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Specified by:
`listIterator` in interface `List`
Specified by:
`listIterator` in class `AbstractSequentialList`

Parameters:
`index` - index of first element to be returned from the list-iterator (by a call to `next`).
Returns:
a ListIterator of the elements in this list (in proper sequence), starting at the specified position in the list.
Throws:
`IndexOutOfBoundsException` - if index is out of range (`index < 0 || index > size()`).
See Also:
`List.listIterator(int)`

- a ListIterator includes

Method Summary	
void	**add**(Object o) Inserts the specified element into the list (optional operation).
boolean	**hasNext**() Returns `true` if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious**() Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
Object	**next**() Returns the next element in the list.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to `next`.
Object	**previous**() Returns the previous element in the list.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to `previous`.
void	**remove**() Removes from the list the last element that was returned by `next` or `previous` (optional operation).
void	**set**(Object o) Replaces the last element returned by `next` or `previous` with the specified element (optional operation).

Iterators

- Why use iterators?
  - More generic code
    - you can change the data structure, and the loop remains the same