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 array 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?
  - $O(n^2)$?
  - $1 + 2 + 3 + 4 + \ldots + n$?

- 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 $O(1)$

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

<table>
<thead>
<tr>
<th>max capacity</th>
<th>cost of copy</th>
<th>cost of add</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>2</td>
<td>add()</td>
<td>$2 \times O(1)$</td>
</tr>
<tr>
<td>4</td>
<td>add()</td>
<td>$4 \times O(1)$</td>
</tr>
<tr>
<td>8</td>
<td>add()</td>
<td>$8 \times O(1)$</td>
</tr>
<tr>
<td>16</td>
<td>add()</td>
<td>$16 \times O(1)$</td>
</tr>
<tr>
<td>32</td>
<td>add()</td>
<td>$32 \times O(1)$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Imagine you charge each add() $3$
  - You use $S1$ to pay for the actual add()
  - You leave $S2$ 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 $S3$, so you have $S2$ left

- The array gets full again after $N$ add()
  - Total credit accumulated: $N \times S2 = 2N$
  - 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
  ```java
  ArrayList<Type> a;
  Iterator<Type> it = a.iterator();
  while (it.hasNext()) {
    Type e = it.next();
    //process e
    //...
  }
  ```

List iterators

- The preferred way to access a Java.util.LinkedList is through an iterator
  ```java
  for (Type x: v) {
    System.out.print("the current element is " + x);
  }
  ```
• 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