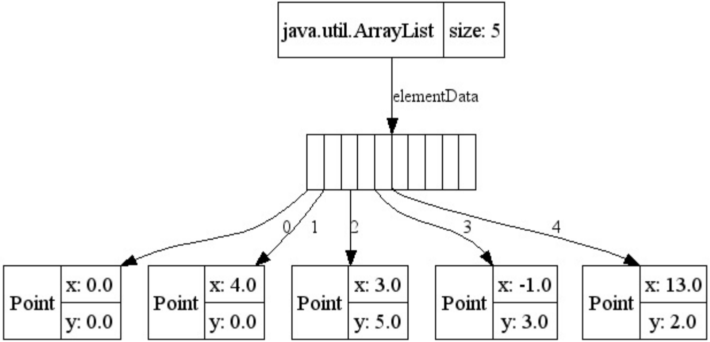
Computer Science 3A - CSC3A10

Lecture 5: Lists and Iterators

Academy of Computer Science and Software Engineering
University of Johannesburg

5 Sequence ADT
- Sequence Properties
- Applications of Sequences
- Sequence Implementations
- Sequence Examples

Array List ADT



Array List Properties

- Collection S of n elements with a certain linear order.
- Also called a list, or sequence.
- We refer to each element e in S through it's index.
- The index of e in S, is the number of elements that are before e in S.
- Index vs Rank.
- This type of sequence is known as an array list or vector (older term).

Array List Properties II

Main methods:

- *get(i)*
- *set(i,e)*
- *add(i,e)*
- *remove(i)*

Supporting/auxiliary methods

- *size()*
- *isEmpty()*

There is no restriction on using an array to implement an array list. What is important is the index definition!

Adapter pattern

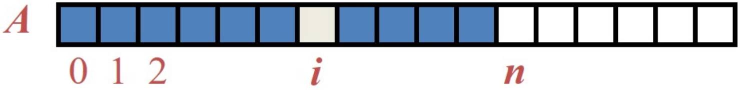
- Writing a new class that uses another class to provide the functionality of the new class.
- A Stack that uses a linked list to provide the Stack functionality.
- An array list to accomplish a Deque.
- Instance of other class as private member reference.
- Table 6.1 example, and Stack and Linked List Example

Array-based implementation

- Use array A: *A[i]* stores a reference to element with index i
- Algorithms: *add(i,e)*, and *remove(i)* (CF 7.3)
- Performance: everything is $O(1)$, except add and remove: $O(n)$ (Table 7.1)
- Actually add and remove run in $O(n - i + 1)$ time.
- Thus adding at the end and removing at the end runs in $O(1)$ time, but adding and removing at the front in $O(n)$ time.
- Consequence for using an array list with arrays for a deque?
- Can we do it in $O(1)$?

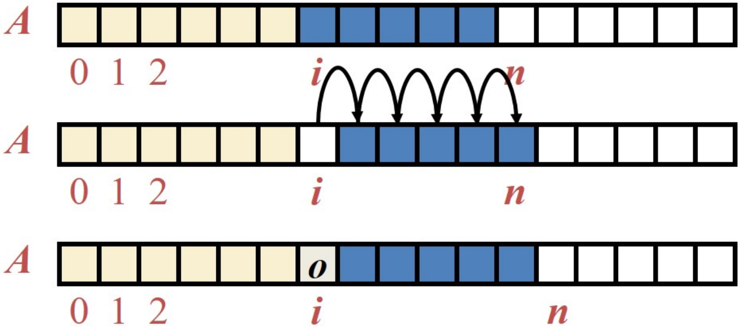
Array-based implementation

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation *get(i)* is implemented in $O(1)$ time by returning $A[i]$
- Operation *set(i,o)* is implemented in $O(1)$ time by performing $t = A[i]$, $A[i] = o$, and returning t.



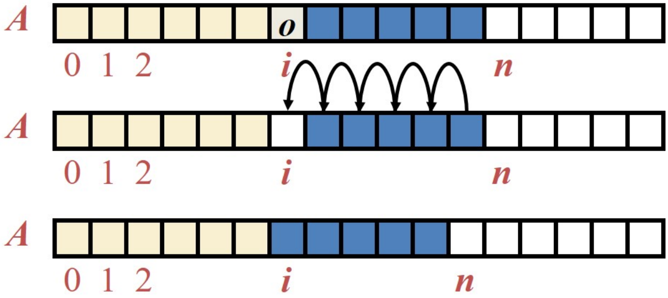
Insertion

- In operation *add(i, o)*, we need to make room for the new element by shifting forward the *n - i* elements *A[i], ..., A[n - 1]*
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

- In operation *remove(i)*, we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

- In the array based implementation of an array list:
  - The space used by the data structure is $O(n)$
  - **size**, **isEmpty**, **get** and **set** run in $O(1)$ time
  - **add** and **remove** run in $O(n)$ time in worst case
- If we use the array in a circular fashion, operations **add(0, x)** and **remove(0, x)** run in $O(1)$ time
- In an **add** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Growable Array-Based Array List

- In an **add(o)** operation (without an index), we always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - **Incremental strategy**: increase the size by a constant c
  - **Doubling strategy**: double the size

```
Algorithm add(o)
  if n = S.length − 1 then
    A = new array of size ...
    for i = 0 to n−1 do
      A[i] = S[i]
    S = A
  n = n + 1
  S[n−1] = o
```

Growable Array

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n *add(o)* operations
- We assume that we start with an empty stack represented by an array of size *1*
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array k = n/c times
- The total time *T(n)* of a series of *n* add operations is proportional to:

$$n + c + 2c + 3c + 4c + \dots + kc =$$
$$n + c(1 + 2 + 3 + \dots + k) =$$
$$n + ck(k + 1)/2$$

Incremental Strategy Analysis II

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e. $O(n^2)$
- k is the number of times the array is replaced therefore it contributes to the runtime.
- The amortized time of a single add operation is $O(n)$
- See Proposition 7.3

Doubling Strategy Analysis

- Fixed size arrays: waste of space, or too little space
- n elements, array A (which supports our array list S) size N.
- As long as $n \leq N$, no problem, and then?

When $n \geq N$, we say the array overflows, and we simply:

1. Allocate new array B of capacity $2N$.
2. Let $B[i] = A[i]$, for $i = 0, 1, ..., N - 1$
3. Let $A = B$, we use B as the array supporting S
4. Insert the new element in A.

Doubling Strategy Analysis II

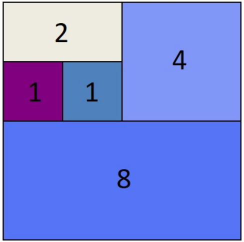
- We replace the array $k = log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to:

$$n + 1 + 2 + 4 + 8 + ... + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

Doubling Strategy Analysis III

- $T(n)$ is $O(n)$
- The amortized time of a single add operation is $O(1)$
- See Proposition 7.2

geometric series



Array List Interface

```java
public interface IndexList<E> {
  public int size( );
  public boolean isEmpty( );
  public void add(int i, E e) throws IndexOutOfBoundsException;
  public E get(i) throws IndexOutOfBoundsException;
  public E remove(int i) throws IndexOutOfBoundsException;
  public E set(int i, E e) throws IndexOutOfBoundsException;
}
```

Array List Interface with indices

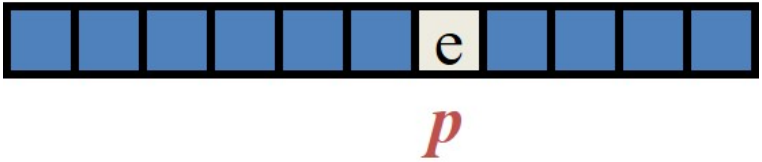
Position ADT

Position ADT

- ADT with a single method (*element()*)
- Positions are relative.
- A position p, associated with an element e does not change even if the index for e changes.
- If we remove e - thereby destroying p, it changes.

Position Implementation

- Say we have a list that contains an element "e" that we want to keep track of
- The index of the element may change depending on insert or remove operations
- A Position p is associated with the element "e" and lets us access it even if the index changes
- A Position ADT is associated with a particular container



Position Implementation II

```java
public interface Position<E> {
  public E element();
}
```

Position Interface

Positional List ADT

Positional List Properties

- Referring to a place in a list without an index.
- If we are using a linked list, then we could use the node as the position for an element in the list.
- Using an index in a linked list means we have to iterate through all the elements, counting as we go.
- Having a node means we can perform $O(1)$ insertions and removals, as the node acts as the position of an element in the list.

Positional List Properties II

- We then have *addBefore(v)*, or *addAfter(v)*, where *v* is a node.
- We do not want to use nodes directly: gives the person using our Positional list access to methods that can change the element, and can also unlink the node, etc.
- Exposes too much information about our implementation.

Positional List Properties III

Elements contained in Nodes, and every node has a Position in the list. Main operations include:

- *first()* - returns the position of the first element in the list or null if empty
- *last()* - returns the position of the last element in the list or null if empty
- *prev(p)* - Returns the position of L immediately before position p (or null if p is the first position).
- *next(p)* - : Returns the position of L immediately after position p (or null if p is the last position)

All implemented referring to Positions, not nodes.

Positional List Properties IV

Update Methods:

- *set(p,e)*
- *addFirst(e)*
- *addLast(e)*
- *addBefore(p,e)*
- *addAfter(p,e)*
- *remove(p)* is similar

Positional List Properties V

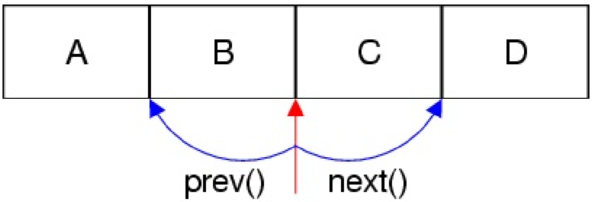
- Redundancy of *addFirst()*, and *addLast()*?
- Exceptions
- Yet Another Deque Adapter using Positional List Methods.
- Implementation using a Doubly Linked List: Implement a $Dnode < E >$ that implements the Position interface.
- Given a position p in S, we can use a narrowing conversion (cast p to a $DNode < E >$) to get the underlying $DNode\ v$

Positional List addAfter

```
addAfter(p,e)
  Create a new node v
  v.setElement(e)
  v.setPrev(p)
  v.setNext(p.getNext())
  (p.getNext()).setPrev(v)
  p.setNext(v)
```

CF addAfter

Iterators



Iterator Properties

- Abstracts the scanning through a collection of elements.
- Can access and make changes to current element in traversal, can go to next element in traversal.
- Traversal is independent from specific implementation of collection.

Iterator Properties II

- Iterator ADT methods:
  - *hasNext()*
  - *next()*
- Extends the concept of position by adding a traversal capability
- Implementation with an array or a singly linked list.

Iterable Classes

- An iterator is typically associated with an another data structure, which can implement the Iterable ADT
- We can augment the Stack, Queue, Array List, List and Sequence ADTs with method:
  - $Iterator < E > iterator()$: returns an iterator over the elements
  - In Java, classes with this method extends $Iterable < E >$

Types of Iterators

Two notions of an iterator:

- **snapshot**: freezes the contents of the data structure at a given time.
- **dynamic**: follows changes to the data structure.
- In Java: an iterator will fail (and throw an exception) if the underlying collection changes unexpectedly.

Using Iterators

- *java.util.Iterator* interface
- *java.lang.Iterable* (!)
- For-each loops
- Create a separate class that stores a reference to the list, and a current location (CF 6.14), and also implements the Iterator

The For-Each Loop

Java provides a simple way of looping through the elements of an Iterable class:

```
for (type name: expression)
  loop_body
```

for each structure

```
List<Integer> values;
int sum=0;
for (Integer i : values)
  sum += i; //is this statement allowed? why?
```

for each example

Implementing Iterators

Array-based:

- array A of the elements
- index i that keeps track of the cursor

Linked List based

- doubly-linked list L storing the elements, with sentinels for header and trailer
- pointer p to node containing the last element returned (or the header if this is a new iterator).

We can add methods to our ADTs that return *iterable* objects, so that we can use the for-each loop on their contents

List Iterators in Java

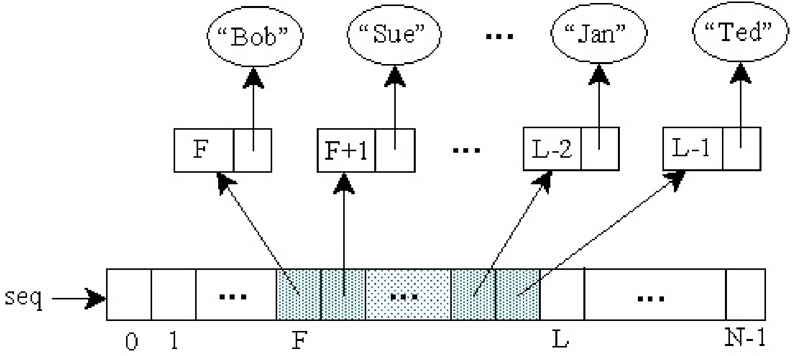
Java uses the *ListIterator* ADT for node-based lists. This iterator includes the following methods:

- *add(e)*: add e at the current cursor position
- *hasNext()*: true if there is an element after the cursor
- *hasPrevious()*: true if there is an element before the cursor
- *previous()*: return the element *e* before the cursor and move cursor to before *e*
- *next()*: return the element *e* after the cursor and move cursor to after *e*
- *set(e)*: replace the element returned by last next or previous operation with *e*
- *remove()*: remove the element returned by the last next or previous method

Position Iterators

- Create a *positions()* method which returns an *Iterable* object for the positions.
- This *Iterable* object (list of positions) contains a method *iterator()* which can be called to return an iterator on the elements.
- List Iterators in Java invalidate on update for multiple Iterators

Sequence ADT



Sequence Properties

- Provides explicit access to the elements in the list either by indices or positions
- Multiple Inheritance.
- Implementing with an Array?

Sequence Properties II

- The Sequence ADT is the union of the Array List and Positional List ADTs
- Elements accessed by
  - Index, or
  - Position
- Generic methods:
  - size(), isEmpty()
- ArrayList-based methods:
  - get(i), set(i, o), add(i, o), remove(i)

Sequence Properties III

- List-based methods:
  - first()
  - last()
  - prev(p)
  - next(p)
  - replace(p, o)
  - addBefore(p, o)
  - addAfter(p, o)
  - addFirst(o)
  - addLast(o)
  - remove(p)
- Bridge methods:
  - atIndex(i), indexOf(p)

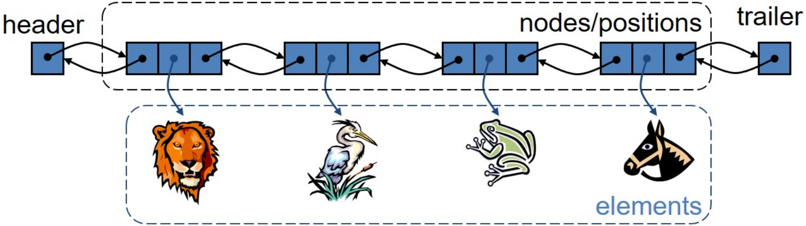
Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- Direct applications:
  - Generic replacement for stack, queue, array list, or list
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures

Linked List Implementation

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

Linked List Implementation II



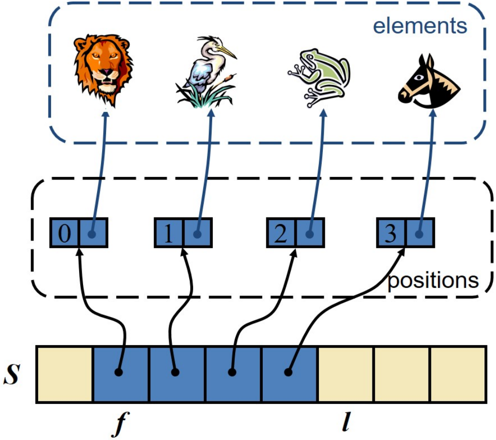
Linked List Implementation III

- Position-based methods run in constant time
- Index-based methods require searching from header or trailer while keeping track of indices; hence, run in linear time

Array-based Implementation

- We use a circular array storing positions
- A position object stores:
  - Element
  - Index
- Indices f and l keep track of first and last positions

Array-based Implementation II



Comparing Sequence Implementations

Operation	Array	List
size, isEmpty	1	1
atIndex, indexOf, get	1	n
first, last, prev, next	1	1
set(p,e)	1	1
set(i,e)	1	n
add, remove(i)	n	n
addFirst, addLast	1	1
addAfter, addBefore	n	1
remove(p)	n	1

Move to Front Heuristic (Favourite Lists)

- List that orders our favourite things (in nondecreasing number of times of access).
- Move to front (principal of locality).
- Run times of a favourites list without the Move to Front.
- Heuristic.
- Trade-offs when finding the top k elements.