Java Collections API

Java Collections Framework

- Contains interfaces, implementation classes for a lot of useful data structures .These are generic interfaces. Located in java.util package.
- Contains algorithms for manipulation of data such as filtering, searching, sorting and aggregation.
- We will focus on interfaces and implementation classes.
- Has two interface hierarchies: (a) Collection and (b) Map
- Collection has sub interfaces : (a) Set, (b) List, (c) Queue

Java Generics

- They are similar to C++ templates
- Useful for parameterizing functions or classes to accomplish functions in a type generic way. Also useful to enforce type restrictions for a lot of classes especially collection classes.

e.g. Arrays.sort() function – to sort array of elements of any type

ArrayList<E> class -- List of elements of any type E

Java Generics Example (contd.)

- Generics introduced in Java 5 allow typed lists to be created
- List interface is generic – List<T>

```
List<String> strList = new ArrayList<String>();
// creates a list to store string objects only
```

In Java 8 and above,

```
List<String> strList = new ArrayList<>();  // no need to specify type in new operation as it is inferred.
strList.add("Hello"); // OK
strList.add(new Integer(17));   // Compiler error
String item = strList.get(0);  // no need to cast
```

Java Generics (contd.)

- Lot of Java API classes including collection framework are generic types.
- Java 7 and above allows both generic and non-generic versions of these classes (e.g. List<T>, List) for backward compatibility
- Use always generic types when available.

List ADT

- List<E> has basic methods:

1. Add an element E at index position; changes indices of elements at positions >= index

   void add(int index, E element);

   Special case : add(E element) => append to list

2. Set an element E at index

   void set(int index, E element);

3. Get an element at index

   E get(int index);

4. Remove an element at index; changes indices of elements at positions >= index

   E remove(int index);

5. Useful additional methods : size(), isEmpty()

Implementations of list

- ArrayList<E> - List implemented as an array
- LinkedList<E> - List implemented as a doubly linked list of nodes.
- In Java, implementation classes usually derive from an abstract class for implementing methods common to all specific implementations of the interface.

  (e.g. AbstractList<E>)

ArrayList<E>

- Uses dynamic arrays that can expand or contract as needed.
- Usually an array is expanded to have extra capacity so as to avoid repeated expansions in the future.
- You might have observed that doubling an array during expansion gives good average time complexity over a sequence of insert operations (also called “amortized time”)

Amortization

- Many dynamic data structures do well for a sequence of operations though the worst-case time complexity based on a single operation may be high
- But we amortize the restructuring cost of a data structure over the sequence of n operations – restructure for future benefit
- Amortized time complexity measures the average time complexity over a sequence of operations.

Amortized time complexity

- Consider a sequence of n add() (append) operations and we measure the cost as # of element inserts + # of element copy operations
- What is the cost c_i of i-th add operation starting from empty array?

  c_i = 1 if there is no array expansion else $c\_i = i$ (1 insert and $(i-1)$ copy operations). Thus we have costs : 1,2,3,1,5,1,1,1,9,….
- May conclude that the worst-case time complexity is O(n) for a single insert with total time complexity is $O(n^2)$ in the worst-case but there is a tighter bound
- Note that $c\_i = i$ only if $(i-1)$ is an exact power of 2, else $c\_i = 1$
- $\sum_{i=1}^{i=n} c\_i$ = n * 1 + (total number of copy operations) < n + $\sum_{j=1}^{\log n} 2^j$ = n+ 2n = 3n which is O(n)
- Amortized complexity for a single insert = total complexity / n = O(1)

ArrayList complexity

- get(index) – O(1) worst-case
- set(index() – O(1) worst-case
- add(e) – O(1) amortized complexity
- add(index,e) – O(n) worst-case where n is size of array at time of insert + O(1) amortized complexity for copying
- remove(index) – O(n) worst-case + O(1) amortized complexity for copying
- remove(size()- 1) (special case) – O(1) amortized complexity
- indexOf(elem) – O(n) worst-case

Linked list implementation

- Has a list node that contains element. List node has references to other elements in the list
- Singly linked list – has reference to only next element in the list. The list data structure typically needs to store reference to first list element.
- Doubly linked list - has references to next element as previous element in the list. The list data structure typically needs to store references to first and last list elements.

ListNode<E>

```
public class MyLinkedList<E> implements List<E> {
        private class ListNode<E> {
            private ListNode<E> prev, next;
            private E elem;
            public ListNode(E elem, ListNode<E> prev,
                                    ListNode<E> next) {
               this.elem = elem;
               this.prev = prev; this.next = next;
            }
        }
        private ListNode<E> first, last;
        private int size;
}
```

Linked list methods

- Accessor methods (return positions):
  - getFirst(), getLast()
  - getPrev(p), getNext(p)
- Update methods:
  - set(p, e)
  - addBefore(p, e), addAfter(p, e),
  - addFirst(e), addLast(e)
  - remove(p)

Linked list methods

```
private ListNode<E> addBefore(ListNode<E> pos,
                              E elem) {
    ListNode<E> newNode = new ListNode<>(elem,
            pos == null ? null : pos.prev, pos);
    if (newNode.next != null) {
        newNode.next.prev= newNode;
    } else { last = newNode; }
    if (newNode.prev != null) {
        newNode.prev.next = newNode;
    } else { first = newNode;  }
    size++;
    return newNode;
}
```

Linked list methods

```
private E remove(ListNode<E> pos) {
      if (pos.next != null) {
            pos.next.prev= pos.prev;
       } else { last = pos.prev; }

      if (pos.prev != null) {
            pos.prev.next = pos.next;
      } else {  first = pos.next;  }

       pos.next = pos.prev = null;
       size--;
       return pos.elem;
}
```

Linked list methods

- All the operations knowing the positions can be done in O(1) time
- Finding the position for an index takes O(n) time in the worst-case.
- Hence all index related operations of List ADT take O(n) time in the worst-case.
- No additional space complexity unlike arraylist which can have as much as O(n) time in the worst-case

List use examples

```
List<Employee> empList = new ArrayList<>();
Employee employee1 = new Employee(14567);
Employee employee2 = new Manager(45789, 345);
empList.add(employee1);
empList.add(employee2);
Manager mgr1 =  (Manager) empList.get(1);
empList.contains(employee1)  -- returns true
empList.remove(0);  // removes employee1
```

- Note that when comparing each element, it invokes each element's equals() method. For first element Employee class's equals() implementation (if it exists) will be called and for second element Manager class's equals() implementation (if it exists) will be called.

Iterator Design Pattern

- Allows a collection to control its own navigation order and hides the details from the consumer

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    …..
}
```

- Collection<E> has a method to get an iterator:

```
Iterator<E> iterator() { ..}
```

Collection Iterators

```
List<String> list = new ArrayList<>();
list.add("Hello");
list.add("!!");
list.add(1,"World");
```

- Old style code:

```
Iterator<String> lit = list.iterator();
while (lit.hasNext()) {
   System.out.println(lit.next());
} // prints " Hello","World","!!"
```

- New style code:

```
for (String str : list) {
    System.out.println(str);
}
```