Vectors, Lists, and Sequences

Sequence Types

Sequence: collection of elements organized in a specified order

- allows random access by rank or position

Stack: sequence that can be accessed in LIFO fashion

Queue: sequence that can be accessed in FIFO fashion

Deque: sequence accessed by added to or removing from either end

Vector: sequence with random access by rank

List: sequence with random access by position

Places in a Sequence

Rank

- Place specified by number of places before the place in question
- Abstraction of the concept of array index

Position

- Place specified by which place precedes and which place follows the place in question
- Abstraction of the concept of (doubly) linked list node

Vector ADT

elemAtRank(r) : return element at specified rank
replaceAtRank(r,e) : replace element at specified rank with new element

- provided for efficiency

insertAtRank(r,e) : insert new element at specified rank

- existing places at that rank or higher are incremented

removeAtRank(r) : remove element at specified rank

- places following that rank are decremented

Analysis

elemAtRank: \(O(1)\)
replaceAtRank: \(O(1)\)
insertAtRank: \(O(n)\)
removeAtRank: \(O(n)\)

Note: insert and remove are \(O(1)\) at end

- may make \(O(1)\) at start with index wrapping, as in queue

Array-based Vector Implementation

```java
insertAtRank(r, e) {
    if (size == A.length)
        throw new VectorFull();
    for (int i=size-1; i>=r; i--)
        A[i+1] = A[i];
    A[r] = e;
    size++;
}
removeAtRank(r) {
    Object e = A[r];
    for (int i=r; i<size-1; i++)
        A[i] = A[i+1];
    size--;
}
```
Extendable (dynamic) Array Implementation

Grow array as necessary

- Allocate new, larger array
- Copy old elements to new array

Stack push() with Dynamic Array

```java
public void push(Object obj) {
    if (size() == S.length) {
        Object[] A = new Object[S.length*2];
        for (int i=0; i<S.length; i++)
            A[i] = S[i];
        S = A;
    }
    S[++topIndex] = obj;
}
```

Efficient Stack push() with Dynamic Array

```java
public void push(Object obj) {
    if (size() == S.length) {
        Object[] A = new Object[S.length*2];
        for (int i=0; i<S.length; i++)
            A[i] = S[i];
        S = A;
    }
    S[++topIndex] = obj;
}
```

Amortization

Useful analysis tool

When some calls are more expensive than others, average out the costs over the total number of calls

- After every n calls to push, 1 call takes $O(n)$ instead of $O(1)$
- Averaged out over n calls, each call is still $O(1)$

Formal Amortization Analysis

Assume each push() costs $1 in compute time

Overcharge these push() operations

- charge $3$ each
- store $S2$ in the bank for each operation

Now when the extend happens, use money from the bank to pay for the copy operations

We pay for all n operations using a constant cost for each operation

- implies $O(n)$ total cost, or average of $O(1)$ per operation

Analysis with Summations

Work done in “extend” for n pushes:

$$1 + 2 + 4 + 8 + \ldots + n = \sum_{i=2}^{n+1} 2^i = 2^{\log_2 n + 2} - 1 = 2n - 1 = O(n)$$ for n pushes

(this is a base 2 number with $\log n + 1$ bits, all set to 1)
Vector Insert with Dynamic Array

```java
public void insertAtRank(int r, Object e) {
    if (size == a.length) {
        Object b[] = new Object[a.length*2];
        for (int i=0; i<size; i++)
            b[i] = a[i];
        a = b;
    }
    for (int i=size-1; i>=r; i--)
        a[i+1] = a[i];
    a[r] = e;
    size++;
}
```

Analysis of Insert with Extendable Array

For general insert (at any rank), still $O(n)$
For insert at last position, still $O(1)$

- Naïve analysis might yield $O(n)$
- Amortized analysis reveals $O(1)$

List ADT

```java
size( ), isEmpty( ), isFirst(p), isLast(p)
first/last( ); Return first/last position
before/after(p) : Return preceding/following position
replaceElement(p,e) : Set the element for a position
swapElements(p,q) : Exchange elements for two positions
insertBefore(After(p,e) : Create new position
before/after p containing element e
remove(p) : Remove the position p and the element it contains
insertFirst, insertLast(e) : Create new position at
start/end of list and set its element
```

What's a position, again?

```java
interface Position {
    public Object element();
}
```

May be implemented as singly- or doubly-linked list node, array element, etc.

List containing the position must have some way of locating the position before and after a given position

Doubly Linked List Implementation

```java
public class MyList implements List {
    protected int numElts=0;
    protected DLNode header=new Node(null,null,null);
    protected DLNode trailer = new Node(header, null,
null);
    public Position insertBefore(Position p, Object e) {
        numElts++;
        DLNode nd = (DLNode)p;
        DLNode newNode=new DLNode(nd.getPrev(), nd, e);
        nd.getPrev().setNext(newNode);
        newNode.setPrev(nd);
        return newNode;
    }
}
```

Analysis

All methods of List using doubly linked list are $O(1)$
Sequence ADT

Supports ADT of vector and list

Plus:

• atRank(r) : Converts a rank to a position
• rankOf(p) : Converts a position to a rank

Sequence Interface

interface Sequence extends List, Vector {
    public Position atRank(int rank);
    public int rankOf(Position position);
}

Implementing with DL List

All methods from List interface are \(O(1)\)

atRank() and rankOf()?

• Both \(O(n)\)

Perform Vector methods by first finding Position at proper rank, then doing insert, delete, etc.

• Finding position is \(O(n)\), though the actual insert/delete is only \(O(1)\)

Implementing with Array

atRank( ) ?

• \(O(1)\)

rankOf( ) ?

• At each position, store element plus array index
• \(O(1)\)

Insert/remove atRank/Before/After?

• all \(O(n)\)

Comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>first, last, before, after</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>insertFirst, insertLast</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>replaceElement, swapElement</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>insertAfter, insertBefore</td>
<td>(O(n)) &gt; (O(1))</td>
<td>(O(n)) &gt; (O(1))</td>
</tr>
<tr>
<td>remove</td>
<td>(O(n)) &gt; (O(1))</td>
<td>(O(n)) &gt; (O(1))</td>
</tr>
<tr>
<td>atRank, rankOf, elemAtRank</td>
<td>(O(1)) &lt; (O(n))</td>
<td>(O(1)) &lt; (O(n))</td>
</tr>
<tr>
<td>replaceAtRank</td>
<td>(O(1)) &lt; (O(n))</td>
<td>(O(1)) &lt; (O(n))</td>
</tr>
<tr>
<td>insertAtRank, remvAtRank</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>