Chapter 2

Array-Based Lists

In this chapter, we will study implementations of the `List` and `Queue` interfaces where the underlying data is stored in an array, called the *backing array*. The following table summarizes the running times of operations for the data structures presented in this chapter:

	`get(i)`/`set(i,x)`	`add(i,x)`/`remove(i)`
`ArrayStack`	$O(1)$	$O(n-i)$
`ArrayDeque`	$O(1)$	$O(\min\{i,n-i\})$
`DualArrayDeque`	$O(1)$	$O(\min\{i,n-i\})$
`RootishArrayStack`	$O(1)$	$O(n-i)$

Data structures that work by storing data in a single array have many advantages and limitations in common:

- Arrays offer constant time access to any value in the array. This is what allows `get(i)` and `set(i,x)` to run in constant time.

- Arrays are not very dynamic. Adding or removing an element near the middle of a list means that a large number of elements in the array need to be shifted to make room for the newly added element or to fill in the gap created by the deleted element. This is why the operations `add(i,x)` and `remove(i)` have running times that depend on `n` and `i`.

- Arrays cannot expand or shrink. When the number of elements in the data structure exceeds the size of the backing array, a new array needs to be allocated and the data from the old array needs to be copied into the new array. This is an expensive operation.

The third point is important. The running times cited in the table above do not include the cost associated with growing and shrinking the backing array. We will see that, if carefully managed, the cost of growing and shrinking the backing array does not add much to the cost of an *average* operation. More precisely, if we start with an empty data structure, and perform any sequence of m add(i, x) or remove(i) operations, then the total cost of growing and shrinking the backing array, over the entire sequence of m operations is $O(m)$. Although some individual operations are more expensive, the amortized cost, when amortized over all m operations, is only $O(1)$ per operation.

2.1 ArrayStack: Fast Stack Operations Using an Array

An ArrayStack implements the list interface using an array a, called the *backing array*. The list element with index i is stored in a[i]. At most times, a is larger than strictly necessary, so an integer n is used to keep track of the number of elements actually stored in a. In this way, the list elements are stored in a[0],...,a[n − 1] and, at all times, a.length ≥ n.

ArrayStack
```
T[] a;
int n;
int size() {
  return n;
}
```

2.1.1 The Basics

Accessing and modifying the elements of an ArrayStack using get(i) and set(i, x) is trivial. After performing any necessary bounds-checking we simply return or set, respectively, a[i].

ArrayStack
```
T get(int i) {
  return a[i];
}
T set(int i, T x) {
  T y = a[i];
  a[i] = x;
  return y;
}
```

The operations of adding and removing elements from an ArrayStack are illustrated in Figure 2.1. To implement the add(i, x) operation, we first check if a is already full. If so, we call the method resize() to increase the size of a. How resize() is implemented will be discussed later. For now, it is sufficient to know that, after a call to resize(), we can be sure that $\mathtt{a.length} > \mathtt{n}$. With this out of the way, we now shift the elements $\mathtt{a}[\mathtt{i}],\ldots,\mathtt{a}[\mathtt{n}-1]$ right by one position to make room for x, set a[i] equal to x, and increment n.

ArrayStack
```
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  for (int j = n; j > i; j--)
    a[j] = a[j-1];
  a[i] = x;
  n++;
}
```

If we ignore the cost of the potential call to resize(), then the cost of the add(i, x) operation is proportional to the number of elements we have to shift to make room for x. Therefore the cost of this operation (ignoring the cost of resizing a) is $O(\mathtt{n}-\mathtt{i}+1)$.

Implementing the remove(i) operation is similar. We shift the elements $\mathtt{a}[\mathtt{i}+1],\ldots,\mathtt{a}[\mathtt{n}-1]$ left by one position (overwriting a[i]) and decrease the value of n. After doing this, we check if n is getting much smaller than a.length by checking if $\mathtt{a.length} \geq 3\mathtt{n}$. If so, then we call resize() to reduce the size of a.

ArrayStack
```
T remove(int i) {
  T x = a[i];
```

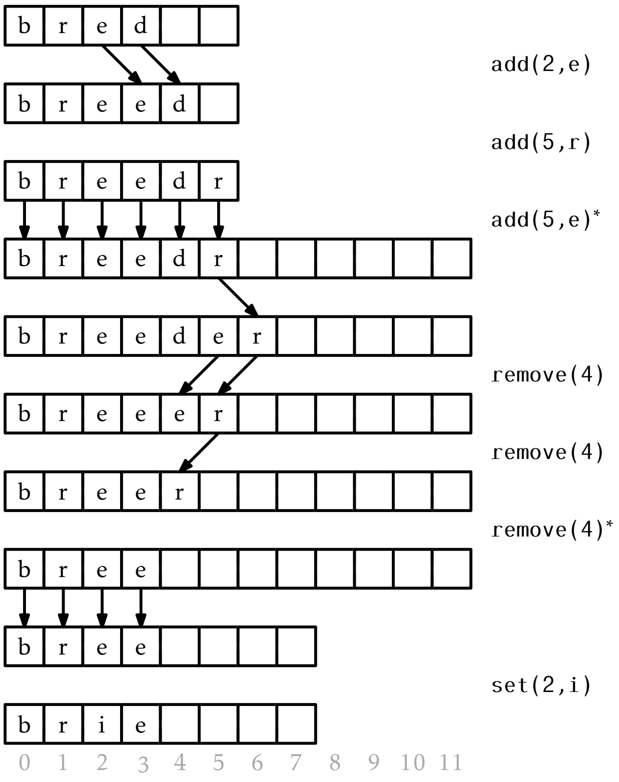

Figure 2.1: A sequence of add(i, x) and remove(i) operations on an ArrayStack. Arrows denote elements being copied. Operations that result in a call to resize() are marked with an asterisk.

```
  for (int j = i; j < n-1; j++)
    a[j] = a[j+1];
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

If we ignore the cost of the resize() method, the cost of a remove(i) operation is proportional to the number of elements we shift, which is $O(\mathtt{n}-\mathtt{i})$.

2.1.2 Growing and Shrinking

The resize() method is fairly straightforward; it allocates a new array b whose size is 2n and copies the n elements of a into the first n positions in b, and then sets a to b. Thus, after a call to resize(), a.length = 2n.

ArrayStack
```
void resize() {
  T[] b = newArray(max(n*2,1));
  for (int i = 0; i < n; i++) {
    b[i] = a[i];
  }
  a = b;
}
```

Analyzing the actual cost of the resize() operation is easy. It allocates an array b of size 2n and copies the n elements of a into b. This takes $O(\mathtt{n})$ time.

The running time analysis from the previous section ignored the cost of calls to resize(). In this section we analyze this cost using a technique known as *amortized analysis*. This technique does not try to determine the cost of resizing during each individual add(i,x) and remove(i) operation. Instead, it considers the cost of all calls to resize() during a sequence of m calls to add(i,x) or remove(i). In particular, we will show:

Lemma 2.1. *If an empty* ArrayList *is created and any sequence of $m \geq 1$ calls to* add(i,x) *and* remove(i) *are performed, then the total time spent during all calls to* resize() *is $O(m)$.*

Proof. We will show that any time `resize()` is called, the number of calls to `add` or `remove` since the last call to `resize()` is at least $\mathtt{n}/2-1$. Therefore, if $\mathtt{n}_i$ denotes the value of `n` during the ith call to `resize()` and r denotes the number of calls to `resize()`, then the total number of calls to `add(i,x)` or `remove(i)` is at least

$$\sum_{i=1}^{r}(\mathtt{n}_i/2-1) \le m \ ,$$

which is equivalent to

$$\sum_{i=1}^{r}\mathtt{n}_i \le 2m+2r \ .$$

On the other hand, the total time spent during all calls to `resize()` is

$$\sum_{i=1}^{r}O(\mathtt{n}_i) \le O(m+r) = O(m) \ ,$$

since r is not more than m. All that remains is to show that the number of calls to `add(i,x)` or `remove(i)` between the $(i-1)$th and the ith call to `resize()` is at least $\mathtt{n}_i/2$.

There are two cases to consider. In the first case, `resize()` is being called by `add(i,x)` because the backing array `a` is full, i.e., $\mathtt{a.length} = \mathtt{n} = \mathtt{n}_i$. Consider the previous call to `resize()`: after this previous call, the size of `a` was `a.length`, but the number of elements stored in `a` was at most $\mathtt{a.length}/2 = \mathtt{n}_i/2$. But now the number of elements stored in `a` is $\mathtt{n}_i = \mathtt{a.length}$, so there must have been at least $\mathtt{n}_i/2$ calls to `add(i,x)` since the previous call to `resize()`.

The second case occurs when `resize()` is being called by `remove(i)` because $\mathtt{a.length} \ge 3\mathtt{n} = 3\mathtt{n}_i$. Again, after the previous call to `resize()` the number of elements stored in `a` was at least $\mathtt{a.length}/2-1$.[1] Now there are $\mathtt{n}_i \le \mathtt{a.length}/3$ elements stored in `a`. Therefore, the number of

[1]The -1 in this formula accounts for the special case that occurs when $\mathtt{n} = 0$ and $\mathtt{a.length} = 1$.

remove(i) operations since the last call to resize() is at least

$$\begin{aligned} R &\geq \texttt{a.length}/2 - 1 - \texttt{a.length}/3 \\ &= \texttt{a.length}/6 - 1 \\ &= (\texttt{a.length}/3)/2 - 1 \\ &\geq \texttt{n}_i/2 - 1 \ . \end{aligned}$$

In either case, the number of calls to add(i,x) or remove(i) that occur between the $(i-1)$th call to resize() and the ith call to resize() is at least $\texttt{n}_i/2-1$, as required to complete the proof. □

2.1.3 Summary

The following theorem summarizes the performance of an ArrayStack:

Theorem 2.1. *An ArrayStack implements the List interface. Ignoring the cost of calls to resize(), an ArrayStack supports the operations*

- get(i) *and* set(i,x) *in* $O(1)$ *time per operation; and*
- add(i,x) *and* remove(i) *in* $O(1+\texttt{n}-\texttt{i})$ *time per operation.*

Furthermore, beginning with an empty ArrayStack and performing any sequence of m add(i,x) *and* remove(i) *operations results in a total of* $O(m)$ *time spent during all calls to* resize().

The ArrayStack is an efficient way to implement a Stack. In particular, we can implement push(x) as add(n,x) and pop() as remove(n − 1), in which case these operations will run in $O(1)$ amortized time.

2.2 FastArrayStack: An Optimized ArrayStack

Much of the work done by an ArrayStack involves shifting (by add(i,x) and remove(i)) and copying (by resize()) of data. In the implementations shown above, this was done using for loops. It turns out that many programming environments have specific functions that are very efficient at copying and moving blocks of data. In the C programming language, there are the memcpy(d,s,n) and memmove(d,s,n) functions. In the C++

language there is the std :: copy(a0,a1,b) algorithm. In Java there is the System.arraycopy(s,i,d,j,n) method.

```
FastArrayStack
void resize() {
  T[] b = newArray(max(2*n,1));
  System.arraycopy(a, 0, b, 0, n);
  a = b;
}
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  System.arraycopy(a, i, a, i+1, n-i);
  a[i] = x;
  n++;
}
T remove(int i) {
  T x = a[i];
  System.arraycopy(a, i+1, a, i, n-i-1);
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

These functions are usually highly optimized and may even use special machine instructions that can do this copying much faster than we could by using a for loop. Although using these functions does not asymptotically decrease the running times, it can still be a worthwhile optimization. In the Java implementations here, the use of the native System.arraycopy(s,i,d,j,n) resulted in speedups of a factor between 2 and 3, depending on the types of operations performed. Your mileage may vary.

2.3 ArrayQueue: An Array-Based Queue

In this section, we present the ArrayQueue data structure, which implements a FIFO (first-in-first-out) queue; elements are removed (using the remove() operation) from the queue in the same order they are added (using the add(x) operation).

Notice that an ArrayStack is a poor choice for an implementation of a FIFO queue. It is not a good choice because we must choose one end of the list upon which to add elements and then remove elements from the other end. One of the two operations must work on the head of the list, which involves calling add(i, x) or remove(i) with a value of $i = 0$. This gives a running time proportional to n.

To obtain an efficient array-based implementation of a queue, we first notice that the problem would be easy if we had an infinite array a. We could maintain one index j that keeps track of the next element to remove and an integer n that counts the number of elements in the queue. The queue elements would always be stored in

$$\mathtt{a}[\mathtt{j}], \mathtt{a}[\mathtt{j}+1], \ldots, \mathtt{a}[\mathtt{j}+\mathtt{n}-1] \enspace .$$

Initially, both j and n would be set to 0. To add an element, we would place it in $\mathtt{a}[\mathtt{j}+\mathtt{n}]$ and increment n. To remove an element, we would remove it from $\mathtt{a}[\mathtt{j}]$, increment j, and decrement n.

Of course, the problem with this solution is that it requires an infinite array. An ArrayQueue simulates this by using a finite array a and *modular arithmetic*. This is the kind of arithmetic used when we are talking about the time of day. For example 10:00 plus five hours gives 3:00. Formally, we say that

$$10 + 5 = 15 \equiv 3 \pmod{12} \enspace .$$

We read the latter part of this equation as "15 is congruent to 3 modulo 12." We can also treat mod as a binary operator, so that

$$15 \bmod 12 = 3 \enspace .$$

More generally, for an integer a and positive integer m, $a \bmod m$ is the unique integer $r \in \{0, \ldots, m-1\}$ such that $a = r + km$ for some integer k. Less formally, the value r is the remainder we get when we divide a by m. In many programming languages, including Java, the mod operator is represented using the % symbol.[2]

[2]This is sometimes referred to as the *brain-dead* mod operator, since it does not correctly implement the mathematical mod operator when the first argument is negative.

Modular arithmetic is useful for simulating an infinite array, since i mod a.length always gives a value in the range $0,\ldots,$ a.length -1. Using modular arithmetic we can store the queue elements at array locations

$$a[j\%a.length], a[(j+1)\%a.length], \ldots, a[(j+n-1)\%a.length] \ .$$

This treats the array a like a *circular array* in which array indices larger than a.length -1 "wrap around" to the beginning of the array.

The only remaining thing to worry about is taking care that the number of elements in the ArrayQueue does not exceed the size of a.

ArrayQueue
```
T[] a;
int j;
int n;
```

A sequence of add(x) and remove() operations on an ArrayQueue is illustrated in Figure 2.2. To implement add(x), we first check if a is full and, if necessary, call resize() to increase the size of a. Next, we store x in a[(j + n)%a.length] and increment n.

ArrayQueue
```
boolean add(T x) {
  if (n + 1 > a.length) resize();
  a[(j+n) % a.length] = x;
  n++;
  return true;
}
```

To implement remove(), we first store a[j] so that we can return it later. Next, we decrement n and increment j (modulo a.length) by setting j = (j + 1) mod a.length. Finally, we return the stored value of a[j]. If necessary, we may call resize() to decrease the size of a.

ArrayQueue
```
T remove() {
  if (n == 0) throw new NoSuchElementException();
  T x = a[j];
  j = (j + 1) % a.length;
```

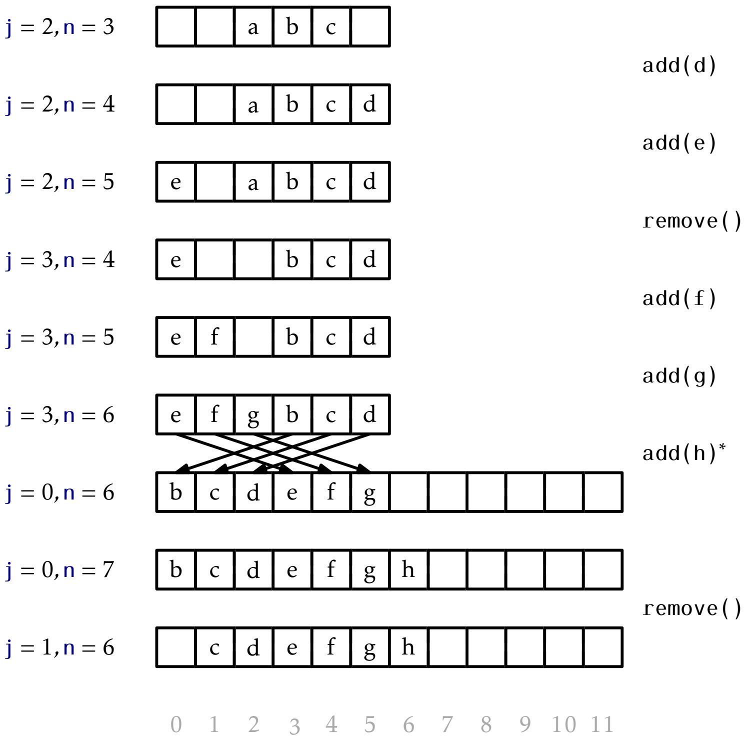

Figure 2.2: A sequence of add(x) and remove(i) operations on an ArrayQueue. Arrows denote elements being copied. Operations that result in a call to resize() are marked with an asterisk.

```
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

Finally, the resize() operation is very similar to the resize() operation of ArrayStack. It allocates a new array b of size 2n and copies

$$\mathtt{a}[\mathtt{j}], \mathtt{a}[(\mathtt{j}+1)\%\mathtt{a.length}], \ldots, \mathtt{a}[(\mathtt{j}+\mathtt{n}-1)\%\mathtt{a.length}]$$

onto

$$\mathtt{b}[0], \mathtt{b}[1], \ldots, \mathtt{b}[\mathtt{n}-1]$$

and sets $\mathtt{j} = 0$.

```
                                ArrayQueue
void resize() {
  T[] b = newArray(max(1,n*2));
  for (int k = 0; k < n; k++)
    b[k] = a[(j+k) % a.length];
  a = b;
  j = 0;
}
```

2.3.1 Summary

The following theorem summarizes the performance of the ArrayQueue data structure:

Theorem 2.2. *An ArrayQueue implements the (FIFO) Queue interface. Ignoring the cost of calls to resize(), an ArrayQueue supports the operations add(x) and remove() in $O(1)$ time per operation. Furthermore, beginning with an empty ArrayQueue, any sequence of m add(i,x) and remove(i) operations results in a total of $O(m)$ time spent during all calls to resize().*

2.4 ArrayDeque: Fast Deque Operations Using an Array

The ArrayQueue from the previous section is a data structure for representing a sequence that allows us to efficiently add to one end of the sequence and remove from the other end. The ArrayDeque data structure allows for efficient addition and removal at both ends. This structure implements the List interface by using the same circular array technique used to represent an ArrayQueue.

ArrayDeque
```
T[] a;
int j;
int n;
```

The get(i) and set(i,x) operations on an ArrayDeque are straightforward. They get or set the array element a[(j + i) mod a.length].

ArrayDeque
```
T get(int i) {
  return a[(j+i)%a.length];
}
T set(int i, T x) {
  T y = a[(j+i)%a.length];
  a[(j+i)%a.length] = x;
  return y;
}
```

The implementation of add(i,x) is a little more interesting. As usual, we first check if a is full and, if necessary, call resize() to resize a. Remember that we want this operation to be fast when i is small (close to 0) or when i is large (close to n). Therefore, we check if i < n/2. If so, we shift the elements a[0],...,a[i − 1] left by one position. Otherwise (i ≥ n/2), we shift the elements a[i],...,a[n − 1] right by one position. See Figure 2.3 for an illustration of add(i,x) and remove(x) operations on an ArrayDeque.

ArrayDeque
```
void add(int i, T x) {
  if (n+1 > a.length) resize();
  if (i < n/2) { // shift a[0],..,a[i-1] left one position
    j = (j == 0) ? a.length - 1 : j - 1; //(j-1)mod a.length
    for (int k = 0; k <= i-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
```

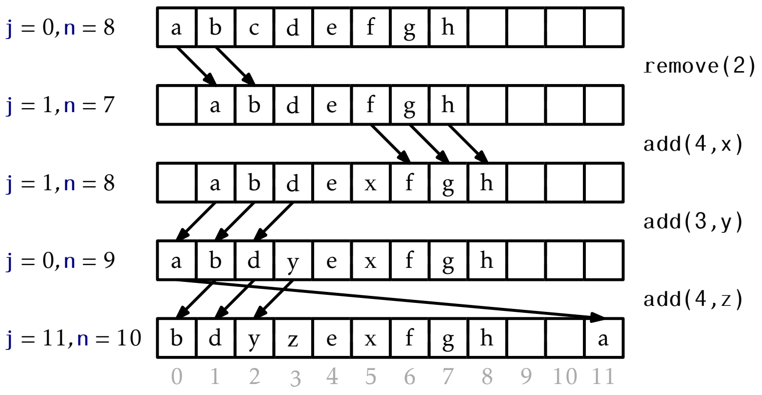

Figure 2.3: A sequence of add(i,x) and remove(i) operations on an ArrayDeque. Arrows denote elements being copied.

```
  } else { // shift a[i],..,a[n-1] right one position
    for (int k = n; k > i; k--)
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
  }
  a[(j+i)%a.length] = x;
  n++;
}
```

By doing the shifting in this way, we guarantee that add(i,x) never has to shift more than $\min\{i,n-i\}$ elements. Thus, the running time of the add(i,x) operation (ignoring the cost of a resize() operation) is $O(1+\min\{i,n-i\})$.

The implementation of the remove(i) operation is similar. It either shifts elements $a[0],\ldots,a[i-1]$ right by one position or shifts the elements $a[i+1],\ldots,a[n-1]$ left by one position depending on whether $i < n/2$. Again, this means that remove(i) never spends more than $O(1+\min\{i,n-i\})$ time to shift elements.

ArrayDeque

```
T remove(int i) {
  T x = a[(j+i)%a.length];
  if (i < n/2) {  // shift a[0],..,[i-1] right one position
    for (int k = i; k > 0; k--)
```

```
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
    j = (j + 1) % a.length;
  } else { // shift a[i+1],..,a[n-1] left one position
    for (int k = i; k < n-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
  }
  n--;
  if (3*n < a.length) resize();
  return x;
}
```

2.4.1 Summary

The following theorem summarizes the performance of the ArrayDeque data structure:

Theorem 2.3. *An ArrayDeque implements the List interface. Ignoring the cost of calls to resize(), an ArrayDeque supports the operations*

- get(i) *and* set(i,x) *in* $O(1)$ *time per operation; and*
- add(i,x) *and* remove(i) *in* $O(1+\min\{i,n-i\})$ *time per operation.*

Furthermore, beginning with an empty ArrayDeque, performing any sequence of m add(i,x) *and* remove(i) *operations results in a total of* $O(m)$ *time spent during all calls to* resize().

2.5 DualArrayDeque: Building a Deque from Two Stacks

Next, we present a data structure, the DualArrayDeque that achieves the same performance bounds as an ArrayDeque by using two ArrayStacks. Although the asymptotic performance of the DualArrayDeque is no better than that of the ArrayDeque, it is still worth studying, since it offers a good example of how to make a sophisticated data structure by combining two simpler data structures.

A DualArrayDeque represents a list using two ArrayStacks. Recall that an ArrayStack is fast when the operations on it modify elements near the end. A DualArrayDeque places two ArrayStacks, called front and back, back-to-back so that operations are fast at either end.

DualArrayDeque
```
List<T> front;
List<T> back;
```

A DualArrayDeque does not explicitly store the number, n, of elements it contains. It doesn't need to, since it contains n = front.size() + back.size() elements. Nevertheless, when analyzing the DualArrayDeque we will still use n to denote the number of elements it contains.

DualArrayDeque
```
int size() {
  return front.size() + back.size();
}
```

The front ArrayStack stores the list elements that whose indices are 0,...,front.size() − 1, but stores them in reverse order. The back ArrayStack contains list elements with indices in front.size(),...,size() − 1 in the normal order. In this way, get(i) and set(i,x) translate into appropriate calls to get(i) or set(i,x) on either front or back, which take $O(1)$ time per operation.

DualArrayDeque
```
T get(int i) {
  if (i < front.size()) {
    return front.get(front.size()-i-1);
  } else {
    return back.get(i-front.size());
  }
}
T set(int i, T x) {
  if (i < front.size()) {
    return front.set(front.size()-i-1, x);

  } else {
    return back.set(i-front.size(), x);
  }
}
```

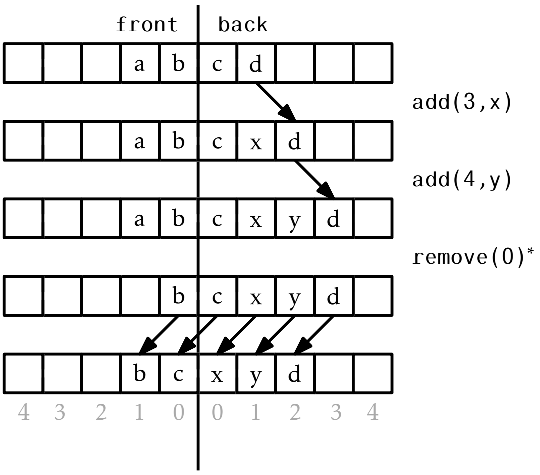

Figure 2.4: A sequence of add(i,x) and remove(i) operations on a DualArrayDeque. Arrows denote elements being copied. Operations that result in a rebalancing by balance() are marked with an asterisk.

Note that if an index i < front.size(), then it corresponds to the element of front at position front.size()−i−1, since the elements of front are stored in reverse order.

Adding and removing elements from a DualArrayDeque is illustrated in Figure 2.4. The add(i,x) operation manipulates either front or back, as appropriate:

```
DualArrayDeque
void add(int i, T x) {
  if (i < front.size()) {
    front.add(front.size()-i, x);
  } else {
    back.add(i-front.size(), x);
  }
  balance();
}
```

The add(i,x) method performs rebalancing of the two ArrayStacks front and back, by calling the balance() method. The implementation of balance() is described below, but for now it is sufficient to know that balance() ensures that, unless size() < 2, front.size() and back.size() do not differ by more than a factor of 3. In particular, $3 \cdot \texttt{front.size()} \geq \texttt{back.size()}$ and $3 \cdot \texttt{back.size()} \geq \texttt{front.size()}$.

Next we analyze the cost of add(i,x), ignoring the cost of calls to balance(). If $\texttt{i} < \texttt{front.size()}$, then add(i,x) gets implemented by the call to $\texttt{front.add}(\texttt{front.size()} - \texttt{i} - 1, \texttt{x})$. Since front is an ArrayStack, the cost of this is

$$O(\texttt{front.size()} - (\texttt{front.size()} - \texttt{i} - 1) + 1) = O(\texttt{i} + 1) \ . \qquad (2.1)$$

On the other hand, if $\texttt{i} \geq \texttt{front.size()}$, then add(i,x) gets implemented as $\texttt{back.add}(\texttt{i} - \texttt{front.size()}, \texttt{x})$. The cost of this is

$$O(\texttt{back.size()} - (\texttt{i} - \texttt{front.size()}) + 1) = O(\texttt{n} - \texttt{i} + 1) \ . \qquad (2.2)$$

Notice that the first case (2.1) occurs when $\texttt{i} < \texttt{n}/4$. The second case (2.2) occurs when $\texttt{i} \geq 3\texttt{n}/4$. When $\texttt{n}/4 \leq \texttt{i} < 3\texttt{n}/4$, we cannot be sure whether the operation affects front or back, but in either case, the operation takes $O(\texttt{n}) = O(\texttt{i}) = O(\texttt{n} - \texttt{i})$ time, since $\texttt{i} \geq \texttt{n}/4$ and $\texttt{n} - \texttt{i} > \texttt{n}/4$. Summarizing the situation, we have

$$\text{Running time of } \texttt{add(i,x)} \leq \begin{cases} O(1 + \texttt{i}) & \text{if } \texttt{i} < \texttt{n}/4 \\ O(\texttt{n}) & \text{if } \texttt{n}/4 \leq \texttt{i} < 3\texttt{n}/4 \\ O(1 + \texttt{n} - \texttt{i}) & \text{if } \texttt{i} \geq 3\texttt{n}/4 \end{cases}$$

Thus, the running time of add(i,x), if we ignore the cost of the call to balance(), is $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$.

The remove(i) operation and its analysis resemble the add(i,x) operation and analysis.

DualArrayDeque

```
T remove(int i) {
  T x;
  if (i < front.size()) {
    x = front.remove(front.size()-i-1);
  } else {
    x = back.remove(i-front.size());
  }
  balance();
  return x;
}
```

2.5.1 Balancing

Finally, we turn to the balance() operation performed by add(i,x) and remove(i). This operation ensures that neither front nor back becomes too big (or too small). It ensures that, unless there are fewer than two elements, each of front and back contain at least $n/4$ elements. If this is not the case, then it moves elements between them so that front and back contain exactly $\lfloor n/2 \rfloor$ elements and $\lceil n/2 \rceil$ elements, respectively.

DualArrayDeque
```
void balance() {
  int n = size();
  if (3*front.size() < back.size()) {
    int s = n/2 - front.size();
    List<T> l1 = newStack();
    List<T> l2 = newStack();
    l1.addAll(back.subList(0,s));
    Collections.reverse(l1);
    l1.addAll(front);
    l2.addAll(back.subList(s, back.size()));
    front = l1;
    back = l2;
  } else if (3*back.size() < front.size()) {
    int s = front.size() - n/2;
    List<T> l1 = newStack();
    List<T> l2 = newStack();
    l1.addAll(front.subList(s, front.size()));
    l2.addAll(front.subList(0, s));
    Collections.reverse(l2);
    l2.addAll(back);
    front = l1;
    back = l2;
  }
}
```

Here there is little to analyze. If the balance() operation does rebalancing, then it moves $O(\mathtt{n})$ elements and this takes $O(\mathtt{n})$ time. This is bad, since `balance()` is called with each call to `add(i,x)` and `remove(i)`. However, the following lemma shows that, on average, `balance()` only spends a constant amount of time per operation.

Lemma 2.2. *If an empty* `DualArrayDeque` *is created and any sequence of $m \geq 1$ calls to* `add(i,x)` *and* `remove(i)` *are performed, then the total time spent during all calls to* `balance()` *is $O(m)$.*

Proof. We will show that, if `balance()` is forced to shift elements, then the number of `add(i,x)` and `remove(i)` operations since the last time any elements were shifted by `balance()` is at least $\mathtt{n}/2-1$. As in the proof of Lemma 2.1, this is sufficient to prove that the total time spent by `balance()` is $O(m)$.

We will perform our analysis using a technique knows as the *potential method*. Define the *potential*, Φ, of the `DualArrayDeque` as the difference in size between `front` and `back`:

$$\Phi = |\mathtt{front.size()} - \mathtt{back.size()}| \enspace .$$

The interesting thing about this potential is that a call to `add(i,x)` or `remove(i)` that does not do any balancing can increase the potential by at most 1.

Observe that, immediately after a call to `balance()` that shifts elements, the potential, Φ_0, is at most 1, since

$$\Phi_0 = \left|\lfloor \mathtt{n}/2\rfloor - \lceil \mathtt{n}/2\rceil\right| \leq 1 \enspace .$$

Consider the situation immediately before a call to `balance()` that shifts elements and suppose, without loss of generality, that `balance()` is shifting elements because $3\mathtt{front.size()} < \mathtt{back.size()}$. Notice that, in this case,

$$\begin{aligned}
\mathtt{n} &= \mathtt{front.size()} + \mathtt{back.size()} \\
&< \mathtt{back.size()}/3 + \mathtt{back.size()} \\
&= \frac{4}{3}\mathtt{back.size()}
\end{aligned}$$

Furthermore, the potential at this point in time is

$$\begin{aligned}\Phi_1 &= \texttt{back.size()} - \texttt{front.size()} \\ &> \texttt{back.size()} - \texttt{back.size()}/3 \\ &= \frac{2}{3}\texttt{back.size()} \\ &> \frac{2}{3} \times \frac{3}{4}\texttt{n} \\ &= \texttt{n}/2\end{aligned}$$

Therefore, the number of calls to add(i,x) or remove(i) since the last time balance() shifted elements is at least $\Phi_1 - \Phi_0 > \texttt{n}/2 - 1$. This completes the proof. □

2.5.2 Summary

The following theorem summarizes the properties of a DualArrayDeque:

Theorem 2.4. *A DualArrayDeque implements the List interface. Ignoring the cost of calls to* resize() *and* balance()*, a DualArrayDeque supports the operations*

- get(i) *and* set(i,x) *in* $O(1)$ *time per operation; and*
- add(i,x) *and* remove(i) *in* $O(1+\min\{\texttt{i},\texttt{n}-\texttt{i}\})$ *time per operation.*

Furthermore, beginning with an empty DualArrayDeque, any sequence of m add(i,x) *and* remove(i) *operations results in a total of* $O(m)$ *time spent during all calls to* resize() *and* balance().

2.6 RootishArrayStack: A Space-Efficient Array Stack

One of the drawbacks of all previous data structures in this chapter is that, because they store their data in one or two arrays and they avoid resizing these arrays too often, the arrays frequently are not very full. For example, immediately after a resize() operation on an ArrayStack, the backing array a is only half full. Even worse, there are times when only 1/3 of a contains data.

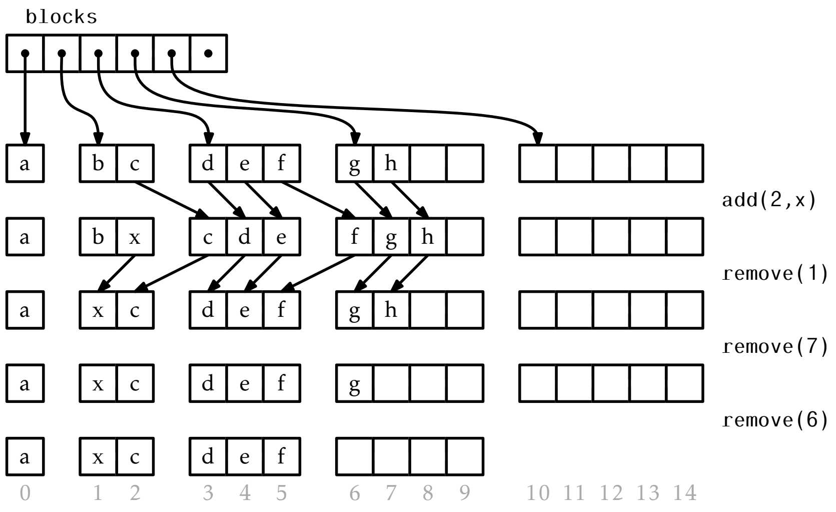

Figure 2.5: A sequence of add(i,x) and remove(i) operations on a RootishArrayStack. Arrows denote elements being copied.

In this section, we discuss the RootishArrayStack data structure, that addresses the problem of wasted space. The RootishArrayStack stores n elements using $O(\sqrt{n})$ arrays. In these arrays, at most $O(\sqrt{n})$ array locations are unused at any time. All remaining array locations are used to store data. Therefore, these data structures waste at most $O(\sqrt{n})$ space when storing n elements.

A RootishArrayStack stores its elements in a list of r arrays called *blocks* that are numbered $0,1,\ldots,r-1$. See Figure 2.5. Block b contains $b+1$ elements. Therefore, all r blocks contain a total of

$$1+2+3+\cdots+r = r(r+1)/2$$

elements. The above formula can be obtained as shown in Figure 2.6.

```
RootishArrayStack
List<T[]> blocks;
int n;
```

As we might expect, the elements of the list are laid out in order within the blocks. The list element with index 0 is stored in block 0,

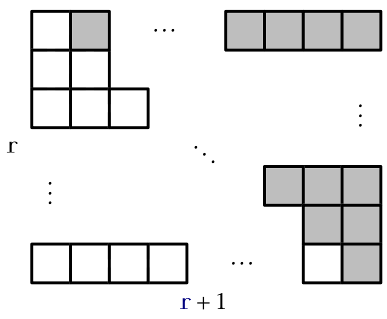

Figure 2.6: The number of white squares is $1+2+3+\cdots+r$. The number of shaded squares is the same. Together the white and shaded squares make a rectangle consisting of $r(r+1)$ squares.

elements with list indices 1 and 2 are stored in block 1, elements with list indices 3, 4, and 5 are stored in block 2, and so on. The main problem we have to address is that of determining, given an index i, which block contains i as well as the index corresponding to i within that block.

Determining the index of i within its block turns out to be easy. If index i is in block b, then the number of elements in blocks $0,\ldots,b-1$ is $b(b+1)/2$. Therefore, i is stored at location

$$j = i - b(b+1)/2$$

within block b. Somewhat more challenging is the problem of determining the value of b. The number of elements that have indices less than or equal to i is $i+1$. On the other hand, the number of elements in blocks $0,\ldots,b$ is $(b+1)(b+2)/2$. Therefore, b is the smallest integer such that

$$(b+1)(b+2)/2 \geq i+1 \ .$$

We can rewrite this equation as

$$b^2 + 3b - 2i \geq 0 \ .$$

The corresponding quadratic equation $b^2+3b-2i=0$ has two solutions: $b=(-3+\sqrt{9+8i})/2$ and $b=(-3-\sqrt{9+8i})/2$. The second solution makes no sense in our application since it always gives a negative value. Therefore, we obtain the solution $b=(-3+\sqrt{9+8i})/2$. In general, this solution is not an integer, but going back to our inequality, we want the smallest integer b such that $b \geq (-3+\sqrt{9+8i})/2$. This is simply

$$b = \left\lceil (-3+\sqrt{9+8i})/2 \right\rceil \ .$$

RootishArrayStack

```
int i2b(int i) {
  double db = (-3.0 + Math.sqrt(9 + 8*i)) / 2.0;
  int b = (int)Math.ceil(db);
  return b;
}
```

With this out of the way, the get(i) and set(i,x) methods are straightforward. We first compute the appropriate block b and the appropriate index j within the block and then perform the appropriate operation:

RootishArrayStack

```
T get(int i) {
  int b = i2b(i);
  int j = i - b*(b+1)/2;
  return blocks.get(b)[j];
}
T set(int i, T x) {
  int b = i2b(i);
  int j = i - b*(b+1)/2;
  T y = blocks.get(b)[j];
  blocks.get(b)[j] = x;
  return y;
}
```

If we use any of the data structures in this chapter for representing the blocks list, then get(i) and set(i,x) will each run in constant time.

The add(i,x) method will, by now, look familiar. We first check to see if our data structure is full, by checking if the number of blocks r is such that $r(r+1)/2 = n$. If so, we call grow() to add another block. With this done, we shift elements with indices $i,\ldots,n-1$ to the right by one position to make room for the new element with index i:

```
                                RootishArrayStack
void add(int i, T x) {
  int r = blocks.size();
  if (r*(r+1)/2 < n + 1) grow();
  n++;
  for (int j = n-1; j > i; j--)
    set(j, get(j-1));
  set(i, x);
}
```

The grow() method does what we expect. It adds a new block:

```
                                RootishArrayStack
void grow() {
  blocks.add(newArray(blocks.size()+1));
}
```

Ignoring the cost of the grow() operation, the cost of an add(i,x) operation is dominated by the cost of shifting and is therefore $O(1+\mathtt{n}-\mathtt{i})$, just like an ArrayStack.

The remove(i) operation is similar to add(i,x). It shifts the elements with indices $\mathtt{i}+1,\ldots,\mathtt{n}$ left by one position and then, if there is more than one empty block, it calls the shrink() method to remove all but one of the unused blocks:

```
                                RootishArrayStack
T remove(int i) {
  T x = get(i);
  for (int j = i; j < n-1; j++)
    set(j, get(j+1));
  n--;
  int r = blocks.size();
  if ((r-2)*(r-1)/2 >= n)  shrink();
  return x;
}
```

```
                                RootishArrayStack
void shrink() {
  int r = blocks.size();
```

```
    while (r > 0 && (r-2)*(r-1)/2 >= n) {
      blocks.remove(blocks.size()-1);
      r--;
    }
  }
```

Once again, ignoring the cost of the shrink() operation, the cost of a remove(i) operation is dominated by the cost of shifting and is therefore $O(n - i)$.

2.6.1 Analysis of Growing and Shrinking

The above analysis of add(i,x) and remove(i) does not account for the cost of grow() and shrink(). Note that, unlike the ArrayStack.resize() operation, grow() and shrink() do not copy any data. They only allocate or free an array of size r. In some environments, this takes only constant time, while in others, it may require time proportional to r.

We note that, immediately after a call to grow() or shrink(), the situation is clear. The final block is completely empty, and all other blocks are completely full. Another call to grow() or shrink() will not happen until at least $r-1$ elements have been added or removed. Therefore, even if grow() and shrink() take $O(r)$ time, this cost can be amortized over at least $r-1$ add(i,x) or remove(i) operations, so that the amortized cost of grow() and shrink() is $O(1)$ per operation.

2.6.2 Space Usage

Next, we analyze the amount of extra space used by a RootishArrayStack. In particular, we want to count any space used by a RootishArrayStack that is not an array element currently used to hold a list element. We call all such space *wasted space*.

The remove(i) operation ensures that a RootishArrayStack never has more than two blocks that are not completely full. The number of blocks, r, used by a RootishArrayStack that stores n elements therefore satisfies

$$(r-2)(r-1) \le n \ .$$

Again, using the quadratic equation on this gives

$$r \leq (3 + \sqrt{1 + 4n})/2 = O(\sqrt{n}) \ .$$

The last two blocks have sizes r and $r - 1$, so the space wasted by these two blocks is at most $2r-1 = O(\sqrt{n})$. If we store the blocks in (for example) an `ArrayList`, then the amount of space wasted by the `List` that stores those r blocks is also $O(r) = O(\sqrt{n})$. The other space needed for storing n and other accounting information is $O(1)$. Therefore, the total amount of wasted space in a `RootishArrayStack` is $O(\sqrt{n})$.

Next, we argue that this space usage is optimal for any data structure that starts out empty and can support the addition of one item at a time. More precisely, we will show that, at some point during the addition of n items, the data structure is wasting an amount of space at least in $\sqrt{n}$ (though it may be only wasted for a moment).

Suppose we start with an empty data structure and we add n items one at a time. At the end of this process, all n items are stored in the structure and distributed among a collection of r memory blocks. If $r \geq \sqrt{n}$, then the data structure must be using r pointers (or references) to keep track of these r blocks, and these pointers are wasted space. On the other hand, if $r < \sqrt{n}$ then, by the pigeonhole principle, some block must have a size of at least $n/r > \sqrt{n}$. Consider the moment at which this block was first allocated. Immediately after it was allocated, this block was empty, and was therefore wasting $\sqrt{n}$ space. Therefore, at some point in time during the insertion of n elements, the data structure was wasting $\sqrt{n}$ space.

2.6.3 Summary

The following theorem summarizes our discussion of the `RootishArray-Stack` data structure:

Theorem 2.5. *A `RootishArrayStack` implements the `List` interface. Ignoring the cost of calls to `grow()` and `shrink()`, a `RootishArrayStack` supports the operations*

- `get(i)` *and* `set(i,x)` *in* $O(1)$ *time per operation; and*
- `add(i,x)` *and* `remove(i)` *in* $O(1 + n - i)$ *time per operation.*

Furthermore, beginning with an empty `RootishArrayStack`*, any sequence of m* `add(i,x)` *and* `remove(i)` *operations results in a total of $O(m)$ time spent during all calls to* `grow()` *and* `shrink()`*.*

The space (measured in words)[3] *used by a* `RootishArrayStack` *that stores* $\mathtt{n}$ *elements is* $\mathtt{n}+O(\sqrt{\mathtt{n}})$*.*

2.6.4 Computing Square Roots

A reader who has had some exposure to models of computation may notice that the `RootishArrayStack`, as described above, does not fit into the usual word-RAM model of computation (Section 1.4) because it requires taking square roots. The square root operation is generally not considered a basic operation and is therefore not usually part of the word-RAM model.

In this section, we show that the square root operation can be implemented efficiently. In particular, we show that for any integer $\mathtt{x}\in\{0,\ldots,\mathtt{n}\}$, $\lfloor\sqrt{\mathtt{x}}\rfloor$ can be computed in constant-time, after $O(\sqrt{\mathtt{n}})$ preprocessing that creates two arrays of length $O(\sqrt{\mathtt{n}})$. The following lemma shows that we can reduce the problem of computing the square root of $\mathtt{x}$ to the square root of a related value $\mathtt{x}'$.

Lemma 2.3. *Let $\mathtt{x}\geq 1$ and let $\mathtt{x}'=\mathtt{x}-a$, where $0\leq a\leq\sqrt{\mathtt{x}}$. Then $\sqrt{x'}\geq\sqrt{\mathtt{x}}-1$.*

Proof. It suffices to show that

$$\sqrt{\mathtt{x}-\sqrt{\mathtt{x}}}\geq\sqrt{\mathtt{x}}-1 \quad .$$

Square both sides of this inequality to get

$$\mathtt{x}-\sqrt{\mathtt{x}}\geq\mathtt{x}-2\sqrt{\mathtt{x}}+1$$

and gather terms to get

$$\sqrt{\mathtt{x}}\geq 1$$

which is clearly true for any $\mathtt{x}\geq 1$. □

[3]Recall Section 1.4 for a discussion of how memory is measured.

Start by restricting the problem a little, and assume that $2^{r} \le x < 2^{r+1}$, so that $\lfloor \log x \rfloor = r$, i.e., x is an integer having $r+1$ bits in its binary representation. We can take $x' = x - (x \bmod 2^{\lfloor r/2 \rfloor})$. Now, x' satisfies the conditions of Lemma 2.3, so $\sqrt{x} - \sqrt{x'} \le 1$. Furthermore, x' has all of its lower-order $\lfloor r/2 \rfloor$ bits equal to 0, so there are only

$$2^{r+1-\lfloor r/2 \rfloor} \le 4 \cdot 2^{r/2} \le 4\sqrt{x}$$

possible values of x'. This means that we can use an array, sqrttab, that stores the value of $\lfloor \sqrt{x'} \rfloor$ for each possible value of x'. A little more precisely, we have

$$\mathtt{sqrttab}[i] = \left\lfloor \sqrt{i 2^{\lfloor r/2 \rfloor}} \right\rfloor .$$

In this way, sqrttab$[i]$ is within 2 of $\sqrt{x}$ for all $x \in \{i2^{\lfloor r/2 \rfloor}, \ldots, (i+1)2^{\lfloor r/2 \rfloor} - 1\}$. Stated another way, the array entry s = sqrttab[x>>$\lfloor r/2 \rfloor$] is either equal to $\lfloor \sqrt{x} \rfloor$, $\lfloor \sqrt{x} \rfloor - 1$, or $\lfloor \sqrt{x} \rfloor - 2$. From s we can determine the value of $\lfloor \sqrt{x} \rfloor$ by incrementing s until $(s+1)^2 > x$.

FastSqrt
```
int sqrt(int x, int r) {
  int s = sqrtab[x>>r/2];
  while ((s+1)*(s+1) <= x) s++; // executes at most twice
  return s;
}
```

Now, this only works for $x \in \{2^{r}, \ldots, 2^{r+1} - 1\}$ and sqrttab is a special table that only works for a particular value of $r = \lfloor \log x \rfloor$. To overcome this, we could compute $\lfloor \log n \rfloor$ different sqrttab arrays, one for each possible value of $\lfloor \log x \rfloor$. The sizes of these tables form an exponential sequence whose largest value is at most $4\sqrt{n}$, so the total size of all tables is $O(\sqrt{n})$.

However, it turns out that more than one sqrttab array is unnecessary; we only need one sqrttab array for the value $r = \lfloor \log n \rfloor$. Any value x with $\log x = r' < r$ can be *upgraded* by multiplying x by $2^{r-r'}$ and using the equation

$$\sqrt{2^{r-r'} x} = 2^{(r-r')/2}\sqrt{x} .$$

The quantity $2^{r-r'}x$ is in the range $\{2^{r}, \ldots, 2^{r+1} - 1\}$ so we can look up its square root in sqrttab. The following code implements this idea to compute $\lfloor\sqrt{x}\rfloor$ for all non-negative integers x in the range $\{0,\ldots,2^{30}-1\}$ using an array, sqrttab, of size 2^{16}.

FastSqrt
```
int sqrt(int x) {
  int rp = log(x);
  int upgrade = ((r-rp)/2) * 2;
  int xp = x << upgrade;  // xp has r or r-1 bits
  int s = sqrtab[xp>>(r/2)] >> (upgrade/2);
  while ((s+1)*(s+1) <= x) s++;  // executes at most twice
  return s;
}
```

Something we have taken for granted thus far is the question of how to compute $r' = \lfloor\log x\rfloor$. Again, this is a problem that can be solved with an array, logtab, of size $2^{r/2}$. In this case, the code is particularly simple, since $\lfloor\log x\rfloor$ is just the index of the most significant 1 bit in the binary representation of x. This means that, for $x > 2^{r/2}$, we can right-shift the bits of x by $r/2$ positions before using it as an index into logtab. The following code does this using an array logtab of size 2^{16} to compute $\lfloor\log x\rfloor$ for all x in the range $\{1,\ldots,2^{32}-1\}$.

FastSqrt
```
int log(int x) {
  if (x >= halfint)
    return 16 + logtab[x>>>16];
  return logtab[x];
}
```

Finally, for completeness, we include the following code that initializes logtab and sqrttab:

FastSqrt
```
void inittabs() {
  sqrtab = new int[1<<(r/2)];
  logtab = new int[1<<(r/2)];
  for (int d = 0; d < r/2; d++)
    Arrays.fill(logtab, 1<<d, 2<<d, d);
  int s = 1<<(r/4);                    // sqrt(2^(r/2))
  for (int i = 0; i < 1<<(r/2); i++) {
    if ((s+1)*(s+1) <= i << (r/2)) s++; // sqrt increases
    sqrtab[i] = s;
  }
}
```

To summarize, the computations done by the i2b(i) method can be implemented in constant time on the word-RAM using $O(\sqrt{n})$ extra memory to store the sqrttab and logtab arrays. These arrays can be rebuilt when n increases or decreases by a factor of two, and the cost of this rebuilding can be amortized over the number of add(i,x) and remove(i) operations that caused the change in n in the same way that the cost of resize() is analyzed in the ArrayStack implementation.

2.7 Discussion and Exercises

Most of the data structures described in this chapter are folklore. They can be found in implementations dating back over 30 years. For example, implementations of stacks, queues, and deques, which generalize easily to the ArrayStack, ArrayQueue and ArrayDeque structures described here, are discussed by Knuth [46, Section 2.2.2].

Brodnik *et al.* [13] seem to have been the first to describe the RootishArrayStack and prove a $\sqrt{n}$ lower-bound like that in Section 2.6.2. They also present a different structure that uses a more sophisticated choice of block sizes in order to avoid computing square roots in the i2b(i) method. Within their scheme, the block containing i is block $\lfloor \log(i+1) \rfloor$, which is simply the index of the leading 1 bit in the binary representation of $i+1$. Some computer architectures provide an instruction for computing the index of the leading 1-bit in an integer.

A structure related to the RootishArrayStack is the two-level *tiered-vector* of Goodrich and Kloss [35]. This structure supports the get(i,x) and set(i,x) operations in constant time and add(i,x) and remove(i) in $O(\sqrt{n})$ time. These running times are similar to what can be achieved with the more careful implementation of a RootishArrayStack discussed in Exercise 2.11.

Exercise 2.1. In the ArrayStack implementation, after the first call to remove(i), the backing array, a, contains $n+1$ non-null values despite the fact that the ArrayStack only contains n elements. Where is the extra non-null value? Discuss any consequences this non-null value might have on the Java Runtime Environment's memory manager.

Exercise 2.2. The List method addAll(i,c) inserts all elements of the Collection c into the list at position i. (The add(i,x) method is a special case where $c = \{x\}$.) Explain why, for the data structures in this chapter, it is not efficient to implement addAll(i,c) by repeated calls to add(i,x). Design and implement a more efficient implementation.

Exercise 2.3. Design and implement a *RandomQueue*. This is an implementation of the Queue interface in which the remove() operation removes an element that is chosen uniformly at random among all the elements currently in the queue. (Think of a RandomQueue as a bag in which we can add elements or reach in and blindly remove some random element.) The add(x) and remove() operations in a RandomQueue should run in constant time per operation.

Exercise 2.4. Design and implement a Treque (triple-ended queue). This is a List implementation in which get(i) and set(i,x) run in constant time and add(i,x) and remove(i) run in time

$$O(1+\min\{i, n-i, |n/2-i|\}) \ .$$

In other words, modifications are fast if they are near either end or near the middle of the list.

Exercise 2.5. Implement a method rotate(a,r) that "rotates" the array a so that a[i] moves to a[(i + r) mod a.length], for all $i \in \{0,\ldots,a.length\}$.

Exercise 2.6. Implement a method rotate(r) that "rotates" a List so that list item i becomes list item $(i+r) \bmod n$. When run on an ArrayDeque, or a DualArrayDeque, rotate(r) should run in $O(1+\min\{r, n-r\})$ time.

Exercise 2.7. Modify the ArrayDeque implementation so that the shifting done by add(i,x), remove(i), and resize() is done using the faster System.arraycopy(s,i,d,j,n) method.

Exercise 2.8. Modify the ArrayDeque implementation so that it does not use the % operator (which is expensive on some systems). Instead, it should make use of the fact that, if a.length is a power of 2, then

$$\mathtt{k\%a.length} = \mathtt{k\&(a.length} - 1) \ .$$

(Here, & is the bitwise-and operator.)

Exercise 2.9. Design and implement a variant of ArrayDeque that does not do any modular arithmetic at all. Instead, all the data sits in a consecutive block, in order, inside an array. When the data overruns the beginning or the end of this array, a modified rebuild() operation is performed. The amortized cost of all operations should be the same as in an ArrayDeque.
Hint: Getting this to work is really all about how you implement the rebuild() operation. You would like rebuild() to put the data structure into a state where the data cannot run off either end until at least n/2 operations have been performed.

Test the performance of your implementation against the ArrayDeque. Optimize your implementation (by using System.arraycopy(a,i,b,i,n)) and see if you can get it to outperform the ArrayDeque implementation.

Exercise 2.10. Design and implement a version of a RootishArrayStack that has only $O(\sqrt{\mathtt{n}})$ wasted space, but that can perform add(i,x) and remove(i,x) operations in $O(1+\min\{\mathtt{i},\mathtt{n}-\mathtt{i}\})$ time.

Exercise 2.11. Design and implement a version of a RootishArrayStack that has only $O(\sqrt{\mathtt{n}})$ wasted space, but that can perform add(i,x) and remove(i,x) operations in $O(1+\min\{\sqrt{\mathtt{n}},\mathtt{n}-\mathtt{i}\})$ time. (For an idea on how to do this, see Section 3.3.)

Exercise 2.12. Design and implement a version of a RootishArrayStack that has only $O(\sqrt{\mathtt{n}})$ wasted space, but that can perform add(i,x) and remove(i,x) operations in $O(1+\min\{\mathtt{i},\sqrt{\mathtt{n}},\mathtt{n}-\mathtt{i}\})$ time. (See Section 3.3 for ideas on how to achieve this.)

Exercise 2.13. Design and implement a CubishArrayStack. This three level structure implements the List interface using $O(n^{2/3})$ wasted space. In this structure, get(i) and set(i,x) take constant time; while add(i,x) and remove(i) take $O(\mathtt{n}^{1/3})$ amortized time.