An Encyclopædia of Elementary Data Structures

John C. Lusth and Nicholas A. Kraft

Revision Date: August 18, 2014

Contents

Chapter 1

Notes on Terminology

This document assumes familarity with objects and the object syntax of C-based languages. Data structures are assumed to be implemented as objects. The 'dot' operator is used to access a component of an object for either retrieval or updating, depending on context.

The typical structure of a data structure is given using a class specification, as in:

```
class fillable-array
    {
    var store = ?;          //points to a normal array
    var capacity = ?;       //set by the constructor
    var size = 0;
    //operations
    ...
    }
```

If a component is initialized to `?`, its initial value is expected to be set by the constructor. Otherwise, the initialized value is the default value for the component. Operations refer to the methods of the class/object.

The explanation of an operator may use a phrase such as "the given item". Anything thus modified with the word *given* is expected to be passed as an argument to the operator. No error checking is shown in pseudo-code.

Both clientship and inheritance are used to base one data structure upon another. If inheritance is used, operators/methods in the subclass are marked with "(override)" if they override operators/methods in the superclass.

Chapter 2

An Introduction to Order Notation

Order notation is used to describe the run time of an algorithm.[1] in a big-picture sort of way. We often use order notation to compare two algorithms or to compare a single algorithm to some benchmark. For example, we might say:

$$f = O(n^2)$$

This compares algorithm f to a benchmark named n^2 and can be read as "algorithm f runs no worse than the square of the input size (times some constant) for arbitrary input above a certain problem size n". From this reading, we can deduce that n refers to the problem size and that big Oh (actually Omicron) refers to the concept *less than or equal to*.

There are a number of letters from the Greek alphabet used to make these comparisons. They are: Θ (theta), which means EQUIVALENT TO, o (little omicron), which means LESS THAN, ω (little omega), which means GREATER THAN, O (big omicron), which means LESS THAN OR EQUIVALENT TO, and Ω (big omega), which means GREATER THAN OR EQUIVALENT TO. The adjectives *less* and *greater* are general terms whose meaning depends on the particular aspect of the program being investigated. If how fast a program runs is of interest, then less means "takes less time to complete (runs faster)". If how much space a program takes up when running is of interest, then less means "uses less space". For all these symbols, it is generally implied that the definitions hold only for the worst case and for sufficiently large input. Also, unless space or some other quantity is specified, time is assumed.

As an example, consider two functions f and g that perform the same task. Consider also data sets that cause the functions f and g to exhibit worst case behavior (i.e., take the most time or use the most space). Under these conditions, the phrase $f = \omega(g)$ can be read as f is ALWAYS GREATER THAN g in the worst case and for sufficiently large input. It can also be read as g is ALWAYS LESS THAN THAN f for sufficiently large input in the worst case. Suppose we are concerned with execution speed. If indeed $f = \omega(g)$, then we can say without a doubt that function f will run slower than function g on the same data set, *provided* the data set is large enough and the data set forces both f and g to exhibit their worst case behavior. That is, as the sizes of these worst case data sets grow, the ratio of g's running time to f's running time will become smaller and smaller. Eventually, the data set sizes becomes large enough that g will run faster than f in an absolute sense (on a worst case data set) and the ratio of running times will be less than 1. Moreover, this ratio will diminish towards zero as the data sets get ever larger.

2.1 Comparing algorithms against benchmarks

Here's another example. Let's say that by counting steps we see that function r takes exactly n^2 steps in the worst case where n is representative of the input size. We would say that $r = \Theta(n^2)$. If another function

[1] In this document, the terms algorithm, function, and program are used interchangeably

s compares to r this way: $s = \omega(r)$, then we can also say that $s = \omega(n^2)$ as well. This means that the number of steps that s takes is larger (by a non-constant amount). How much larger? This is unspecified, but example step counts might be n^3 or $n^2 logn$ or 2^n. All of these counts are $\omega(n^2)$.

The common benchmarks against which programs are often compared are:

benchmark	*meaning*
1	constant
$\log n$	logarithmic
n	linear
n^2	quadratic
n^3	cubic
$n \log n$	log-linear
2^n	exponential
$n!$	factorial

2.2 Constants and lower order terms

When using order notation, we ignore constants and lower order terms. Suppose an program f takes time $4n^2 - 3n + 105$. For an input size $n = 4$, then f takes 157 units of time to complete. Suppose a program g takes time $5n^2 + 10n + 2$. For an input size $n = 4$, g takes 122 units of time. Although g runs faster than f for small inputs and f runs faster than g for larger inputs, both f and g are $\Theta(n^2)$. The reason is g is never slower than twice the run time of f. When $n = 100$, f takes 39,805 units of time, while g takes 51,002 units of time. When $n = 10,000$, f takes $3.99701 * 10^8$ units of time, while g takes $5.00100 * 10^8$ units of time. Regardless of how big n gets, g never takes more time than twice the time f takes. Since their run times are within a constant factor of each other (that constant is somewhat less than 2), that means that:

$$f = \Theta(g)$$

We can see that this is true by ignoring constants and lower order terms. The program f starts out as:

$$f = \Theta(4n^2 - 3n + 105)$$

Removing lower order terms yields:

$$f = \Theta(4n^2)$$

Ignoring constant factors yields:

$$f = \Theta(n^2)$$

By the same token

$$g = \Theta(5n^2 + 1n + 2)$$
$$g = \Theta(5n^2)$$
$$g = \Theta(n^2)$$

Since both f and g are $\Theta(n^2)$, they are Θ of each other.

2.3 What order notation does not mean

Suppose we have two algorithms, f and g, and that:

$$f = o(g)$$

While f will run faster than g when the input size is large enough and the input causes both f and g to exhibit worst case behavior, it does *not* mean that f runs faster than g in *all* cases. There may be small worst case data sets for which g runs faster. There may be rather large data sets for which f exhibits worst case behavior and g does not, leading g to again have a faster running time. The function f runs faster than g ONLY in the worst case *and* ONLY for sufficiently large inputs. A common mistake is to make the assumption that order notation holds for all inputs; it does not.

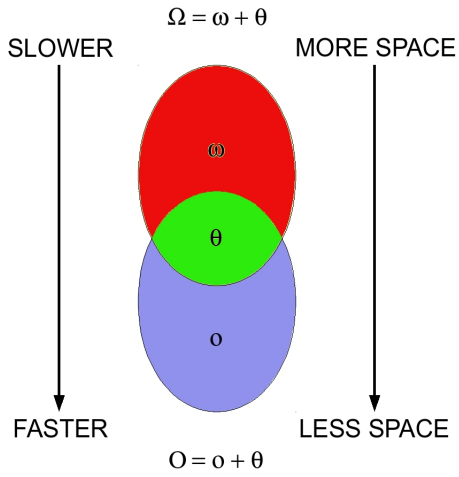
Suppose instead that:

$$f = \Theta(g)$$

Does this mean that f has exactly the same running time as g? Again, no, not really. What this does say is that f and g have running times within a constant factor of each other when worst case behavior is exhibited with sufficiently large inputs.

2.4 A pictorial view

With the following Venn diagram, the definitions of the symbols used for order notation will become more intuitive (hopefully):



Suppose:

$$f = \Omega(g)$$

Start by placing g in the Θ region. Since f is Ω, f can be placed in either the Θ region or the ω region. Thus, f runs within a constant factor of g or slower, for sufficiently large input that forces worst case behavior.

Chapter 3

Arrays

3.1 Synopsis

An array is a fundamental data structure that supports the getting and setting of values. The size of an array dictates how many values it can hold. An array of size n can hold n values. Arrays are distinguished by the fact that getting or setting any element in the array can be done in constant time, regardless of the size of the array.

Arrays may be *homogeneous* or *heterogeneous*, depending on the language. In a homogeneous array, all the stored values must have the same type (*e.g.* all the elements are integers or all are strings). In a heterogeneous array, the values do not need to be of the same type (*e.g.* the array can hold both integers and strings).

3.2 Structure

Arrays are typically built in to the language.

3.3 Vocabulary

The terms used in describing an array are:

element A value stored in an array. Each element is associated with a unique *index*. Elements are said to be *stored* at an index.

first element The element with the lowest index. Also known as the *leftmost* element.

last element The element with the highest index. Also known as the *rightmost* element.

index An index is an integer value used to indicate a particular element in an array. An array of size n has n indices and holds n elements. In modern programming languages, a technique known as *zero-based counting* is used to number the elements. Instead of numbering the elements from leftmost to rightmost as 1, 2, 3, 4 ..., the elements are numbered 0, 1, 2, 3, ... Thus, the first element in the array has index 0, the second, index 1, the third, index 2, and so on.

length The number of elements in the array. Also known as the *capacity*.

3.4 Supports

Arrays can be used to implement:

- Fillable Arrays
- Heaps

3.5 Dependencies

Arrays are usually fundamental data structures provided by the language processor.

3.6 Operations

3.6.1 get

Languages that support arrays often provide special syntax for getting an element of an array. For C-based languages, square brackets are used to indicate which element is to be gotten; the index of the desired element is placed between the brackets. The array the element is to be retrieved from is given immediately before the opening square bracket. Here are some examples, with *items* being the name of the array:

```
println(items[0]);                //print the first item
last = items[length(items) - 1]   //get the last item, assign to last
```

Functional languages may use function calls to get elements from an array:

```
println(array-get(items,0))
set(last,array-get(items,length(items) - 1));
```

Getting an element takes $\Theta(1)$ time.

3.6.2 set

Setting an element in an array overwrites the previous value. Getting an element at an index always gets the value most recently set at that index. In C-based languages, setting looks like getting, except the array name, brackets, and index appear to the left of an assignment operator. Here are some examples, with *items* being the name of the array:

```
items[0] = items[1];            //set the first element to the second
items[length(items) - 1] = 0;   //set the last element to zero
```

Functional languages may use function calls to set elements from an array:

```
array-set(items,0,array-get(items,1))
array-set(items,length(items) - 1,0)
```

Setting an element takes $\Theta(1)$ time.

3.7 Traversals

Arrays are usually traversed from left to right:

```
for (i = 0; i < length(items); i += 1) //zero-based counting!
    println(items[i])
```

However, it is not uncommon to traverse them from right to left:

```
for (i = length(items) - 1; i >= 0; i -= 1) //zero-based counting!
    println(items[i])
```

3.8 Concept Inventory

Array Concept Inventory (C Language)

1. Consider a small array *a* and large array *b*. Accessing the first element of *a* takes more/less/the same amount of time as accessing the first element of *b*.
   (a) the same amount of time
   (b) more time
   (c) less time
2. Consider a small array *a* and large array *b*. Accessing the last element of *a* takes more/less/the same amount of time as accessing the first element of *b*.
   (a) the same amount of time
   (b) more time
   (c) less time
3. C arrays are:
   (a) homogeneous
   (b) heterogeneous
   (c) neither homogeneous nor heterogeneous
   (d) both homogeneous and heterogeneous
4. C arrays can be:
   (a) dynamically allocated only
   (b) statically allocated only
   (c) dynamically and statically allocated
   (d) neither dynamically nor statically allocated
5. Why is the following array declaration not proper?

   ```
   int z1[];
   ```

   (a) it *is* proper
   (b) the square brackets are in the wrong place
   (c) the size of the array is missing
   (d) *z1* is not a legal variable name
6. Why is the following array declaration not proper?

```
int _[] = { 1 };
```

(a) it *is* proper
(b) arrays have to have more than one slot
(c) the size of the array is missing
(d) _ is not a legal variable name

7. Why is the following array declaration not proper?

```
int[1] x = { 1 };
```

(a) it *is* proper
(b) arrays have to have more than one slot
(c) the size of the array is missing
(d) the brackets are in the wrong place

8. Why is the following array declaration improper?

```
int y[2] = { 1, 2, 3 };
```

(a) it *is* proper, the array is expanded to three elements
(b) it *is* proper, the last initializer is ignored
(c) there are too many initializers
(d) there are too few initializers

9. Why is the following array declaration improper?

```
int w = { 1, 2, 3 };
```

(a) it *is* proper
(b) there are too many initializers
(c) there are too few initializers
(d) the square brackets are missing

10. Why is the following array declaration improper?

```
double y[3] = { 1, 2, 3 };
```

(a) it *is* proper
(b) you cannot have integer initializers
(c) you cannot give a size if you have initializers
(d) there should be a comma after the 3

11. Why is the following array declaration improper?

```
double y[1];
```

(a) it *is* proper
(b) initializers are missing
(c) arrays need more than one slot
(d) there should be an empty initializer list

12. Why is the following array declaration improper?

```
double y[2] = {};
```

    (a) it *is* proper
    (b) there should be initializer values
    (c) the initializer list should be `{,}`
    (d) the initializer list should be `{}`

13. Why is the following array declaration improper?

```
double y[3] = { 1.1 2.2 3.3 };
```

    (a) it *is* proper, commas are not necessary
    (b) commas after each initializer are missing
    (c) commas between initializers are missing

14. Why is the following array declaration improper?

```
double y[3] = { "1", "2", "3" };
```

    (a) it *is* proper
    (b) you cannot have string initializers
    (c) you cannot give a size if you have initializers
    (d) there should be a comma after the 3

15. Why is the following array declaration improper?

```
void y[2];
```

    (a) it *is* proper
    (b) `void` is not a valid type for arrays
    (c) the initializer is missing

16. Why is the following array declaration improper?

```
void *y[2];
```

    (a) it *is* proper
    (b) `void *` is not a valid type for arrays
    (c) the initializer is missing

17. **T** or **F**: Given the declaration `int x[3];` and `int y[3];`, then `y[0] = x[0];` is a logically correct assignment.

18. **T** or **F**: Given the declaration `int x[3];` and `int y[3];`, then `y[0] = x[1];` is a logically correct assignment.

19. **T** or **F**: Given the declaration `int x[3];` and `int y[3];`, then `y[0] = x[3];` is a logically correct assignment.

20. **T** or **F**: Given the declaration `int x[3];` and `int y[3];`, then `y[3] = x[2];` is a logically correct assignment.

21. An array has size n. What is the index of the last element?

    (a) n

    (b) $n-1$

    (c) $n+1$

    (d) $n-2$

22. You wish to set change the first element in an array a to the value of the last element. Which of the following accomplishes that task?

    (a) `a[0] = a[n-1];`

    (b) `a[1] = a[n-1];`

    (c) `a[1] = a[n];`

    (d) `a[0] = a[n];`

23. You know that the second element of array a holds a valid index for a. You wish to change the element at that index to zero. Which of the following reliably accomplishes that task?

    (a) `a[a[1]] = 0;`

    (b) `a = a[a[2]];`

    (c) `a[a[2]] = 0;`

    (d) `a[a[1]] = a[1];`

24. Consider the declaration `int y[3];`. As an rvalue, the expression `y[2]`:

    (a) generates an out-of-bounds error message

    (b) references the last element in the array

    (c) may cause the program to crash

    (d) will always cause the program to crash

25. Consider the declaration `int y[3];`. As an rvalue, the expression `y[3]`:

    (a) generates an out-of-bounds error message

    (b) references the last element in the array

    (c) may cause the program to crash

    (d) will always cause the program to crash

26. Consider the declaration `int y[3];`. As an rvalue, the expression `y[4]`:

    (a) generates an out-of-bounds error message

    (b) references the last element in the array

    (c) may cause the program to crash

    (d) will always cause the program to crash

27. Consider the declaration `int y[3];`. The proper declaraction for a pointer that can point to array y is:

    (a) `int *z;`

    (b) `int z;`

    (c) `int z(*[3]);`

    (d) `int *z[3];`

28. Consider the declaration `char *y[3];`. The proper declaraction for a pointer that can point to array y is:

    (a) `char *z;`

    (b) `char **z;`

    (c) `char *z(*[3]);`

    (d) `char *z[3];`

29. Suppose z points to array a. To access the first element of a using z, one would use the expression:

    (a) `z[0]`

    (b) `z[1]`

    (c) `*z[0]`

    (d) `*z[1]`

    (e) `z+0`

    (f) `z+1`

30. Consider the declaration `int *y[3];`. To make the assignment `p = y;` legal, the declaration of p would have to be:

    (a) `int ***p;`

    (b) `int **p;`

    (c) `int *p;`

    (d) `int p;`

31. Consider the declaration `int **y[3];`. To make the assignment `p = y;` legal, the declaration of p would have to be:

    (a) `int ***p;`

    (b) `int **p;`

    (c) `int *p;`

    (d) `int p;`

32. Consider the declaration `int y[3];`. To make the assignment `p = y;` legal, the declaration of p would have to be:

    (a) `int ***p;`

    (b) `int **p;`

    (c) `int *p;`

    (d) `int p;`

33. Consider the declaration `char *y[3];`. To make the assignment `p[0] = y;` legal, the declaration of p would have to be:

    (a) `char ***p;`

    (b) `char **p;`

    (c) `char *p;`

    (d) `char p;`

34. Consider the declaration `char y[3];`. To make the assignment `p[0] = y;` legal, the declaration of p would have to be:

(a) `char ***p;`

(b) `char **p;`

(c) `char *p;`

(d) `char p;`

35. Consider the declaration `char **y[3];`. To make the assignment `p[0] = y[0];` legal, the declaration of *p* would have to be:

(a) `char ***p;`

(b) `char **p;`

(c) `char *p;`

(d) `char p;`

36. Consider the declaration `char *y[3];`. To make the assignment `p[0] = y;` legal, the declaration of *p* would have to be:

(a) `char ***p;`

(b) `char **p;`

(c) `char *p;`

(d) `char p;`

37. **T** or **F**: Given the declaration `int x[3];` and `int y[3];`, then `x = y;` will cause a compiler error.

38. **T** or **F**: Given the declaration `int x[3];` and `int y[3];`, then `y = x;` will cause a compiler error.

39. **T** or **F**: Given the declaration `int x[3];` and `int y[3];`, then `y = x[0];` will cause a compiler error.

40. **T** or **F**: Given the declaration `int x[3];` and `int y[3];`, then `y[0] = x;` will cause a compiler error.

41. Consider the declaration `int z[3];`. The name *z* is:

(a) a pointer

(b) a pseudopointer

(c) neither a pointer nor a pseudopointer

42. **T** or **F**: You can assign a pointer to a pseudopointer.

43. **T** or **F**: You can assign a pseudopointer to a pseudopointer.

44. **T** or **F**: You can assign a pointer to a pointer.

45. **T** or **F**: You can assign a pseudopointer to a pointer.

46. Using the *sizeof* operator on a pseudopointer to an array gives you:

(a) the number of slots in the array

(b) the number of bytes in the array

(c) the size of the pseudopointer

(d) you are not allowed to use *sizeof* on a pseudopointer

47. Using the *sizeof* operator on a pointer to an array gives you:

(a) the number of slots in the array

(b) the number of bytes in the array

(c) the size of the pointer

(d) you are not allowed to use *sizeof* on a pointer

48. Consider:

```
int a[3];
int *p = a;
```

The value stored at the memory location associated with p is:

(a) the address of the first slot of the array
(b) the name a
(c) the value of the first element of the array
(d) undefined, since the operation is illegal

49. Consider:

```
int a[3];
int *p = a;
```

The value stored at the memory address found in p is:

(a) the address of the first slot of the array
(b) the name a
(c) the value of the first element of the array
(d) undefined, since the operation is illegal

50. The array access `a[3]` can be rewritten as:

(a) `*(a + 2)`
(b) `*(a + 3)`
(c) `*(a) + 2`
(d) `*(a) + 3`

51. The pointer access `*(a+4)` can be rewritten as:

(a) `a[3]`
(b) `a[4]`
(c) `*(a[3])`
(d) `*(a[4])`

52. The array access `a[0]` cannot be rewritten as:

(a) `*(a + 2) - 2`
(b) `*(a + 2 - 2)`
(c) `*(a + 0)`
(d) `*a`

53. Consider:

```
int a[] = { 10, 100, 100 };
int *p = a;
```

The expression $p + 1$:

(a) points to the first slot of a

(b) points to the second slot of a

(c) evaluates to 11

(d) evaluates to 101

54. Consider:

```
int a[] = { 10, 100, 100 };
int *p = a;
```

The expression $p + 2$:

(a) points to the first memory location beyond a

(b) points to the third slot of a

(c) evaluates to 101

(d) evaluates to 1001

55. Consider:

```
int a[] = { 10, 100, 100 };
int *p = a;
```

The expression $p + 3$:

(a) points to the first memory location beyond a

(b) points to the third slot of a

(c) evaluates to 1001

(d) would generate an error by the compiler

56. The standard library function that can be used to dynamically allocate an array is called:

(a) *malloc*

(b) *allocate*

(c) *dallocate*

(d) *dymalloc*

57. The string `"dog"` is stored as:

(a) an array of three characters

(b) an array of four characters

(c) a single location in memory

(d) three non-contiguous memory locations

58. A pointer to the string `"rat"` has the type:

(a) char

(b) char *

(c) char [3]

(d) string

(e) string *

59. Consider a pointer p to the string `"bat"`. What is the rvalue of `p[1]`?

(a) the second memory location in the string
(b) the letter `'b'`
(c) the letter `'a'`
(d) the first memory location in the string

60. Consider a pointer p to the string `"bat"`. What is the lvalue of `p[1]`?

(a) the second memory location in the string
(b) the letter `'b'`
(c) the letter `'a'`
(d) the first memory location in the string

61. Consider a pointer p to the string `"bat"`. What is the rvalue of `p[3]`?

(a) unknown
(b) the letter `'t'`
(c) the null character
(d) the last memory location in the string

62. Consider a pointer p to the string `"bat"`. What is the lvalue of `p[4]`?

(a) the first memory location beyond the string
(b) the second memory location beyond the string
(c) unknown
(d) the null character

Chapter 4

Fillable Arrays

4.1 Synopsis

Conceptually, a fillable array is composed of slots, which are either empty or filled. A filled slot holds an element. Each index of the array is associated with a slot and slots are filled preferentially from left to right. Thus, if zero-based counting is used, the first slot to be filled has index 0, the second has index one and so on.

Practically, a fillable array is usually implemented as an object with a normal array as a component. This array component is typically known as a *store*, since the values stored in a fillable array are actually stored in the component array.

4.2 Structure

The typical fillable array object has three components, plus operators:

```
class fillable-array
    {
    var store = ?;          //points to a normal array
    var capacity = ?;       //set by the constructor
    var size = 0;
    //operations
    ...
    }
```

The *store* component points to a normal array and is set by the constructor. The *capacity* is usually given as an argument to the constructor and is used to allocate the *store*.

4.3 Vocabulary

The terms used in describing a fillable array are:

store The actual array where elements are stored.

slot A spot in the store which may or may not hold an element. Each spot is associated with a unique *index*.

empty slot A spot in the store that does not (yet) hold an element. A slot is empty if its index lies between *size* and *capacity*-1, inclusive.

filled slot A spot in the store that holds an element. A slot is filled if its index lies between 0 and *size*-1, inclusive.

capacity The maximum number of elements that can be placed in the store. Also, the length of the store array.

front The leftmost value in the store.

back The rightmost value in the store. With zero-based counting, if the array is non-empty, the back (or *rear*) element in the array has index *size* - 1.

size The number of elements placed in the store. The size ranges from 0 (the store is completely empty) to the capacity of the store (the store is completely full). Also, with zero-based counting the size of the store is also an index that points to the next available slot.

4.4 Supports

Fillable arrays can be used to implement:

- Bounded Stacks

Important fillable arrays are:

- Circular Arrays
- Dynamic Arrays

4.5 Dependencies

Fillable arrays are implemented using arrays.

4.6 Operations

4.6.1 add-to-front

This operation places the given value in the first slot of the store. This necessitates shifting all previously added elements one slot to the right. Because *add-to-front* takes takes $\Theta(n)$ time, it is rarely implemented for fillable arrays.

4.6.2 remove-from-front

This operation removes the in the first slot of the store. This necessitates shifting all previously added elements one slot to the left, in order to fill the hole left by the removed element. Because *remove-from-front* takes takes $\Theta(n)$ time, it is rarely implemented for fillable arrays.

4.6.3 add-to-back

This operation adds the given *value* to the store. The value is stored to the immediate right of the previously added values or, in other words, the leftmost empty slot. The size is also incremented to reflect the fact that a value has been added.

```
function add-to-back(value)
    {
    store[size] = value;
    size += 1;
    }
```

Unlike adding a value to the front, adding a value at the back takes $\Theta(1)$ time, since no shifting of previously added values is needed.

4.6.4 remove-from-back

This operation removes the rightmost *value* in the store. This can be done simply by decrementing the size.

Removing a value from the back takes $\Theta(1)$ time.

4.6.5 get, set

These operations are similar to those of normal arrays, but a valid index lies between 0 and *size* - 1.

4.6.6 find

The *find* operation for fillable arrays, assuming the elements are unsorted, uses a traversal as shown below.

4.7 Traversals

Like normal arrays, fillable arrays are usually traversed from left to right. However, it is important to remember that the last filled slot has index *size* - 1, so any traversal has to take this into account:

```
for (i = 0; i < size; i += 1)
    println(items[i])
```

Note that this traversal works from left to right. To insert in the middle of the array, a traversal of the upper part of the array must be performed from right to left, in order that shifting an element rightward does not trash the element to the immediate right. In a right-to-left traversal, the element to the immediate right has already been shifted rightward, leaving a hole; the element on the left is shifted into this hole, leaving a new hole. Thus, the hole appears to move leftward.

To delete from the middle of an array, a left-to-right traversal is performed.

4.8 Concept Inventory

1. Suppose a fillable array has size s and capacity c. The next value to be added to the array will be placed at index:
   (a) s
   (b) c
   (c) c - 1

(d) $c + 1$

(e) s - 1

2. Suppose for a fillable array, the size is equal to the capacity. Can a value be added?

   (a) Yes, there is room for one more value

   (b) No, the array is completely full

3. Suppose a fillable array is empty. The size of the array is:

   (a) zero

   (b) one

   (c) the length of the array

   (d) the capacity of the array.

Chapter 5

Circular Arrays

5.1 Synopsis

A circular array is like a fillable array except values can be added to and removed from the front of the store as well as the back. A consequence of this is the front element of the store may have an associated index other than zero. To avoid shifting elements when adding an element to the front of the store, indices are allowed to *roll over*. For example, suppose the capacity is 10 and the size is 5, with the first element residing at index 0 (using zero-based counting). Adding a new value to the front should place it at index -1, which does not exist. Since the index is too small (*i.e.*, negative), the capacity is added to the index to make it non-negative. For our example, the new index is -1 plus 10, or 9. Thus the new element is added at index 9, the rightmost slot. Conversely, with the same size and capacity as before, but with the index of the first element being 5, consider adding an element to the back. In this case, the index of the most rightmost element is 9, because the size is 5 (the five values are stored at indices 5, 6, 7, 8, and 9). Adding an element at the back would place it at index 10, which does not exist. Since the index is too large, the proper index is then computed by subtracting off the capacity. The places the to-be-added element at index 0. The tests for indices being too small or too large can be eliminated by the use of modular arithmetic.

As with fillable arrays circular arrays use a normal array as a store.

5.2 Structure

The typical circular array object has three components plus a set of operators:

```
class circular-array
    {
    var store = ?;      //points to an array
    var capacity = ?;   //length of the store
    var size = 0;       //net number of elements added
    var startIndex = 0; //index of the first/leftmost element
    var endIndex = 0;   //optional: index of the first open slot
    //operations
    ...
    }
```

The *store* component points to a normal array and is set by the constructor. The *capacity* is usually given as an argument to the constructor and is used to allocate the *store*. Both the start and end indices are set to zero; the only case when these indices are equal is when the circular array is empty.

5.3 Vocabulary

The terms used in describing a circular array, in addition to those of normal arrays, are:

start index The index of the first element in the store. When computing a new start index, if the result is too large, it is corrected by subtracting off the capacity. If it becomes to small (*e.g.*, negative for zero-based counting), it is corrected by adding in the capacity.

end index The index of the first open slot in the store. It is computed dynamically by adding the size to the start index. If the generated index is too large, it is corrected by subtracting off the capacity. The end index may also be cached as a component of the data structure.

index correction The process of correcting an index if it is too small or too large. A convenient method of correction uses modular arithmetic.

modular arithmetic A calculated index, whether it be too small or two large, can be corrected by adding in the capacity and then *modding* by the capacity. If the calculated index was neither too small or too large, the operations of adding in the capacity and then modding has no effect.

5.4 Supports

Circular arrays can be used to implement:

- Bounded Stacks with a *least-recently-pushed* replacement strategy
- Bounded Queues

Circular arrays are used as a basis for:

- Dynamic Circular Arrays

5.5 Dependencies

Circular arrays are implemented using arrays, with similarities to fillable arrays.

5.6 Operations

5.6.1 decrement-index

This operation subracts one from the given index. If the resulting value becomes too small, the value is corrected by adding in the capacity. In any case, the new, perhaps corrected, value is returned.

This operation is generally a private method.

5.6.2 increment-index

Like *decrement-index*, except one is added to the given index (with possible correction).

This operation is generally a private method.

5.6.3 correct

The *increment* and *decrement* operations can be replaced with this operation which typically uses modular arithmetic to adjust the given index. The calls:

```
u = decrementIndex(v)
x = incrementIndex(y)
```

would be replaced with:

```
u = correctIndex(v - 1);
x = correctIndex(y + 1);
```

This operation is generally a private method.

5.6.4 add-to-front

This operation adds a given *value* to the front of the store. A special case exists if the array is empty. If so, *add-to-back* is called to handle the addition. Otherwise, the start index is decremented and corrected, if necessary.

```
function add-to-front(value)
    {
    if (size == 0)
        add-to-back(value)
    else
        {
        startIndex = decrementIndex(startIndex);
        store[startIndex] = value;
        size += 1;
        }
    }
```

Adding an element to the front takes $\Theta(1)$ time.

5.6.5 add-to-back

This operation adds a given *value* to the back of the store. Unlike fillable arrays, the next available slot may not have index *size*. The start index is added to the *size* to generate the actual index of the next available slot. The generated index is corrected, if necessary.

```
function add-to-back(value)
    {
    store[endIndex] = value;
    endIndex = incrementIndex(endIndex)
    size += 1;
    }
```

Adding an element to the back takes $\Theta(1)$ time.

5.6.6 remove-from-front

This operation removes the leftmost *value* in the store. This can be done simply by incrementing the start index (with correction). The size component is updated accordingly.

Removing the front element takes $\Theta(1)$ time.

5.6.7 remove-from-back

This operation removes the rightmost *value* in the store. This can be done simply by decrementing the size, if the end index is not cached. If the end index is cached, it needs to be decremented (with correction) as well.

Removing the back element takes $\Theta(1)$ time.

5.6.8 get, set

Like fillable arrays, a valid index for *get* and *set* lies between 0 and *size* - 1. However, the value of the start index must be added to the given index and then corrected as necessary. For example, *get* becomes:

```
function get(index)
    {
    var spot = correctIndex(index + startIndex);
    return store[spot];
    }
```

The code for *set* follows a similar pattern.

5.6.9 find

Find is similar to that of fillable arrays. Assuming the elements are unsorted, a modified linear traversal (see below) is used.

5.7 Traversals

Circular arrays support traversals similar to that of arrays, except that the actual starting (or ending) point is the start index while the actual ending (or starting point) is the end index. However, since the end index may be smaller than the start index, it becomes difficult to write a loop based upon these actual values. By basing the traversal on *get* (or *set*), then the starting (or ending) point is zero and the ending (or starting point) is the *size* - 1 and writing such a traversing loop is greatly simplified:

```
for (i = 0; i < size; i += 1)
    println(get(i));
```

A right-to-left traversal is similar.

5.8 Concept Inventory

1. In a circular array, the start index (after correction) can never equal the size.
   (a) True
   (b) False
2. In a circular array, the start index (after correction) can never equal the capacity.
3. Suppose a circular array has size s, capacity c, and start index of i. Before correction, the next value to be added to the front of the array will be placed at index:
   (a) $s + i$
   (b) s - i
   (c) i
   (d) i - 1
   (e) c - 1
   (f) c - i
4. Suppose for a circular array, the size is equal to the capacity. Can a value be added?
   (a) Yes, there is room for one more value
   (b) No, the array is completely full
5. Suppose a circular array is empty. The size of the array is:
   (a) zero
   (b) one
   (c) the length of the array
   (d) the capacity of the array.

Chapter 6

Dynamic Arrays

6.1 Synopsis

A dynamic array is a fillable array that can grow in capacity should the array become completely filled, thus making room for more additions. Conversely, a dynamic array can shrink in capacity if it becomes mostly empty.

6.2 Structure

The typical dynamic array object adds one component to those of a fillable array.

```
class dynamic-array extends fillable-array
    {
    var factor = 2;         //how much to grow or shrink the store by
    //operations
    ...
    }
```

The *factor* is typically set to two, which means growing the array doubles its capacity. Conversely, shrinking the array halves the capacity.

6.3 Vocabulary

The terms used in describing a dynamic array, in addition to those of normal arrays, are:

dynamic fillable array a more precise description of this data structure.

slot A spot in the array which may or may not hold an element. Each spot is associated with a unique *index*.

empty slot A spot in the array that does not (yet) hold an element.

filled slot A spot in the array that holds an element.

capacity The maximum number of elements that can be stored in an array. Also, the length of the array.

front The first/leftmost value in the array.

back The last/rightmost value in the array. With zero-based counting, if the array is non-empty, the back (or *rear*) element in the array has index *size* - 1.

size The number of elements stored in the array. The size ranges from 0 (the array is completely empty) to the capacity of the array (the array is completely full). Also, with zero-based counting the size of the array is also an index that points to the next available slot.

6.4 Supports

Dynamic arrays can be used to implement:

- Unbounded Stacks

They are also useful for reading in an unknown quantity of data:

```
items = new-dynamic-array();

value = readValue();
if (valid(value))
    {
    items.add-to-back(value);
    value = readValue();
    }
items.shrink-to-fit();
```

The *shrink-to-fit* is like shrink, but the operation removes all empty slots.

6.5 Dependencies

Dynamic arrays can be implemented using fillable arrays.

6.6 Operations

6.6.1 add-to-back (override)

This operation works the same as *add-to-back* for fillable arrays. The only difference is, at the very start of the operation, a check is made to see if the array is full. If it is, a *grow* operation is performed. Whether or not, the array was full at the start of *add-to-back*, there is now at least one empty slot in which the new value can be placed.

```
function add-to-back(value)
    {
    if (size == capacity)
        grow();
    store[size] = value;
    size += 1;
    }
```

This operation takes $\Theta(1)$ time if the array is not full, but takes $\Theta(n)$ time if it is, due to the expense of the *grow* operation.

6.6.2 remove-from-back (override)

As with fillable arrays, this operation removes the rightmost *value* in the array and can be done simply by decrementing the size. However, a test can be made to see if the array has become mostly empty. If so, a *shrink* operation can be performed. A good metric: when the ratio of size to capacity is less than 0.25, the array should be shrunk.

This operation takes $\Theta(n)$ time if shrinking occurs, due to the expense of the *shrink* operation. Otherwise, the operation takes $\Theta(1)$ time.

6.6.3 grow

The *grow* operation performs the following steps:

- calculate the new capacity
- allocate a new store with the new capacity
- copy the elements from the old store to the new store
- free the old store
- set the data structure's store to the new store
- update the capacity

Although it is beyond this document to explain why, computing the new capacity using a constant multiplicative factor (*e.g.*, doubling the capacity) gives far better efficiency than by using a constant additive term (*e.g.*, increasing the capacity by 100).

6.6.4 shrink

The *shrink* operation is similar to that of the *grow* operation, except that the capacity is made smaller. As with grow, computing the new capacity by dividing by a constant gives better performance than by subtracting off a constant.

A variant of *shrink* is *shrink-to-fit*, where the capacity is reduced to the size so that the resulting store is completely filled.

6.6.5 get, set, find

These operations are the same as those for (or inherited from) fillable arrays.

6.7 Traversals

Traversing a dynamic array proceeds exactly as that for a fillable array.

6.8 Concept Inventory

1. Suppose a dynamic array has size s and capacity c, with s equal to c. Is the array required to grow on the next addition?

(a) true
(b) false

Chapter 7

Dynamic Circular Arrays

7.1 Synopsis

Like dynamic arrays, dynamic circular arrays appear never to become full. When an attempt is made to add an element to a full array, the store is grown to accomodate the addition. When elements are removed, dynamic circular arrays can also shrink, if desired.

7.2 Structure

The typical dynamic circular array object adds one component to those of a circular array.

```
class dynamic-circular-array extends circular-array
    {
    var factor = 2;         //how much to grow or shrink the store by
    //operations
    ...
    }
```

As with dynamic arrays, the *factor* is typically set to two, which means growing the array doubles its capacity. Conversely, shrinking the array halves the capacity.

7.3 Vocabulary

The vocabulary of dynamic circular arrays combines the vocabularies of dynamic arrays. and circular arrays.

7.4 Supports

Dynamic circular arrays are primarily used to implement queues. They can be used to implement stacks, but the simple dynamic arrays suffice.

7.5 Dependencies

Dynamic circular arrays can be implemented with *circular arrays*.

7.6 Operations

7.6.1 incrementIndex, decrementIndex, correctIndex

These operations are the same as that of circular arrays.

7.6.2 add-to-front (override), add-to-back (override)

These routines call *grow* if the array is full, before performing the logic of the superclass version of the functions.

See the *add* operations of dynamic arrays for more information.

7.6.3 remove-from-front (override), remove-from-back (override)

These routines call *grow*, if the array becomes significantly empty, before performing the logic of the superclass version of the functions.

See the *remove* operations of dynamic arrays for more information.

7.6.4 grow, shrink

The *grow* and *shrink* operations are the same as that of dynamic arrays, but the start index is usually reset to zero. Thus, the element at the start index of the old store gets placed in the leftmost slot in the new store, and so on. When computing the index of the next element to be transferred, the *incrementIndex* or *correctIndex* functions should be used.

7.6.5 get, set, find

These operations are the same as that of circular arrays.

7.7 Traversals

Traversals over dynamic circular arrays are the same as that for circular arrays.

7.8 Concept Inventory

1. Suppose a dynamic array has size s and capacity c, with s equal to c. Is the array required to grow on the next addition?
   (a) true
   (b) false

Chapter 8

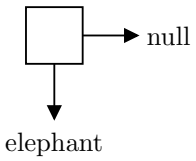
Nodes

8.1 Synopsis

A *node* is a simple data structure that bundles together a value and a set of node pointers, each of which point to a node or to nothing. At its simplest, there is but a single node pointer.

To signify that a node pointer points to nothing, a sentinal value, usually named *null* or *nil*, is stored in the variable representing the node pointer.

Pictorially, a node is often drawn using a box to represent a node with arrows representing the value and node pointers. Usually, the value pointer is drawn in the downward direction:



Here, the value is "elephant" and a single node pointer is shown, set to null.

8.2 Structure

The typical node object holds a single value pointer and some number of node pointers. As an example, when nodes are used to build a singly linked-list, a single node pointer, usually named *next*, is sufficient.

```
class node
    {
    var value = ?;          //points to the stored value
    var next = ?;           //optional: for singly/doubly linked-lists
    var prev = ?;           //optional: for doubly linked-lists
    var left = ?;           //optional: for binary trees
    var right = ?;          //optional: for binary trees
    //operations
    ...
    }
```

The initializers for *value* and the node pointer(s) are expected to be passed as arguments to the constructor.

8.3 Vocabulary

The terms used in describing a node are:

value The value stored in the node.

next In the case of a node with a single node pointer that is used to implement a singly linked-list, the node pointer is usually named *next*.

prev/next In the case of a node with two node pointers that is used to implement a doubly linked-list, the two node pointers are commonly named *prev* (for previous) and *next*.

left/right In the case of a node with two node pointers that is used to implement a binary tree, the two node pointers are often named *left* and *right*.

8.4 Supports

A node is used to implement:

- linked-lists
- linked-lists
- binary trees
- binary search trees

8.5 Dependencies

A node is a fundamental data structure often provided by the language processor or, if not provided, easily constructed using a heterogeneous array or object. A heterogeneous array can hold elements of different types (*e.g.* the first element can be an integer, the second a string, and so on). In the case of a heterogeneous array, the first slot can hold the value and subsequent slots can hold the node pointers.

8.6 Operations

8.6.1 get-value

This operation returns the value stored in the node.

8.6.2 set-value

This operation updates the value in the node with the given value.

8.6.3 get-next/get-prev/get-left/get-right

This operation returns the node that is stored in the associated node pointer.

8.6.4 set-next/set-prev/set-left/set-right

This operation updates the associated node pointer with the given node. The actual names of the operations depend on the number of node pointers.

8.7 Traversals

Nodes, having usually one value pointer, are not traversed.

8.8 Concept Inventory

1. Generally, a node pointer points to...
   (a) another node
   (b) a value
   (c) a string

Chapter 9

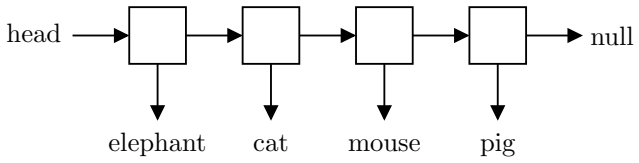
Singly Linked-Lists

9.1 Synopsis

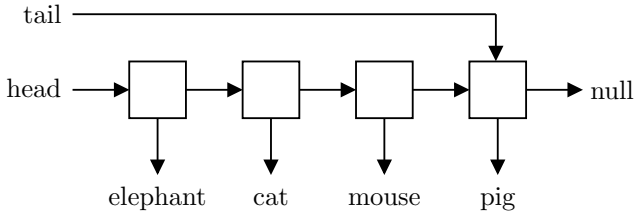
A linked-list is one of the more important data structures in computer science, having a wide application in a number of situations. Linked-listes are usually constructed from nodes, each node having a single node pointer. The nodes are chained together with this node pointer. For example, if the node pointer is named next and the variable a points to the first node in the chain, the subsequent nodes in the chain could be accessed with the expressions:

```
a.next;             //the second node
a.next.next;        //the third node
a.next.next.next;   //the fourth node
```

Pictorially, a linked-list is often represented with a box-and-pointer diagram, where each box represents a node and where a downward arrow represents a value pointer and a rightward arrow represents the next node pointer:



To improve performance when adding a new element to the rear of the list, a tail pointer can be used. The tail pointer is set to always point to the last node in the list:



9.2 Structure

```
class singly-linked-list
    {
    var head = null;        //pointer to the first/leftmost node
    var tail = null;        //optional: pointer to last/rightmost node
    var size = 0;           //optional: holds net number of nodes added
    // operations
    ...
    }
```

9.3 Vocabulary

The terms used in describing an singly linked-list are:

chain A series of nodes linked using a single node pointer

head of the list The first/leftmost value stored in the list.

tail of the list The list minus the first/leftmost node.

head pointer A component of the list that points to the first/leftmost node in the list. When the list is empty, the head pointer has a value of null. The head pointer serves as the data structure's store.

tail pointer An additional component of the list that points to the last/rightmost node in the list. When using a tail pointer, it is imperative to remember to set the tail pointer to null whenever and however a list becomes empty.

index An index is an integer value used to indicate a particular node in the list. The first node has index 0, the second has index 1, and so on.

size The number of nodes in the list. Also called *length*.

dummy head node A device to simplify the add-to-front operation. A dummy node is a node with no value. When using a dummy head node, the head pointer always points to this node, even if the list is empty. The first/leftmost node follows the dummy head node. The size of the list does not include the dummy head node.

walking a list Traversing the list, usually from the head node to the tail node.

link A node in the list is often called a *link*.

9.4 Supports

Singly linked-lists are used as a basis for:

- stacks
- pritority queues

If a tail pointer is implemented, then a singly linked-list can be used as a basis for a queue.

9.5 Dependencies

Singly linked-lists are implemented using nodes. with a single node pointer, typically named *next*.

9.6 Operations

9.6.1 add-to-front

The *add-to-front* operation is quite simple. It simply points the head to a new node whose *next* pointer points to the old head:

```
function add-to-front(value)
     {
     head = new node(value,head);
     }
```

However, if the list incorporates a dummy head node, the *add-to-front* function becomes:

```
function add-to-front(value)
     {
     head.next = new-node(value,head.next);
     }
```

If a tail pointer is used, the tail pointer must be set the newly created node if the list was empty prior to the addition.

This operation takes $\Theta(1)$ time.

9.6.2 add-to-back

Without a tail pointer, adding to the back involves traversing the list, stopping at the last node. Then a new node is added:

```
function add-to-back(value)
    {
    if (is-empty())
        add-to-front(value);
    else
        {
        var last = get-last-node();
        last.next = new-node(value,null);
        }
    }
```

If the list is empty, then an *add-to-front* is performed to make sure the head pointer is updated properly. Otherwise, the *get-last-node* function is tasked with the job of walking the list, starting at the head, and returning the last node in the list. As written, *add-to-back* takes $\Theta(n)$ time, due to the cost of the *get-last-node* function.

However, if a tail pointer is used, this operation can be reduced to $\Theta(1)$ time since the last node in the list is cached. When adding to the back, care must be taken to update the tail pointer to the new node.

9.6.3 remove-from-front

Removing from the front updates the head pointer to point to the head's next pointer. In the case of a dummy head node, the dummy's next pointer is updated.

If a tail pointer is used, a check must be made if the list becomes empty. If so, the tail pointer should be set to null.

This operation takes $\Theta(1)$ time.

9.6.4 remove-from-back

If the size of the list is one, then this operation should call remove-from-front to ensure a newly empty list is dealt with properly. Otherwise, one must walk to list to find the next to the last node and set its next pointer to null. If a tail pointer is kept, it is set to the new "last" node.

This operation takes $\Theta(n)$ time, regardless of whether a tail pointer is kept or not.

9.6.5 insert-at-index

Inserting at a given index entails walking the list. With each step, the index is reduced by one. When the index reaches one (using zero-based counting), a new node containing the given value is inserted between the current node, and the node after the current node. Care must be taken so that all pointers are updated correctly. A special case occurs when the given index is zero. If so, *add-to-front* is called instead of performing the walk and subsequent node insertion.

If a dummy head node is used, the walk stops when the index reaches zero. There are no special cases when using a dummy head node.

This operation takes $\Theta(n)$ time.

9.6.6 remove-from-index

Similar to *add-at-index*, but after walking to the proper node, the next node is unlinked from the current node:

```
current.next = current.next.next;
```

Usually, the deleted node or the value of the deleted node is returned after unlinking. Special cases exist if a dummy head node is not used or if a tail pointer is used and the list becomes empty.

This operation takes $\Theta(n)$ time.

9.6.7 is-empty

If the size of the list is cached, then this operation returns whether or not the size is zero. Otherwise, the list is walked and the whether or not the number of steps taken is zero is returned.

9.6.8 size

If the size of the list is cached, then the size must be updated when add or remove operations are performed.

9.6.9 find

Finding a value in the list involves traversing the list, similar to that of finding a value in an array.

9.7 Traversals

Traversing (or walking) a list involves following *next* pointers until a null is found:

```
current = head;
while (current != null)
    {
    current = current.next;
    }
```

If a dummy head node is used, the walk is started using the dummy head node's *next* pointer.

Traversing the list takes $\Theta(n)$ time.

9.8 Concept Inventory

1. This is a question.
   (a) This is one possible answer
   (b) This is another possible answer

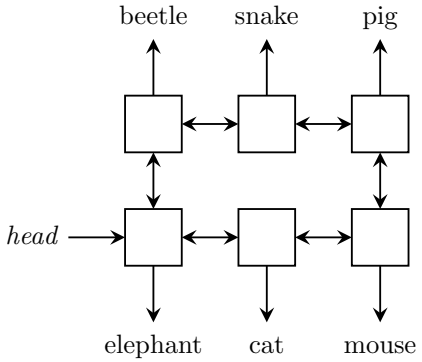
Chapter 10

Doubly Linked-Lists

10.1 Synopsis

A double linked-list is a more versatile version of a singly linked-list. A doubly linked-list node has two node pointers, conventionally named *next* and *prev*. As the names imply, it is possible to move from the head node to the tail (via the *next* pointer) as well as from the tail node to the head (via the *prev* pointer). The major import of this ability is that the tail pointer can be reset in constant time for *both* adding to the back *and* removing from the back.

By making the doubly-linked list circular, the tail pointer can be dispensed with. Here is an illustration of circular doubly-linked list with six nodes:



The double-ended arrows indicate both *next* and *prev* pointers are present. The *next* pointers proceed counter-clockwise in the illustration, while the *prev* pointers proceed clockwise. To find the last node, one follows the *prev* pointers from the head node one step.

A doubly linked-list also benefits from a dummy head node as well as a dummy tail node.

10.2 Structure

The structure is the same as that of singly linked-lists:

```
class doubly-linked-list
```

```
{
var head = null;        //pointer to the first/leftmost node
var tail = null;        //optional: pointer to last/rightmost node
var size = 0;           //optional: holds net number of nodes added
// operations
...
}
```

The difference is in the type of nodes comprising the lists. A doubly linked-list uses nodes with two node pointers.

10.3 Vocabulary

The terms used in describing an doubly linked-list are identical to those of a singly linked-list and a node.

10.4 Supports

Doubly linked-lists are used as a basis for advanced data structures such as binomial and fibonacci heaps.

10.5 Dependencies

Doubly linked-lists are implemented using nodes. with a two node pointers, typically named *next* and *prev*.

10.6 Operations

The operations of a doubly linked-list are very similar to that of singly-linked lists. An additional complication is that the *prev* pointers need to updated consistently.

As stated early, a node previous to the last node can be found in constant time by following the *prev* pointer of the last node. This is useful for resetting the tail pointer when deleting from the back of the list.

10.7 Traversals

A doubly linked-list can easily be traversed from the tail to the head by following *prev* pointers from the tail node.

Traversing the list in this direction takes $\Theta(n)$ time.

10.8 Concept Inventory

1. This is a question.
   (a) This is one possible answer
   (b) This is another possible answer

Chapter 11

Stacks

11.1 Synopsis

A stack is a data structure in which the order items are added to the structure affect the order in which those items are removed. In particular, a stack enforces LIFO ordering, where LIFO stands for *Last-In-First-Out*. For example, suppose the values 5, 13, and 7 were added to a stack in the order given. Then the first item removed from the stack would be 7, the second 13, and the last 5.

Stacks can either be *bounded* or *unbounded*. A bounded stack can only store a fixed number of values while an unbounded stack can store an unlimited number of values, conceptually speaking.

A bounded stack can used a least-recently-pushed replacement strategy when pushing a value onto a stack that is filled to capacity. That is, the least recently pushed value is replaced by the next-to-the-least recently pushed value, and so on.

11.2 Structure

The typical stack has one component, plus operations:

```
class stack
    {
    var store = ?;
    //operations
    ...
    }
```

The *store* can be set to a fillable, circular, or dynamic array or a linked-list. If a fillable or circular array is used, a capacity is typically given as an argument to the constructor; this capacity is used to allocate the array.

11.3 Vocabulary

The terms used in describing a stack are:

push The name of the operation that adds a value to a stack.

pop The name of the operation that removes a value from a stack. Usually, the value removed is returned by the *pop* operation. The *pop* operation returns the most recent value pushed that has not already been popped.

peek Like *pop*, but the value is not removed from the stack. Consecutive peeks, with no intervening pushes or pops, always return the same value. Sometimes, *peek* is named *top*.

top of the stack The most recently pushed item that has not been popped.

bottom of the stack The least recently pushed item that has not been popped.

11.4 Supports

A stack is a top-level data structure that is used in various algorithms.

11.5 Dependencies

Bounded stacks can be implemented with fillable arrays, while unbounded stacks can be implemented with dynamic arrays and singly linked-lists,

A bounded stack with a least-recently-pushed replacement strategy can be efficiently implemented with a circular array.

11.6 Operations

11.6.1 push

This operation adds a given *value* to the top of the stack. If the stack is implemented with a fillable array (dynamic or otherwise), the *push* operation translates to the fillable array's *add-to-back* operation. If the stack is implemented with a linked-list, the push operation translates to the linked-list's *add-to-front* operation.

In the case of a circular array-based bounded stack with a least-recently-pushed replacement strategy, the *push* operation translates to the array's *add-to-back* operation. However, the *add-to-back* operation should not signal an error if the array is full but should instead overwrite the value at the start index as well as incrementing the start index. This can be accomplished with a *add-to-back* operation followed by a *remove-from-front* operation.

Pushing a value onto a stack is expected to take $\Theta(1)$ time.

11.6.2 pop

This operation removes (and returns) the *value* at the top of the stack. If the stack is implemented with a fillable array (dynamic or otherwise), the *pop* operation translates to the array's *remove* operation. If the stack is implemented with a linked-list, the *pop* operation translates to the linked-list's *remove-from-front* operation.

Popping a value from a stack is expected to take $\Theta(1)$ time.

11.6.3 peek

The *peek* operation can be implemented solely using *pop* and *push*:

```
function peek()
    {
    var value;
    value = pop();
```

```
push(value);
return value;
}
```

but this is not considered a good idea since it inflates the number of pushes and pops a stack handles over the course of its lifetime. Typically, for stacks implemented with fillable arrays, the peek operation translates to the array's *get* operation using an index of *size* - 1. For stacks implemented with linked-lists, the peek operation translates to the list's *get* operation using an index of zero.

11.6.4 is-empty

This operation returns true if all pushed values have been popped, false otherwise. This can be done simply by incrementing a counter (initialized to zero) every time a push is performed and decrementing the counter every time a pop is performed. At points in time when the counter is zero, the stack is empty.

It would be an error, of course, if the counter was ever less than zero, indicating there were more pops than pushes.

The *is-empty* operation is expected to take $\Theta(1)$ time.

11.6.5 is-full

This operation returns true if the store is full, but only needs to be implemented for bounded stacks without a replacement policy.

11.7 Concept Inventory

1. These values are pushed onto a stack in the order given: 3, 8, 1. A pop operation would return which value?
   (a) 1
   (b) 3
   (c) 8
   (d) 3 and 1
   (e) 3 and 8
2. Consider a stack based upon a fillable array with pushes onto the back of the array. What is the best one can do in terms of worst case behavior for *push* and *pop*, respectively? You may assume there is sufficient space for each operation.
   (a) constant and constant
   (b) constant and linear
   (c) linear and linear
   (d) linear and constant
3. Consider a stack based upon a circular array with pushes onto the back of the array. What is the best one can do in terms of worst case behavior for *push* and *pop*, respectively? You may assume there is sufficient space for each operation.
   (a) constant and constant

(b) constant and linear

(c) linear and linear

(d) linear and constant

4. Consider a stack based upon a dynamic array with pushes onto the back of the array. What is the best one can do in terms of worst case behavior for *push* and *pop*, respectively? You may assume the array does shrink.

   (a) constant and constant

   (b) constant and linear

   (c) linear and linear

   (d) linear and constant

5. Consider a stack based upon a dynamic circular array with pushes onto the front of the array. What is the best one can do in terms of worst case behavior for *push* and *pop*, respectively? You may assume the array does not shrink.

   (a) constant and constant

   (b) constant and linear

   (c) linear and linear

   (d) linear and constant

6. Consider a stack based upon a singly-linked list without a tail pointer with pushes onto the front of the list. What is the best one can do in terms of worst case behavior for *push* and *pop*, respectively?

   (a) constant and constant

   (b) constant and linear

   (c) linear and linear

   (d) linear and constant

7. Consider a stack based upon a singly-linked list with a tail pointer with pushes onto the back of the list. What is the best one can do in terms of worst case behavior for *push* and *pop*, respectively?

   (a) constant and constant

   (b) constant and linear

   (c) linear and linear

   (d) linear and constant

8. Consider a stack based upon a doubly-linked list with a tail pointer with pushes onto the back of the list. What is the best one can do in terms of worst case behavior for *push* and *pop*, respectively?

   (a) constant and constant

   (b) constant and linear

   (c) linear and linear

   (d) linear and constant

Chapter 12

Queues

12.1 Synopsis

A queue is an *ordered* data structure in that the order of items are added to the structure affect the order in which those items are removed. In particular, a queue enforces FIFO ordering, where FIFO stands for *First-In-First-Out*. For example, suppose the values 5, 13, and 7 were added to a queue in the order given. Then the first item removed from the queue would be 5, the second 13, and the last 7.

Queues can either be *bounded* or *unbounded*. A bounded queue can only store a fixed number of values while an unbounded queue can store an unlimited number of values, subject to memory constraints.

12.2 Structure

A typical queue has one component, plus operations:

```
class queue
    {
    var store = ?;
    //operations
    ...
    }
```

The *store* can be set to a circular array, a dynamic circular array or a linked-list. If a circular array is used as the store, a capacity is typically given as an argument to the constructor; this capacity is used to allocate the array.

12.3 Vocabulary

The terms used in describing a queue are:

enqueue The name of the operation that adds a value to a queue. Analagous to the push operation of stacks.

dequeue The name of the operation that removes a value from a queue. Usually, the value removed is returned by the *dequeue* operation. The *dequeue* operation returns the least recent value enqueued that has not already been dequeued. Analagous to the pop operation of a stack.

peek Like *dequeue*, but the value is not removed from the queue. Consecutive peeks, with no intervening enqueues or dequeues, always return the same value. Sometimes, *peek* is named *front*.

front of the queue The least recently enqueued item that has not been dequeued.

back of the queue The most recently enqueued item that has not been dequeued.

12.4 Supports

A queue is a top-level data structure that is used in various algorithms.

12.5 Dependencies

A bounded queue can be implemented with a circular array, while an unbounded queue can be built from either a dynamic circular array or a singly linked-list that holds a tail pointer.

A queue can also be built using two stacks.

12.6 Operations

12.6.1 enqueue

This operation adds a given *value* to the end of the queue. If the queue is implemented with a circular array (dynamic or otherwise) or a singly linked-list as its store, the *enqueue* operation translates to the store's *add-to-back* operation.

Enqueuing a value is expected to take $\Theta(1)$ time.

12.6.2 dequeue

This operation removes (and returns) the *value* at the beginning of the queue. If the queue's store is a circular array (dynamic or otherwise) or singly-linked list, the *dequeue* operation translates to the store's *remove-from-front* operation.

Removing a value from a queue is expected to take $\Theta(1)$ time.

12.6.3 peek

This operation returns the value that is to be dequeued next, without modifying the state of the queue. Unlike a stack, a *peek* operation cannot be implemented by a *dequeue* followed by an *enqueue*. Thus the implementation of this operator must use an appropriate operation of the store.

12.6.4 is-empty

This operation returns true if all pushed values have been popped, false otherwise. This can be done simply by incrementing a counter (initialized to zero) every time a push is performed and decrementing the counter every time a pop is performed. At points in time when the counter is zero, the queue is empty.

It would be an error, of course, if the counter was ever less than zero, indicating there were more pops than pushes.

The *is-empty* operation is expected to take $\Theta(1)$ time.

12.6.5 is-full

This operation returns true if the store is full, but only needs to be implemented for bounded queues.

12.7 Traversals

Queues, in general, do not support traversals.

12.8 Concept Inventory

1. These values are enqueued onto a queue in the order given: 3, 8, 1. A dequeue operation would return which value?
   (a) 1
   (b) 3
   (c) 8
   (d) 3 and 1
   (e) 3 and 8
2. Consider a queue based upon a regular array with enqueues onto the back of the array. What is the best one can do in terms of worst case behavior for *enqueue* and *dequeue*, respectively?
   (a) constant and constant
   (b) constant and linear
   (c) linear and linear
   (d) linear and constant
3. Consider a queue based upon a circular array with enqueues onto the back of the array. What is the best one can do in terms of worst case behavior for *enqueue* and *dequeue*, respectively?
   (a) constant and constant
   (b) constant and linear
   (c) linear and linear
   (d) linear and constant
4. Consider a queue based upon a singly-linked list without a tail pointer with enqueues onto the front of the list. What is the best one can do in terms of worst case behavior for *enqueue* and *dequeue*, respectively?
   (a) constant and constant
   (b) constant and linear
   (c) linear and linear
   (d) linear and constant
5. Consider a queue based upon a singly-linked list with a tail pointer with enqueues onto the back of the list. What is the best one can do in terms of worst case behavior for *enqueue* and *dequeue*, respectively?
   (a) constant and constant
   (b) constant and linear

(c) linear and linear

(d) linear and constant

6. Consider a queue based upon a doubly-linked list with a tail pointer with enqueues onto the front of the list. What is the best one can do in terms of worst case behavior for *enqueue* and *dequeue*, respectively?

(a) constant and constant

(b) constant and linear

(c) linear and linear

(d) linear and constant

Chapter 13

Priority Queues

13.1 Synopsis

Abstractly, a priority queue is an ordered list of elements. Inserting into a priority queue entails placing the new element in its proper position to maintain the ordering. Thus, the point of the insertion can be anywhere in the list: front, interior, or back. Removing from a priority queue means removing an extreme element, either the 'smallest' or the 'largest'. An auxiliary value, called a *rank*, is used to decide which element is the smallest or largest.

A priority queue is neither a stack nor a queue, but is similar to both. Both a stack and a queue order their elements by the insertion time, whereas a priority queue orders its elements by their ranks. Here is an example:

```
pq = new PriorityQueue();
pq.enqueue("alpha",3);        //element first, rank second
pq.enqueue("beta",2);
pq.enqueue("gamma",4);
pq.enqueue("delta",1);
```

At this point, the ordered list of elements would be:

```
("delta","beta","alpha","gamma")
```

Typically, but not always, the smallest rank is the highest priority, so a removal (dequeuing) would remove the element `"delta"`.

13.2 Structure

Generally, the structure of a priority queue resembles that of a queue or a stack.

13.3 Vocabulary

The terms used in describing a priority queue are:

enqueue The name of the operation that adds an element, with a given rank, to a priority queue.

dequeue The name of the operation that removes the element with the smallest (largest) rank. Usually, the element removed is returned by the *dequeue* operation, with the rank being thrown away.

peek Like *dequeue*, but the element is not removed from the queue. Consecutive peeks, with no intervening enqueues or dequeues, always return the same element.

rank The value used to order the elements in a priority queue. Each element is associated with a *rank*.

priority Another name for *rank*.

first-come-first-served A strategy for dealing with the insertion of an element with the same rank as a previously enqueued, but not yet dequeued, element. Under this strategy, the newly inserted element should be dequeued *after* the previously enqueued element.

13.4 Supports

Priority queues are used in some important graph algorithms as well as in discrete-event simulations.

A priority queue, where the rank of an insertion is forced to be greater than the rank of the previous insertion implements a stack. In such a case, the last element inserted has the highest rank and thus is the next element to come off the priority queue. Conversely, forcing the rank of an insertion to be less than that of the previous insertion implements a queue.

13.5 Dependencies

Priority queues are often built from singly-linked lists or heaps. If the priority queue is based upon a linked-list, a tail pointer is not needed (although if present, it can provide a significant optimization if enqueues generally end up at the back of the list).

13.6 Operations

The operations of a priority queue have the same name as the operations of a queue, namely *enqueue*, *dequeue*, *peek*, and *isEmpty*. Of course, the *enqueue* operation takes a rank as well as the element to be enqueued.

The enqueuing operation can take anywhere from constant to linear time, depending on the underlying data structure used to implement the priority queue. The dequeuing operation can take anywhere from constant to logarithmic time, again depending on the underlying structure. The choice of the underlying structure depends on whether one wishes for faster enqueues (at the cost of slower dequeues) or vice versa.

13.7 Traversals

Priority queues are generally not traversed.

13.8 Concept Inventory

1. This is a question.
   (a) This is one possible answer
   (b) This is another possible answer

Chapter 14

Binary Trees

14.1 Synopsis

A binary tree is a two-dimensional linked-list. Typically, a node in a binary tree has two *next* pointers, named *left* and *right*, and a *value* pointer. Binary tree nodes may have *parent* pointers, as well

14.2 Structure

```
class binaryTreeNdde
    {
    var value;
    var left;
    var right;
    }

class binaryTree
    {
    var root;              //points to the root binary node
    //operations
    ...
    }
```

14.3 Vocabulary

The terms used in describing a binary tree are:

full Each node in the tree has either two children or no children.

perfect A full tree with all the leaves at the same level.

complete A tree that is perfect except for, perhaps, the level furthest from the root. In that furthest level, all the leaves are piled up to the left. A heap is an example of a complete binary tree.

balanced A tree is balanced if the longest path from the root to a leaf is not much longer than the shortest path. In other words, a nearly perfect tree. What constitutes "not much more" and "nearly" depends on the situation.

completely unbalanced A tree is completely unbalanced if it has but one leaf.

root The root node of a binary tree is the only node with no parent. It is considered the "top" of the tree, since trees are commonly drawn upside down, with the branches pointing downwards.

leaf,leaves A node whose left and right pointers are null. In other words, a node with no children.

parent Other than the root, each node n has exactly one other node p pointing to it. The node p is said to be the parent of n.

child,children Other than the leaves, each node p points to one or two other nodes. These nodes are said to be the children of p.

sibling The other child of a parent node (if it exists).

14.4 Supports

Complete binary trees (heaps) are used in heapsorting and other higher-order algorithms.

The most common use of a binary tree is when it is formulated as a binary search tree.

14.5 Dependencies

Binary trees are implemented with nodes.

14.6 Operations

The common operations on a binary tree are *insert*, *find*, and *delete*. The actual implementation of these operations depends on the kind of binary tree, say a binary search tree or a heap.

14.7 Traversals

14.7.1 in-order

In an *in-order* traversal is easily implemented using a recursive process that implements the following steps:

1. traverse the left side
2. process the node
3. traverse the middle side

Note that the node processing step occurs "in the middle". Hence the name, *in-order*. Processing the node is a phrase meant to capture the purpose of the traversal. For printing out the nodes of a tree, processing the node means printing the value of the node. Here is such a traversal:

```
function print-in-order(node)
    {
    if (node.get-left() == null)
        print-in-order(node.get-left());

    println(node.get-value());

    if (node.get-right() == null)
```

```
        print-in-order(node.get-right());
    }
```

This code can be simplified by the strategy of "falling off the tree". One detects a falling off by seeing if the given node has become null. If so, no processing is done. Recall that the null value is used to signify that a node pointer does not point to any node.

```
function print-in-order(node)
    {
    if (node != null)
        {
        print-in-order(node.get-left());
        println(node.get-value());
        print-in-order(node.get-right());
        }
    }
```

14.7.2 pre-order

A *pre-order* traversal is like an in-order traversal, except node processing happens first:

1. process the node
2. traverse the left side
3. traverse the right side

14.7.3 post-order

In a *post-order* traversal, node processing happens last.

14.7.4 level-order

A level order uses a queue as an auxilliary structure. The traversal starts be enqueuing the root node. Next a loop:

- dequeues a node
- processess it
- enqueues the children of the node

The loop continues until the queue is empty.

14.8 Concept Inventory

1. This is a question.
   (a) This is one possible answer
   (b) This is another possible answer

Chapter 15

Binary Search Trees

15.1 Synopsis

A binary search tree is a binary tree where the nodes in the tree are ordered by ther values. Usually nodes on the left side are "less than" than those on the right side of the tree, recursively. When we say a node A is less than a node B, we mean the value stored at node A is less than the value stored at node B. More specifically, the following rules apply at any point in the tree.

- a left-child node is less than or equal to its parent
- a right-child node is equal to or greater than its parent

15.2 Structure

```
class binaryTreeNode
    {
    var value;
    var left;
    var right;
    var parent;            //optional
    }

class binarySearchTree
    {
    var root;               //the top-level binary tree node
    // operations
    ...
    }
```

15.3 Vocabulary

The terms used in describing a binary search tree are the same as those describing a generic binary tree.

15.4 Supports

Binary search trees are considered top-level data structures.

15.5 Dependencies

Binary search trees are based upon tree nodes.

15.6 Operations

The operation descriptions assume unique values in the tree, with values to the left less than those to the right. For trees with duplicate values or a reversed ordering, appropriate modifications to the descriptions should be made.

15.6.1 insert

The typical insertion strategy looks like this:

- create a node n containing the given value to insert
- call a helper function with the *root* and node n

The helper function implements this straegy, with *current* referring to its first argument and n refering to the second:

```
if node n is less than the current node //n must belong on left side of tree
    if the current node has no left child
        set the left child pointer of the current node to n
    else
        recur, passing the left child and n to the helper
 else //n must belong on right side of tree
     if the current node has no right child
        set the left child pointer of the current node to n
     else
        recur, passing the right child and n to the helper
```

A special case exists if the tree is empty, since the root pointer will need to be set. Inserting a value into a binary tree can take linear time, if the tree is pathologically unbalanced.

15.6.2 find

15.6.3 delete

There are a number of deletion strategies. One that preserves the existing balance of the tree to some degree is as follows...

15.7 Traversals

Traversals for binary search trees are the same as those of binary trees.

15.8 Concept Inventory

1. Consider a binary search tree with n nodes. What is the best case time complexity for finding a value at a leaf?

   (a) constant
   (b) log n
   (c) $\sqrt{n}$
   (d) linear
   (e) n log n
   (f) quadratic

2. Consider a binary search tree with n nodes. What is the worst case time complexity for finding a value at a leaf?

   (a) constant
   (b) log n
   (c) $\sqrt{n}$
   (d) linear
   (e) n log n
   (f) quadratic

3. Which ordering of input values builds the most unbalanced BST? Assume values are inserted from left to right.

   (a) 2
   (b) 3
   (c) 1

4. Which ordering of input values builds the most balanced BST? Assume values are inserted from left to right.

   (a) 5
   (b) 7
   (c) 8

5. For all child nodes in a BST, what relationship holds between the value of a child node and the value of its parent?

   (a) greater than or equal to
   (b) less than or equal to
   (c) greater than or equal to, if the child is a right child
   (d) less than or equal to, if the child is a right child
   (e) there is no relationship

6. For all sibling nodes in a BST, what relationship holds between the value of a left child node and the value of its sibling?

   (a) greater than or equal to
   (b) less than or equal to
   (c) equal to
   (d) there is no relationship

7. Do all these deletion strategies for non-leaf nodes reliably preserve BST ordering?

- Swap the values of the node to be deleted and the smallest leaf node with a larger value, then remove the leaf.
- Swap the values of the node to be deleted with its predecessor or successor. If the predecessor or successor is a leaf, remove it. Otherwise, repeat the process.
- If the node to be deleted does not have two children, simply connect the parent's child pointer to the node to the node's child pointer, otherwise, use a correct deletion strategy for nodes with two children.

(a) true

(b) false

Chapter 16

Heaps

16.1 Synopsis

A heap is very much like a binary search tree. It is a binary tree, but the ordering of the values within the tree differs. While a binary search tree has the ordering: *left* < *parent* < *right*, a heap has the ordering: *parent* < *left*,*right* for a *min heap* and *parent* > *left*,*right* for a *max heap*, all assuming no duplicates. Note that, in a heap, the relationship between the values of two siblings is not specified. In practice, a left child may be less than or greater than its sibling, again assuming no duplicates.

When viewed as a tree, a heap is a complete binary tree.

16.2 Structure

```
class heap
    {
    var store;              //store is usually an array
    //operations
    ...
    }
```

16.3 Vocabulary

The terms used in describing a heap are:

min heap With a min heap, the root is always the smallest element in the heap.

min heap With a max heap, the root is always the largest element in the heap.

16.4 Supports

Heaps are used to partially sort elements. This property is exploited by the heapsort algorithm and by many graph search algorithms.

16.5 Dependencies

Heaps are usually based upon arrays. Academically speaking, they can be based upon binary trees but this is rarely done since arrays work so well. If a tree version of a heap is used, an auxiliary data structure must be used to ensure the desired time bound for the *extract-min* operation.

16.6 Operations

The operation descriptions assume a min heap composed of unique values. For max heaps or for heaps with duplicate values, appropriate modifications to the descriptions should be made.

16.6.1 heapify

This operation begins by ensuring the value at the root of the heap is less than its children. If it is not, the root value is swapped with the minimum value, with respect to the children. The process continues by examining the child that received the root's value to ensure it is smaller than either of its children. If it is not, its value is swapped with the minimum value of its children and so on.

The heapify operation takes Θ (log n) time, since the heap is a complete binary tree (and thus is balanced).

16.6.2 build-heap

The build-heap operation turns an unordered complete binary tree in the heap. It begins by running *heapify* on each parent of the leaves. It continues by running heapify on each grandparent of the leaves, and so on. By working from the leaves upward to the root, the smaller values are guaranteed to bubble upwards, with the smallest value ending up at the root.

A naive analysis of the *build-heap* operation yields an 0 (n log n) time bound. Note that this bound is not tight.

16.6.3 get-parent, get-left-child, get-right-child

For heaps based upon arrays, node values are stored in an array and the indices of that array are considered the node pointers. For example, the root node has index 0, the left child of the root has index 1, the right child of the root has index 2. In general, the index of the left child of any node is one more than twice the index of its parent and the index of the right child of any node is one more than the index of the left child.

To compute the parent index of a node n, simply subtract one from n's index and then divide the result by two (integer division).

The *get-* operations can obviously be performed in constant time.

16.6.4 extract-min

To extract the minimum value of a heap (which is found at the root, one first saves a pointer to the root value. Next, the rightmost leaf in the lowest layer is pruned from the heap and its value is used to replace the root's value. Finally, the heapify operation is performed on the root to ensure heap ordering is preserved.

This is why an array is so convenient for storing values in a heap. The index of the rightmost leaf in the lowest layer is the last index of the array. If there are s elements in the heap, the last index is $s - 1$.

Pruning the rightmost leaf in the lowest layer devolves to reducing the perceived size of the array. Note that the array does not actually have to be reduced in size.

Since pruning takes constant time, the entire *extract-min* operation takes Θ (log n) time since the operation is dominated by *heapify*.

16.6.5 bubble-up

The *bubble-up* operation, given an index, moves the value at that index upwards towards the root, if needed. It swaps the value with the parent if the value is less than the parent's, and then repeats the process for the parent. The *bubble-up* operation is used if an insertion operation is provided. Note that this operation ensures only a single path in the heap adheres to heap ordering.

16.6.6 insert

To insert a value into a heap of size s, one bases one places the new value into the heap at index s. The size is then incremented to reflect the additional element. The *bubble-up* operation is called on the new value to ensure it rises to the proper level in the heap.

Insertion requires the heap be based upon a fillable array (for a bounded heap) or a dynamic array (for an unbounded heap).

16.6.7 peek

The *peek* operation returns the value at the root of the heap.

16.6.8 is-empty

This operation returns true if there are no elements in the heap and false otherwise.

16.7 Traversals

Heaps are generally not traversed.

16.8 Concept Inventory

1. This is a question.
   (a) This is one possible answer
   (b) This is another possible answer

Chapter 17

Hash Tables

17.1 Synopsis

17.2 Structure

17.3 Vocabulary

17.4 Supports

17.5 Dependencies

17.6 Operations

17.7 Traversals

17.8 Concept Inventory

1. Consider chaining as a collision strategy. What is the best possible worst case behaviors for insertion and finding, respectively?

   (a) constant, linear
   (b) constant, constant
   (c) constant, linear
   (d) linear, linear
   (e) linear, quadratic
   (f) quadratic, linear

2. Consider open addressing as a collision strategy. What is the best possible worst case behaviors for insertion and finding, respectively?

   (a) linear, linear
   (b) constant, linear
   (c) linear, constant

(d) constant, constant

(e) linear, quadratic

(f) quadratic, linear

3. Consider rehasing with a perfect hash as a collision strategy. possible worst case behaviors for insertion and finding, respectively?

(a) linear, constant

(b) constant, linear

(c) constant, constant

(d) linear, linear

(e) linear, quadratic

(f) quadratic, linear