Basic Data Structures for an Advanced Audience

Patrick Lin

Very Incomplete Draft Version 2020-10-11

Abstract

I starting writing these notes for my fiancée when I was giving her an improvised course on Data Structures. The intended audience here has somewhat non-standard background for people learning this material: a number of advanced mathematics courses, courses in Automata Theory and Computational Complexity, but comparatively little exposure to Algorithms and Data Structures.

Because of the heavy math background that included some advanced theoretical computer science, I felt comfortable with using more advanced mathematics and abstraction than one might find in a standard undergraduate course on Data Structures; on the other hand there are also some exercises which are purely exercises in implementational details. As such, **these notes would require heavy revision in order to be used for more general audiences**.

I am heavily indebted to Pat Morin's *Open Data Structures* and Jeff Erickson's *Algorithms*, in particular the parts of the "Director's Cut" of the latter that deal with randomization. Anyone familiar with these two sources will undoubtedly recognize their influences.

I apologize for any strange notation, butchered analyses, bad analogies, misleading and/or factually incorrect statements, lack of references, etc. Due to the specific nature of these notes it can be hard to find time/reason to do large rewrites.

Contents

1 Models of Computation for Data Structures

Here are some common models of computation for working with data structures. I intend for these notes to focus on the first model only. Maybe in another set of notes we will talk about the rest.

1. The word RAM model.

   In this model, we will assume that we have random access memory consisting of "cells", where each "cell" contains a w-bit "word", where we assume that $w = \Omega(\log n)$ where n is the number of elements in a given data structure. This is a fairly mild assumption since $\Theta(\log n)$ is roughly the size of a pointer used to access information. Intuitively, we can think of this as modeling running time on an infinite family of machines where for each input size n, we can pick a machine whose word size w is large enough to support the input size. In reality, since we do not have access to such an infinite family, we deal with inputs that are too large to fit into physical RAM using other models.

   In this model we assume that any of the following standard operations on words take $O(1)$ time:

   - arithmetic (+,-,*,/)
   - comparison (>,<,=)
   - bitwise boolean[1] (&,|,!,^)

   We assume that memory I/O takes constant time as well. As such, time and space analysis is done in terms of the number of such operations; operations done on larger numbers are measured in terms of number of word operations needed to simulate them.

   This is an abstraction of operations that occur at the hardware level where we can show that these operations take amortized "$O(1)$" time. We might give some justification for this when talking about cache and the External Memory model, though realistically we will probably defer that discussion for another time.

2. The bit-level RAM model.

   In this model, we will once again assume random access memory, but we are breaking apart the words and working directly with bits.

   In this model, bitwise arithmetic, comparison, and boolean operations take $O(1)$ time here; we measure complexity is done in terms of the bits themselves.

[1] An aside about bitwise boolean operators and the land of complexity dragons:

Most algorithms we will talk about outside of this unit are designed with the word RAM model in mind, minus the bitwise boolean operations. In particular, I will call this the "number API" model: given any two numbers, we can do arithmetic and comparison in constant time. That is to say, if the algorithm only uses this API and not the underlying implementation details, we might as well not concern ourselves with the implementation details (i.e., bit representations). In particular, many of the algorithms we will look at would work just fine on a magical computer that can do computations on the reals.

In practice, algorithms that perform well in theory on a real number API work well in practice on IEEE floating point. We will see one notable exception later on: the Ford-Fulkerson algorithm behaves very differently on integer inputs than on real inputs. This discrepancy suggests that the algorithm might actually perform quite poorly when given floating point input!

One should take care, of course, to not be too cavalier about operations here–there is a reason why the number API only has standard arithmetic and comparisons as allowed operations. Schönhage 1979 showed that if you assume real numbers and then also introduce taking floors in $O(1)$ time (i.e., integer division), then you can solve the PSPACE-complete problem QBF in polynomial time; Hartmanis and Simon 1974 achieved the same result by instead allowing bitwise boolean operators.

On the flip side, Canny 1998 showed that the existential theory of the reals ($\exists\mathbb{R}$) lies in PSPACE. The existential theory of the reals can be thought of as follows. Recall that NP is the class of all problems of the form "$\exists x_1, \dots, x_n \in \{0,1\}, f(x)$", so we can call NP $\exists(\mathbb{Z}/2\mathbb{Z})$. $\exists\mathbb{R}$ is naturally the class of problems of the form "$\exists x_1, \dots, x_n \in \mathbb{R}, f(x)$".

This model is particularly useful when talking about data from a finite set. Examples include situations where we are only ever dealing with integers smaller than $2^w - 1$, or geometric data whose coordinates are real-valued but we only ever care about their relative positions.

3. The cell probe model.

   This model was introduced by Andy Yao, but was used with great success by Mihai Patrascu when he effectively revolutionized the field of data structure lower bounds. In this model, all operations are free except memory accesses (the so-called cell probes). The motivation for using this model is its inherent simplicity: it is significantly easier to prove lower bounds on the number of memory accesses than on the total number of operations.

4. External memory models.

   When accessing data on your computer, not all of it comes at the same cost. In particular, some data exists in your CPU cache and is blazingly fast to obtain; other data exists in RAM and is slightly slower. More data probably exists in slower storage like an SSD (or HDD); even more egregiously your data might be in another castle entirely. A lot of the classical work done here was motivated by efficiently accessing databases, where the data was almost definitely in another castle.

   External memory models are meant to break open the black box on the $O(1)$ memory access, and work out the right parameters to get reasonable access times in spite of the vast differences in access speeds at each layer. Mind you, even these models are abstracted away from the hardware–they still assume that each layer of hardware can return an arbitrary "block" of size B in $O(B)$ time. There are a few variations. For example, the cache-aware model knows what the values of B are for the different levels; the cache-oblivious model abstracts it away.

5. etc etc

   There are many more than touched upon here: the nuances of unrealistic word RAM models mentioned above, pointer machines, parallel machine models, quantum models, and undoubtedly others that I am not equipped to talk about in detail. Even writing some of the details above required some research!

2 APIs (ADTs) vs Data Structures

Before we dive into data structures themselves we should take care to distinguish them from the APIs (application programming interface) they implement. Some people prefer to refer to more specifically refer to the APIs implemented by data structures as ADTs (abstract data type). An ADT is simply a specification for what operations are required to be supported by a data structure implementing it, with no guarantees on performance (i.e., running time of operations/space usage of the data structure). Many algorithms will specify a desired ADT to use; the running time of the algorithm will then depend on the specific implementation used.

On the other hand, a data structure is free to implement operations not specified in the specification of an ADT that it is an implementation of. In fact it is common for a single data structure to implement multiple ADTs simultaneously.

Notation: given a(n instance of a) data structure D, and an operation $\mathsf{op}(x, y, z)$ that it implements, we will use the (object-oriented) notation $D.\mathsf{op}(x, y, z)$ to mean excecuting the operation $\mathsf{op}(x, y, z)$ on D.

You might already familiar with some of the following basic ADTs that are commonly used:[2]

- Queues:

  This data type is specified with some kind of order in which to retrieve them.

  Supported operations:

  - `insert`(x): inserts x
  - `delete`(): removes the next element to be retrieved and returns it

  FIFO queues are usually called Queues, LIFO (FILO?) queues are usually called Stacks. Other places might call these operations enqueue/dequeue (for queues) or push/pop (for stacks) for obvious reasons. A Priority Queue returns the smallest (or largest) element when `delete`() is called, for this reason other places might call the operation `remove_min`() (or `remove_max`()).

- Deque:

  This data type generalizes queues and stacks (deque is short for double-ended queue). Any data structure that implements a deque trivially implements the Stack and Queue data types by losing two of the four operations.

  Supported operations:

  - `insert_first`(x): inserts x at the first position
  - `insert_last`(x): inserts x at the last position
  - `delete_first`(): removes the first element and returns its value
  - `delete_last`(): removes the last element and returns its value

- List: This data type represents a linear sequence of data.

  Supported operations:

  - `size`(): returns the number of elements n in the list
  - `get`(i): returns the i-th element in the sequence

[2]Some of the names of operations I have chosen here are not necessarily standard. Additionally, it is common to find ADTs with the same names that vary slightly in specification. For example, some Queue specifications do not return anything via `delete`(); instead the task of obtaining the next element to be retrieved is given into a separate `peek`() operation.

  - `set`(i, x): sets the value of the i-th element to x and returns the original element at i
  - `insert`(i, x): inserts the value x at position i and increments the position of each element originally at position between i and the end, inclusive.
  - `delete`(i): deletes the i-th element in the sequence and decrements the position of each element originally at position between $i + 1$ and the end, inclusive.

  Note that while Deques easily implement Stacks and Queues, Deques are in turn easily implemented by Lists.

- Unordered Set:

  A bag of data, no duplicates allowed.

  Supported operations:

  - `size`$()$: returns the number of elements n in the bag
  - `insert`(x): inserts x into the bag; optionally returns true if the bag did not already contain x and false otherwise
  - `remove`(x): removes x from the bag; optionally returns true if the bag actually contained x and false otherwise
  - `find`(x): returns true if the bag contains x

- Ordered Set: Same as before except data must be sortable. Specifying an Ordered Set requires a comparison operator. The find operation now returns the largest element y such that $y \leq x$ (or some reasonable alternative) or NULL if no such y exists.

- Dictionary:

TODO Write

- Graph:

TODO Write

Some of these ADTs have many popular implementations that we will cover in these notes. For example, we will describe at least no fewer than four different implementations of the List ADT (some in the exercises), at least some of which are actually used in practice.

3 Array-Based Data Structures

Arrays are a basic underlying structure used to implement many data types. We will punt the details of how an array is implemented because the actual details belong to a lower level of the computing stack than this course is meant for. For the purposes of abstraction you can and should think of an array as a continuous block of memory that has been allocated for use, and therefore reading and writing to arbitrary cells in that block take $O(1)$ time.

The Array API provides the following methods:

- `set`(i, x): sets the i-th cell to x
- `get`(i): returns the value of the i-th cell
- `length`$()$: returns the number of cells

Notation: As is standard in many programming languages, for an Array A, we will write A.`get`(i) as $A[i]$, and A.`set`(i, x) as $A[i] \leftarrow x$. We will access a block of cells as follows: $A[i, \ldots, j]$ is the block of cells starting at $A[i]$ and ending at $A[j]$.

Convention: We will follow the convention that Arrays are 1-indexed, i.e., its cells are indexed by $A[1]$ through $A[n]$, as opposed to the 0-indexing convention in many popular languages such as C++ or Python.

One thing that you probably haven't considered yet is the running time of creating an array. We will assume that there is an operation `new array`(n) that allocates a block of n cells, and that this operation takes $O(n)$ time; this is a fairly reasonable assumption for most implementations of arrays.

3.1 ArrayLists and Amortized Analysis

You probably already noticed that Arrays can be used to implement `size`$()$, `get`(i), and `set`(i, x) operations of the List ADT in $O(1)$ time each. But `insert`(i, x) and `delete`(i) are slow: These operations take $O(n - i)$ time to move all the $n - i$ elements that need to get shifted. There is also the following consideration: the underlying Array has static size! What are we supposed to do when we run out of room?

Let us rewind a bit. We will define a data structure we call an ArrayList. Let n be the number of elements in the ArrayList. At all times the ArrayList consists of an underlying Array A whose true size is at least n; otherwise we cannot store all of the necessary information. We can keep n as a separate counter and return that when `size`$()$ is called:

```
size():
    return $n$
```

The implementations of `get`(i) and `set`(i, x) are obvious, but let us write them out for clarity's sake:

```
get($i$):
    return $A[i]$

set($i, x$):
    $y \leftarrow A[i]$
    $A[i] \leftarrow x$
    return $y$
```

So now what exactly are we to do about `insert`? The answer is rather interesting. Let's actually ignore `delete`(i) for now and assume an insert-only list.

If $n = A.$`length`$()$, we will allocate a new array of length $2n$. But remember, this takes $O(n)$ time, so the worst-case performance of `insert`(i, x) is always $O(n)$, which is way worse than $O(n-i)$ if i is large. What on earth are we thinking?

```
insert($i, x$):
    if ($i > n + 1$)
        throw error
    if ($n = A$.length())
        resize()
    $A[i+1, \ldots, n+1] \leftarrow A[i, \ldots, n]$
    $A[i] \leftarrow x$
    $n \leftarrow n + 1$

resize():
    $B \leftarrow$ new array(max(1, $2n$))
    $B[1, \ldots, n] \leftarrow A[1, \ldots, n]$
    $A \leftarrow B$
```

Well here's the trick: we will now perform what's called Amortized Analysis and show that under this method of performance analysis, `insert`(i, x) takes $O(n-i)$ amortized time.

The idea is as follows: Suppose a data structure supports k operations, with respective amortized running times $t_1, \ldots, t_k$. Then if, say, over the lifetime of the data structure, the i-th operation is executed n_i times, then the total running time of said sequence of events is $O(\sum_i n_i t_i)$.[3]

Tarjan in 1985 described the following method for Amortized Analysis as the "physicist's method"; subsequent texts call it the "potential method".

The true cost of the t-th operation will be denoted by $cost(t)$. Costs will be computed as if each of the "base" operations (arithmetic, comparison, cell read/write) takes 1 step instead of $O(1)$ time: dropping

[3]An aside on amortization and analysis of algorithms:

If you have an algorithm that uses some kind of dynamically sized Array (like C++'s `std::vector` or Python's `lists`) you can usually, for the purposes of time analysis, assume that operations like `append`(x) take constant time in the worst-case. This is perfectly safe if, over the life the algorithm, you will actually perform a sequence of `appends`/`inserts`/`deletes` that can absorb the costs of the `resize`() operations into their amortized costs. For the purposes of saving a constant or two, if you know that you will have a lot of inserts/queries you can always pre-allocate an array of the largest length needed in advance, and this cost will then get overrun later by the costs of the inserts and deletes.

In practice there are situations where amortized costs are insufficient, and predictability is more important than efficiency. For example, in real-time systems, individual operations cannot miss their deadlines, even if the entire pipeline finishes ahead of schedule. In interactive settings, users will notice if suddenly one of the operations suddenly takes significantly longer than average; having every operation take two seconds is preferable to, say, having 99% of operations take one second and then 1% of operations taking one minute. In these settings one can try to de-amortize these data structures.

One example of a de-amortizing technique introduced by Overmars in 1983 called *lazy rebuilding*, where expensive operations are split up and explicitly "charged" to the operations whose contributions to the potential end up "paying" for the expensive operation in the amortized analysis. For example, consider the insert-only ArrayList setting. Suppose we could "reserve" a block of memory for free (or in $O(1)$ time) and then allocate it later. Instead of just one Array A, we keep a "shadow copy" B whose reserved length is twice the length of A. Every time we insert into A, we also insert into B, allocate another cell of memory for B, and copy any previous entry in A that has not been replicated in B. Thus when A is full, B explicitly contains a full copy of A and has the correct amount of allocated empty space, and we can just replace A with B and start a new shadow copy. Thus by increasing the cost of every `insert`(i, x) by some constant factor, we have made it so that `resize`() takes $O(1)$ time. Allowing for deletes is a much more complicated picture and requires more complicated techniques.

the constants greatly simplifies the analysis, and doing so does not matter since we can multiply by the most expensive one to upper bound. We will define a non-negative function Φ, called the "potential energy" or more commonly just the "potential", and define the amortized cost of the t-th operation

$$acost(t) := cost(t) + \Phi(t) - \Phi(t-1).$$

We will assume without loss of generality that $\Phi(0) = 0$. The idea is as follows: we want the data structure to be in a state of minimum energy (some ideal state), and so we will penalize operations that increase the potential energy of the system, and discount operations that decrease the potential energy. Picking the right potential function is something of an art; there are researchers whose entire careers have been devoted to finding potential functions that better capture the behavior of a data structure.

There is a question of why it's okay to be doing this at all. The reason is that by definition,

$$\sum_t acost(t) \geq \sum_t cost(t),$$

and so our analysis will never underestimate the total time cost over the lifetime of the data structure. In essence, we are doing a more careful analysis where we move certain costs around to different operations.

In our case, we will set our "ideal state" to be when $2n = A.\texttt{length}()$, since this is the state we are in after resize() is called. We are in the most unideal state when $n = A.\texttt{length}()$, since after that point we will reach a breaking point and force the data structure back into the ideal state. So one possible potential function would $\Phi = 2n - A.\texttt{length}()$. For reasons that will become clear later, it will prove convenient to use the equivalent potential $\Phi = 3(2n - A.\texttt{length}())$.

`get`(i) and `set`(i, x) do not change the potential, so their amortized costs are the same as their costs. Now suppose the t-th operation is an `insert`(i, x) operation. Either $A.\texttt{length}()/2 \leq n < A.\texttt{length}()$, or $n = A.\texttt{length}()$. In the first case, the potential increases by 6, so

$$\begin{aligned} acost(t) &= cost(t) + 6 \\ &= (n-i) + O(1) + 6 \\ &= O(n-i). \end{aligned}$$

In the second case, the potential decreases from $3n$ to 0, so if we assume `new array`$(2n)$ takes $2n$ steps, and copying the contents of A into the new array takes n steps, then

$$\begin{aligned} acost(t) &= cost(t) - 3n \\ &= 3n + (n-i) + O(1) - 3n \\ &= O(n-i). \end{aligned}$$

In both cases, we find that the amortized running time of `insert`(i, x) is $O(n-i)$.

What about `delete`(i)? Suppose we implement `delete`(i) in the obvious way. Then the invariant $\Phi \geq 0$ would fail. To maintain this invariant, we could shrink the array whenever it was less than half full. But notice that if we did that, then starting with an array that is half full, over a sequence of only deletions, we would have to resize the array at every call! This is not good. Instead, we will do the following:

```
delete($i$):
    $y \leftarrow A[i]$
    $A[i,\ldots,n-1] \leftarrow A[i+1,\ldots,n]$
    $n \leftarrow n-1$
    if ($4n \le A$.length())
        resize()
    return $y$
```

Of course, the potential function we were using before will not work either; one will need to use a modified version to show that `delete`(i) will have amortized $O(n-i)$ running time.

Homework Problem 1.

> Show that for this implementation of `delete`(i), the amortized running time of `insert`(i, x) and `delete`(i) are each $O(n-i)$. *Hint: Your potential function should still be in the "ideal state" when* $2n = A$.`length`().

Food for thought (aka optional homework): Does the choice $n \le A$.`length`()/4 in `delete`(i) matter? Consider both the question of asymptotics and optimizing hidden constants.

3.2 Array{Stack,Queue,Deque}s

We can implement Stacks using Arrays by simply using the ArrayList implementation directly. We will call this implementation an ArrayStack. We will implement `insert`(x) by calling the ArrayList's `insert`(n, x) and `delete`() by calling ArrayList's `delete`(n). This is actually a really good implementation, because using the analysis in the previous subsection, we know that `insert`(x) and `delete`() take $O(1)$ amortized time each.

But Queues are a different story: if we set `insert`(x) as `insert`(n, x) and `delete`() as `delete`(1), then `insert`(x) takes $O(1)$ amortized time but `delete`() takes $O(n)$ amortized time; implementing `insert`(x) as `insert`($1, x$) and `delete`() as `delete`(n) just switches the runtimes. What to do about this?

As a thought experiment, imagine we had an Array of length ∞. Then we can maintain a pointer p pointing at the current "start" of the Array, and have the following implementations: set `insert`(x) as `insert`($p+n-1, x$), and `delete`() would simply be $p \leftarrow p+1$. Of course, we can't have an Array of length ∞, so we will need to simulate it somehow.

One possibility is to use an idea called a CircularArrayQueue: Instead of treating the underlying Array A as a block of memory indexed 1 through A.`length`(), we consider the indices as equivalence classes modulo A.`length`(). The pointer p can be incremented continuously, cycling through the Array. Instead of inserting at position n, we insert at position $((p+n) \bmod A.\texttt{length}()) + 1$. Here the $+1$ is because we are working with 1-indexed arrays. If we used 0-indexed arrays we would have -1's floating around in other places.

```
insert($x$):
    if ($n = A$.length())
        resize()
    $A[((p+n) \bmod A.\texttt{length}()) + 1] \leftarrow x$
    $n \leftarrow n+1$
```

```
delete():
    if ($n = 0$):
        return NULL
    $y \leftarrow A[p]$
    $p \leftarrow (p \bmod A.\texttt{length}()) + 1$
    $n \leftarrow n - 1$
    if ($n \leq A.\texttt{length}()/4$)
        resize()
    return $y$

resize():
    $B \leftarrow \texttt{new array}(\max(1, 2n))$
    $B[1, \ldots, n] \leftarrow A[p, \ldots, ((p + n) \bmod A.\texttt{length}()) - 1]$
    $A \leftarrow B$
    $p \leftarrow 1$
```

Note that while there is no theoretical reason to actually move the Array elements to the beginning when resizing the array, but there is no asymptotic cost to doing so (since the call to `new array` already takes $O(2n)$ time) and there are practical constant factor reasons in the land of there be dragons why it is preferable to be access data towards the beginning of your Array than towards the end.

Homework Problem 2.

Adapt the idea of CircularArrayQueue to give an implementation of CircularArrayList that supports:

- `get`(i) and `set`(i, x) in $O(1)$ time, and
- `insert`(i, x) and `remove`(i) in $O(\min(i, n - i))$ amortized time.

Conclude that this also results in an implementation of CircuarArrayDeque that takes $O(1)$ amortized time for each of the insert/delete operations.
Hint: It might be useful to first show that in the CircularArrayQueue implementation, `insert`(x) *and* `delete()` *each take amortized $O(1)$ time.*

3.3 Graphs and AdjacencyMatrices

TODO Write

4 Pointer-Based Data Structures

The pointer-machine model of data structures is a collection of base objects called *Nodes*, each of which contains a fixed number of member variables called *fields*. These fields are either data, or pointers to other Nodes. Creating a new Node, assigning data to a field, and reading field data are all constant time, as is accessing a Node pointed to by a pointer.

We will use `new node()` to create a new Node.

A pointer-based data structure is commonly passed around as a pointer to a "root" Node, or a small collection of pointers to "source" Nodes.

4.1 Linked{List,Queue,Stack,Deque}s

The most basic example of a pointer-based data structure is an implementation of the List ADT called a SinglyLinkedList.

A SinglyLinkedList consists of a sequence of SinglyLinkedListNodes (for brevity, we will refer to them as just Nodes in this section). Each Node object contains two member variables:

- *value*: the value at that Node
- *next*: a pointer to the next Node in the sequence.

Let's start by looking at how one might implement a SinglyLinkedQueue. We will keep three pieces of data:

- *head*: a pointer to the first Node in the sequence
- *tail*: a pointer to the last Node in the sequence
- *n*: the number of elements in the List

To implement `insert`(x) and `delete`(), we can do the following:

```
insert($x$):
    $u \leftarrow$ new node()
    $u.value \leftarrow x$
    if ($head = \text{NULL}$)
        $head \leftarrow u$
    else
        $tail.next \leftarrow u$
    $tail \leftarrow u$
    $n \leftarrow n + 1$

delete():
    if ($head = \text{NULL}$)
        throw error
    $y \leftarrow head.value$
    $head \leftarrow head.next$
    $n \leftarrow n - 1$
    if ($head = \text{NULL}$)
        $tail \leftarrow \text{NULL}$
    return $y$
```

Clearly both of these operations take worst-case $O(1)$ time.

Easy exercise.

Explain how to implement `insert`(x) and `delete`$()$ for a SinglyLinkedStack in $O(1)$ time.

But if we wanted a SinglyLinkedList, we're in trouble: consider trying to implement `get`(i). With no built-in random access, the following is the best we can hope for with the current specification:

```
get(i):
    $j \leftarrow 1$
    $v \leftarrow head$
    while ($j < i$)
        $v \leftarrow v.next$
    return $v.value$
```

Wow, this is $O(i)$, which is terrible compared to the $O(1)$ afforded to us by ArrayLists. Surely we can do better!

Optional exercise.

Explain how to implement the other SinglyLinkedList operations.

One small improvement we can make is to move from SinglyLinkedList to DoublyLinkedList: now in addition to a pointer *next* pointing to the next Node in the sequence, each Node also has a pointer *prev* pointing to the previous Node. To make it easier to deal with edge cases, we will replace the *head* and *tail* pointers with a *dummy* Node whose *next* pointer points to the first Node in the sequence and whose *prev* pointer points to the last Node. Initially both of these pointers will point to *dummy* itself.

We will make use of the following helper function that returns the i-th Node.

```
get_node(i):
    $v \leftarrow dummy$
    if ($i < n/2$)
        $j \leftarrow 0$
        while ($j < i$)
            $v \leftarrow v.next$
    else
        $j \leftarrow n + 1$
        while ($j > i$)
            $v \leftarrow v.prev$
    return $v$
```

Then `get`(i), `set`(i, x), and `insert`(i, x), and `delete`(i) are easily implemented using `get_node`(i).

```
get(i):
    return get_node($i$).$value$
```

```
set(i, x):
    $v \leftarrow$ get_node($i$)
    $y \leftarrow v.value$
    $v.value \leftarrow x$
    return $y$
```

```
insert($i, x$):
    $u \leftarrow$ new node()
    $u.value \leftarrow x$
    $v \leftarrow$ get_node($i$)
    $u.prev \leftarrow v.prev$
    $u.next \leftarrow v$
    $v.prev \leftarrow u$
    $u.prev.next \leftarrow u$
    $n \leftarrow n + 1$

delete($i$):
    $v \leftarrow$ get_node($i$)
    $v.prev.next \leftarrow v.next$
    $v.next.prev \leftarrow v.prev$
    $n \leftarrow n - 1$
```

Each of these operations has its run time dominated by `get_node`(i), which takes $O(\min(i, n-i))$ time. Ouch.

Easy exercise.

Explain how to implement a DoublyLinkedDeque where all operations take $O(1)$ time.

This begs the question of why anyone would use a LinkedList over an ArrayList when `get`(i) and `set`(i, x) are so much more expensive and ArrayList achieves the same running time anyway for `insert`(i, x) and `remove`(i), at least in an amortized sense.

LinkedLists are most useful when random access is not used (e.g., in SinglyLinkedQueue, SinglyLinked-Stack, and DoublyLinkedDeque, operations take *worst-case* constant time as opposed to *amortized* constant time for their Array-based counterparts), or when there is an external way of directly accessing the Nodes, e.g., some other part of the program keeps pointers to the individual Nodes. If v is a pointer to a Node, then `set`(v, x), `insert_before`(v, x), `insert_after`(v, x), and `delete`(v) can all be implemented in $O(1)$ time.

Another motivation for using LinkedLists is in a situation when the programming paradigm inherently makes working with the concept of Arrays difficult; see the next Homework Problem.

Homework Problem 3: A tangent about purely functional data structures.

This unit implicitly works in the imperative (and more precisely, object-oriented) paradigm of programming. You are undoubtedly familiar with the imperative paradigm of writing code: it is the paradigm in which most C-family code is written. In the imperative paradigm one has a program (or object) state that is modified by various operations. Naturally, this can make reasoning about the correctness of the program quite difficult, because many things can change the program state in different ways.

The functional paradigm prefers, as its name suggests, a functional approach: instead of an object having member operations that change its internal state, we have functions that take an object (and any pertinent inputs) and outputs a new object with the relevant modifications. As its name might suggest, the MapReduce framework for distributed computing operates in the functional paradigm.

Data structures in a functional setting are fully persistent: we have permanent access to all versions of the data structure throughout its history, and given any previous version, we are free to branch off from that point and create a new “timeline”. Even in non-functional settings, pointer machines provide concise and efficient ways of implementing persistent versions of various data structures.

Let us say that a FunctionalNode is a pair $(value, next) \in \text{Word} \times \text{FunctionalNode}$ that never changes. For a FunctionalNode *head*, we will use $value(head)$ and $next(head)$ to obtain the relevant entries in the pair. Since objects never change, objects are always passed by reference. Thus the function `insert_first` : FunctionalNode → FunctionalNode defined by

$$\texttt{insert\_first}(head, x) := (x, head)$$

takes $O(1)$ time to construct and return a reference to the new FunctionalNode. A FunctionalLinkedList can just be given by a FunctionalNode *head*; the function `get` : FunctionalNode × Int → Word that returns i-th entry in the List can be implemented as

$$\begin{aligned}\texttt{get}(head, i) := &\ \text{throw error} && \text{if } i < 1\\ &\mid value(head) && \text{if } i = 1\\ &\mid \texttt{get}(next(head), i-1) && \text{otherwise}\end{aligned}$$

However, the operations $\texttt{set}(head, i, x)$, $\texttt{insert}(head, i, x)$ and $\texttt{delete}(head, i)$ all take $\Omega(i)$ time: although the $(i+1)$-th through n-th FunctionalNodes can remain unmodified, the first i entries must be copied into new FunctionalNodes because we cannot use the old references for those! That being said, the fact that the $(i+1)$-th through n-th FunctionaNodes can remain unchanged serves as the basis for a standard technique in making pointer-based data structures persistent in general called path-copying: only copy what needs to be copied and leave the rest be. There exist different functional implementations of Lists that do not suffer from this time overhead, but their implementation details are very complicated. In many functional languages, such as Standard ML, Haskell, and in fact the very first functional language, LISP, Lists behave internally like FunctionalLinkedLists, so pushing to and popping off the beginning are cheap but accessing the end is expensive (for example, in Haskell `last` A takes $\Theta(n)$ time to return the last value stored in the List A).

The standard functional implementation of a Queue keeps two Lists called *front* and *back*, where *front* contains the front part of the Queue in forward order, and *back* contains the back part in backward order. New values are pushed to the beginning of *back* and old elements are popped off the beginning of front. When *front* is empty, the reversal of *back* is installed as the new *front*, and *back* is replaced by an empty List. The following invariant is maintained: if the *front* List is empty, then the *back* List must be too.

We can implement FunctionalLinkedQueue using two FunctionalLinkedLists. Concretely, a FunctionalLinkedQueue is an ordered pair of FunctionalNodes $(front, back)$. We will have two operations

- `insert` : Queue × Word → Queue, taking a Queue and “appending” the value to the “end”, and
- `delete` : Queue → Queue × Word, removing the “first” element of the Queue and returning it.

These two operations are implemented as follows:

$$\texttt{insert}((\mathit{front},\mathit{back}),x) := \begin{cases}(\texttt{reverse}((x,\mathit{back})),\textsc{Null}) & \text{if } \mathit{front}=\textsc{Null}\\ \mid (\mathit{front},(x,\mathit{back})) & \text{otherwise}\end{cases}$$

$$\texttt{delete}((\mathit{front},\mathit{back})) := \begin{cases}\text{throw error} & \text{if } \mathit{front}=\textsc{Null}\\ \mid ((\texttt{reverse}(\mathit{back}),\textsc{Null}),\mathit{value}(\mathit{front}) & \text{if } \mathit{next}(\mathit{front})=\textsc{Null}\\ \mid ((\mathit{next}(\mathit{front}),\mathit{back}),\mathit{value}(\mathit{front})) & \text{otherwise}\end{cases}$$

The function reverse : FunctionalNode → FunctionalNode returns the reversal of the input List, and is implemented as

$$\texttt{reverse}(\mathit{head}) := \texttt{rev}(\mathit{head},\textsc{Null})$$

where

$$\texttt{rev}(\mathit{head},r) := \begin{cases} r & \text{if } \mathit{head}=\textsc{Null}\\ \mid \texttt{rev}(\mathit{next}(\mathit{head}),(\mathit{value}(\mathit{head}),r)) & \text{otherwise}\end{cases}$$

It is not hard to show that *front* $=$ NULL only if *back* $=$ NULL, and that $\texttt{insert}(Q,x)$ and $\texttt{delete}(Q)$ each take $O(1)$ amortized time, via the potential function $\Phi = \texttt{size}(\mathit{back})$, where size : FunctionalNode → Int can be implemented via

$$\texttt{size}(\mathit{head}) := \begin{cases} 0 & \text{if } \mathit{head}=\textsc{Null}\\ \mid 1+\texttt{size}(\mathit{next}(\mathit{head})) & \text{otherwise}\end{cases}$$

Hoogerwood in 1992 gave the following functional implementation of a Deque as an easy extension of the standard functional implementation of a Queue. *front* and *back* are now treated symmetrically: both must be non-empty if the Deque contains more than one element, and when one List becomes empty, the other is split in half (with one of the halves being reversed).

Explain how to implement Hoogerwood's FunctionalDeque using FunctionalLinkedLists and prove that each operation takes $O(1)$ amortized time.

Homework Problem 4.

Recall that we observed the following runtimes:

	CircularArrayList	DoublyLinkedList
size()	$O(1)$	$O(1)$
get(i)	$O(1)$	$O(\min(i,n-i))$
set(i,x)	$O(1)$	$O(\min(i,n-i))$
insert(i,x)	$O(\min(i,n-i))^\dagger$	$O(\min(i,n-i))$
delete(i)	$O(\min(i,n-i))^\dagger$	$O(\min(i,n-i))$

where † denotes that the running time is amortized.

In this exercise we aim to interpolate between the two choices.

In a BlockLinkedList of block size b, we maintain a DoublyLinkedList where in each Node, instead of keeping a value, we keep an Array B (called the *block* at the Node) of length $b+1$ that never grows and never shrinks. We will require the following invariant: each block, except for the last block, contains at least $b-1$ and at most $b+1$ elements in it. This means that a BlockLinkedList that contains n elements uses $O(n/b)$ space.

Note that if we care about the amount of "wasted space", i.e., space used by the data structure that isn't being used for storing the data itself, we are doing pretty well: consider that CircularArrayList, in the worst case, half the Array is empty, wasting n space, and since DoublyLinkedLists use two pointers per Node, plus the *dummy* Node, it wastes $2n+2$ space. On the flip side, in the worst case, the last block is empty and $2n/b$ pointers are used, for a total of $b+2n/b+2$ wasted space, so for example when $b=\Theta(\sqrt{n})$, we have achieved significant savings.

Implement BlockLinkedList so that for block size b, `get`(i) and `set`(i,x) take $O(\min(i,n-i)/b)$ time, and `insert`(i,x) and `delete`(i) take $O(b+\min(i,n-i)/b)$ amortized time.

Thus when $b=2$, we more or less recover a DoublyLinkedList; when $b>n$, we recover an ArrayList.

4.2 AdjacencyLists

TODO Write

4.3 Binary Search Trees

TODO Formatting, Editing

The next most basic example of a pointer-based data structure is that of a Rooted Tree. This data structure does not inherently implement any of the ADTs we talked about in Section 2, but rather serves as the underlying structure for implementations of various ADTs that we will talk about later. Because we will not talk about Unrooted Trees in this unit, we will henceforth drop the Rooted modifier from the term.

A TreeNode consists of a (optional) data value, and a List of pointers to TreeNodes called the children. Note that while this immediately implies that the list of children is ordered, the ordering need not be meaningful. Some applications (such as the Search Trees seen in the next section) will impose an ordering on the children list; for others the ordering will be arbitrary.

A Tree consists of a distinguished TreeNode called the root, and other TreeNodes. We can define the structure recursively: a single Node u is a tree whose root is u; given a collection of disjoint Trees T_1 to T_k with respective roots r_1 to r_k, a new Tree T can be created by setting a new TreeNode r as the root of T and setting r's list of children to be r_1 through r_k.

We will now establish a laundry list of terminology.

Given a TreeNode u, u is the parent of its child TreeNodes. If, starting from a TreeNode u, one can reach a TreeNode v by following zero or more child pointers, then u is an ancestor of v, and v is a descendant of u. A proper ancestor/descendant is an ancestor/descendant that is not itself. A TreeNode with no children is called a leaf. A TreeNode with children is called internal. Given a TreeNode u, the subtree rooted at u consists of u and all of its descendants (i.e., the recursive substructure whose root is u).

By definition, for every TreeNode u in a Tree, there is a unique path of pointers from the root to u. The level of a u in the Tree is the length of this path. The height of the Tree is maximum level of any Node, or, equivalently, if T consists of root r whose children are the roots of subtrees T_1 through T_k, height(T) = 1 + max{height(T_i) | 1 <= i <= k}.

A m-ary Tree is one where each TreeNode is allowed to have at most m children. A full m-ary Tree is one where each TreeNode has either 0 or m children. A SinglyLinkedList can be thought of as a 1-ary Tree. A full m-ary Tree is perfect if all its leaves are at the same level.

Homework Problem 5: Prove the following: a. A full m-ary Tree with i internal Nodes has mi + 1 Nodes. b. A m-ary Tree of height h contains at most m^{h+1} - 1 Nodes. (Hint: use the recursive definitions of Tree and height) c. For fixed height h, the number of Nodes in a m-ary Tree is maximized when the tree is perfect.

BinarySearchTree is a Tree that implements OrderedSet. As its name suggests, it is a 2-ary Tree. Traditionally, the children of a TreeNode of a BinarySearchTree are called left and right. As usual, we keep a TreeNode pointer root and a counter n, and have

```
size():
    return n
```

We maintain the following invariant, called the binary search tree property: for any TreeNode u, all values stored in the subtree rooted at u.left are smaller than u.value, and all values stored in the subtree rooted at u.right are larger than u.value. This invariant allows us to implement find(x) as follows:

```
find(x):
    curr <- root
    prev <- null
    while (curr $\ne$ null):
        if (x = curr.value):
            return true
        else if (x < curr.value):
            curr <- curr.left
        else if (x > curr.value):
            prev <- curr
            curr <- curr.right
    if (prev = null):
        return null
    return prev.value
```

Keeping track of prev allows us to correctly return the largest y such that y <= x (or null if no such element exists).

We need to maintain the invariant when inserting and removing.

For insert(x), we will first try to find a TreeNode containing x by following left or right pointers just as in find(x). If we find it then there is nothing to do; otherwise, consider the last TreeNode traversed; call this TreeNode u. Since we did not find x, we must have tried to follow u.left or u.right and found null there. But we can then insert x into that position with no problem. This is summarized in the code below:

```
insert(x):
    curr <- root
    prev <- null
    while (curr $\ne$ null):
        prev <- curr
        if (x = curr.value):
            return false
        else if (x < curr.value):
            curr <- curr.left
        else if (x > curr.value):
            curr <- curr.right

    u <- new_node()
    u.value <- x
    if (prev = null):
        root <- u
    else if (x < prev.value):
        prev.left <- u
    else
        prev.right <- u
    n <- n + 1
    return true
```

Removal is a bit more tricky. Let u be the TreeNode containing x. If u is a leaf, then we can just remove it. If u only has one child, then we can have u's parent adopt it. The tricky part is when u has two children. But in this case, the largest value stored in the subtree rooted at u.left is, by the binary search tree property, smaller than all values stored in the subtree rooted at u.right, and thus we can replace u with the corresponding TreeNode.

```
    adopt(parent,child):
        grandchild <- u.left if (u.left $\ne$ null) else u.right
        if (child = root):
            root <- grandchild
        else:
            if (parent.left = child):
                parent.left <- grandchild
            else
                parent.right <- grandchild

    remove(x):
        curr <- root
        prev <- null
        while (curr $\ne$ null):
            prev <- curr
            if (x = curr.value):
                return false
            else if (x < curr.value):
                curr <- curr.left
            else if (x > curr.value):
                curr <- curr.right

        if (curr = null):
            return false

        if (curr.left = null or curr.right = null):
            adopt(prev,curr)
        else:
            left <- curr.left
            leftprev <- curr
            while (left.right $\ne$ null):
                leftprev <- left
                left <- left.right
            curr.value <- left.value
            adopt(leftprev,left)
        n <- n - 1
        return true
```

In each of find(x), insert(x), and remove(x), the bulk of the running time is spent finding a path from the root to either the TreeNode containing x or to a leaf TreeNode. Thus the running time of each of these operations is O(h), where h is the height of the BinarySearchTree.

Homework Problem 6: Design a BinarySearchTree that, using O(h) extra overhead on insert(x) and remove(x), maintains for each TreeNode its level and the height and size information of the subtree rooted at said TreeNode.

Without knowing anything else about the Tree, all we know is that $h <= n$, so the running time of each of these operations is O(n). But we know that if the Tree were full and perfect, then n = 2^{h+1} - 1, in which case $h = O(\log n)$. So our life would be greatly improved if there were a way of making sure that the BinarySearchTree stayed as full and perfect as possible.

There is a family of refinements of BinarySearchTree collectively referred to as BalancedBinarySearchTrees. The common theme between all of them is that $h = O(\log n)$.

We will briefly summarize the simplest of these to implement, the AVL tree, named after its inventors Adelson-Velskii and Landis. AVL trees maintain the height-balanced property: for each TreeNode u, the height of the subtree rooted at u.left and the height of the subtree rooted at u.right differ by at most one.

AVL trees work as follows. When inserting or removing, it is possible to cause one of a TreeNode's subtrees to have heights that differ by two. Let u be a TreeNode where this occurs, such that both of u's subtrees still satisfy the height-balanced property. Without loss of generality, suppose that the height of the subtree rooted at u.left is h and the height of the subtree rooted at u.right is $h + 2$. Set v to be u.right. There are two possibilities. If the subtree rooted at v.right has height $h + 1$ and the subtree rooted at v.left has height h, then the following function returns the root of a Tree that has height $h + 1$:

```
    rotateLeft(u):
        v <- u.right
        u.right <-v.left
        v.left <- u
        return v
```

If instead it is the subtree rooted at v.left that has height $h + 1$, we would do the following:

```
    doubleRotateLeft(u):
        u.right <- rotateRight(v)
        rotateLeft(u)

    where rotateRight(u) is the natural analog of rotateLeft(u).
```

Homework Problem 7: a. Show that for every insert/remove, only O(h) calls to rotateLeft(u) or rotateRight(u) need to be made to rebalance the tree, and thus insert(x) and remove(x) each take O(h) time. You may assume that you know the heights of all subtrees before and after each rotation. b. Prove that the AVL property implies that h = O(log n). Hint: First show that there exists a constant c such that the number of leaves is $\Omega(c^h)$. Then use the fact that n is at least the number of leaves.

Another data structure in the BalancedBinarySearchTree family is the Red-Black Tree. Red-Black Trees are not as balanced as AVL trees, so find(x) is faster for AVL trees. On the other hand, Red-Black trees have a more complicated rotation rule that results in only needing to do O(1) rotations per insert(x)/remove(x) instead of O(h), so the insert(x)/remove(x) are faster for Red-Black trees. In general Red-Black trees are preferred since we expect to be inserting/removing fairly often.

In general, if we were making significantly more calls to find(x) than insert(x)/remove(x), AVL trees are still not the best choice, and in fact, BalancedBinarySearchTrees need not be preferred! One can imagine a sequence of find(x) operations where a tree with height $\Omega(n)$ has better total search time than a BalancedBinarySearchTree. Consider the case where there are no insertions/removals, and the frequency of find(x) queries is known in advance. Then one can compute the optimal BinarySearchTree for the frequencies via dynamic programming.

Homework Problem 8: OptimalBinarySearchTrees Suppose you are given a sorted array of keys A[1,...,n] and an array of corresponding access frequencies f[1,...,n], where we assume that find(A[i]) is called exactly f[i] times. Describe an efficient algorithm that finds the BinarySearchTree that minimizes the total search time over all queries.

In the case where the access frequencies are not known in advance and insertions/removals are possible, one option is to use what is called a Splay Tree, which is also not a BalancedBinarySearchTree. Every time find(x) or insert(x) is called, a series of rotations and "splays" are performed to move the TreeNode containing x to the root. In essence, the most recently accessed elements are very close to the root, and so we expect frequently queried values to have very short query times. The details are beyond the scope of this unit, but remarkably, find(x), insert(x), and remove(x) each have O(log n) amortized running time.

Splay trees are conjectured to be dynamically optimal, i.e., the total search time over all queries is within a constant factor of the total search time of an OptimalBinarySearchTree computed from the access frequencies if they had been known in advance. Tango trees, proposed by Demaine, Harmon, Iacono and Pătrașcu in 2004, have been proven to be within O(loglogn) of dynamic optimality.

BinarySearchTrees can also implement UnorderedSet over any sortable data by a small modification to the find(x) operation. Implementing UnorderedSet for unsortable data will be deferred to the unit on Randomization, where we will spend a decent amount of time on hashing.

Homework Problem 9: Describe an efficient algorithm that, given an array A with n numbers and an integer k, decides whether A has at most k distinct numbers. One can do this by first sorting A in O(nlogn) time, but the goal is an algorithm that is faster when $k \ll n$.

4.4 Heaps

TODO Formatting, Editing

Heap are a class of (Rooted) Tree-based data structures that satisfy the (min-)heap property: for each TreeNode u, the value (or key) at u is smaller than the values at all of u's children. Naturally, maintaining this invariant allows any Heap to implement the Priority Queue data structure.

4.4.1 DaryHeaps

The most basic Heap is the BinaryHeap, which is built on top of a binary Tree structure. More generally, we have DaryHeaps which are built on top of a d-ary Tree structure. In addition to the heap property, DaryHeaps also maintain the shape property: the DaryHeap is a complete d-ary Tree, that is, every level of the tree except the bottom level has the maximum number of possible TreeNodes; furthermore in the bottom level all of the TreeNodes are as far to the left as possible.

The shape property gives two advantages.

To explain the first advantage, let us first suppose that we have a way to manage where the next TreeNode should be added to maintain the shape property. Then insert(x) can be implemented as follows: first we add a TreeNode containing x where it should be, then, if x is smaller than the parent TreeNode's value, we swap the two values. We continue doing this until x is no longer smaller than its parent's value. This swapping takes O(h) time, and the shape property ensures that h = O(log n)!

The second advantage is that it allows us to use what is called Eytzinger's method simulate the DaryTree structure using an Array in such a way that adding x in the right place is trivial. The result is an ImplicitDaryHeap.

Suppose we gave each TreeNode in a Tree an index as follows: The root is given index 1, and then for each level, we look at the TreeNodes from left to right, giving each subsequent TreeNode the next consecutive index value. Given a Tree, one might implement this as follows:

```
breadth-first-ordering(root):
    index <- 1
    q <- new_queue()
    q.insert(root)
    while (not q.empty()):
        node <- q.delete()
        node.index <- index
        index = index + 1
        for (child in node.children):
            q.insert(child)
```

For the remainder of this section we will focus on ImplicitBinaryHeap. For a complete binary Tree, notice that the following property holds: for a TreeNode u, u.left.index = 2u.index, u.right.index = 2u.index + 1, and u.parent.index = floor(u.index / 2).

Thus the shape property allows us to implicitly maintain a BinaryHeap using an Array instead of a pointer machine. The upshot of this is that if we use an underlying Array A and maintain a counter n for the number of values in the BinaryHeap, then insert(x) first consists of setting A[n+1] to x and then doing the parent-swapping can be done using index-manipulation. As before, when A gets too large (or too small), we will resize.

```
insert(x):
    if (A.length = n):
        resize()
    n <- n + 1
```

```
        A[n+1] <- x
        bubble_up(n)

    bubble_up(i):
        p = floor(i/2)
        while (i > 0 and A[i] < A[p]):
            A[i],A[p] <- A[p],A[i]
            i <- p
            p = floor(i/2)
```

To implement delete(), the value we want is at the root of the Tree. But how do we maintain the heap property and the shape property? One way to maintain the shape property is to swap the values of A[1] and A[n] and then decrement n. But this violates the heap property because A[n] might be bigger than A[2] and A[3]. But this is rather simple to fix: we can swap the value originally at A[n] with the smaller of A[2] and A[3], and then repeat. Whereas with insert we had to "bubble up" a small value, here we want to "trickle down" a large one.

```
    delete():
        y <- A[1]
        A[1] <- A[n]
        n <- n - 1
        trickle_down(1)
        if (4n <= A.length()):
            resize()
        return y

    trickle_down(i):
        while (i >= 0):
            j <- -1
            l <- 2 * i
            r <- 2 * i + 1
            if ( r < n and A[r] < A[i]):
                if (A[l] < A[r]):
                    j <- l
                else:
                    j <- r
            else if (l < n and A[l] < A[i]):
                j <- l
            if (j >= 0):
                A[j],A[i] <- A[i],A[j]
            i <- j
```

So, combined with the amortization argument from Section 3, insert(x) and delete() each take O(h) = O(log n) amortized time. As we will see later on, trickle_down(i) is the true workhorse of this data structure.

Homework Problem 10: The shape property allows us to use Eytzinger's method to implicitly represent the underlying binary Tree using an Array and find the next position (for insert(x)) and last position (for delete()) in the Tree for free. However, this means that the O(h) running time must be amortized, due to needing to resize the Array. Explain how to keep track of these positions using O(h) overhead per insert(x)/delete() operation in an explicit BinaryTree structure.

Given an Array A containing n elements in arbitrary order, one can "heapify" it in-place (i.e., using constant auxiliary storage) using the following method developed by Floyd. Let us look at this from the viewpoint of a binary Tree that satisfies the shape property but not the heap property. Suppose we have a TreeNode u for which the subtrees rooted at u's children satisfy the heap property. Then we can fix the Tree rooted at u by calling trickle_down on u! One can do this recursively, or specify the following explicit evaluation order, which is a valid reverse topological sort:

```
heapify():
    for (i from n down to 1):
        trickle_down(i)
```

One can make a small optimization by observing that the subtrees rooted at the leaves trivially satisfy the heap property and so we can start from a smaller index than n (namely, n/2).

heapify and trickle_down give us the ability to sort an array in-place via the following procedure:

```
heap_sort():
    heapify()
    for (n from A.length() down to 1):
        swap A[1] <-> A[n]
        trickle_down(1)
```

Because we discussed BinaryHeap as fulfilling the min-heap property, this results in A being in sorted in reverse order. If the BinaryHeap instead had been implemented as fulfilling the max-heap property, then A would be sorted in forward order.

Naively, the running time of heapify is O(nh) = O(nlogn), but if amortized analysis has taught you anything, it is that we can be more careful with our analysis.

Homework Problem 11: Fix k, and for a binary Tree satisfying the shape property, derive an upper bound on the number of TreeNodes for which the subtree rooted at that TreeNode has height k. Use this to show that the running time of heapify() is actually O(n).

4.4.2 More Advanced Heaps, Briefly

Although we did not explicitly give these operations in the PriorityQueue ADT, many applications that use PriorityQueues also desire at least one of two additional operations.

The first operation is decrease_key(x,k). In this version of a PriorityQueue, the elements are not values x, but rather key-value pairs (k,x), where the PriorityQueue returns the value with the smallest key instead of the smallest value. Naturally, decrease_key(x,k) changes the key attached to value x to k.

A note about implementating decrease_key: Most Heap implementations are backed by some kind of Tree, whether it be pointer-based or implicitly represented by an Array. Because the contents of Heap are a bit of a mess, we assume some auxiliary way that allows us to obtain the TreeNode (or index) containing x in, say, O(1) time. For example, we might ask insert(x) to return a pointer to the TreeNode containing x.

The other operation is meld(P,Q): given two PriorityQueues P and Q, combine them into a single PriorityQueue.

Homework Problem 12: Describe implementations for decrease_key and meld in (Implicit)BinaryHeap.

Challenge: Come up with a Heap whose (amortized) performance is better than that of a (Implicit)BinaryHeap.

There are many applications that desire cheap decrease_key and/or meld operations. For example, Dijkstra's algorithm calls decrease_key many times for each time it calls delete(). The fact that BinaryHeap takes O(logn) time to perform insertion also leaves a lot to be desired.

There are many theoretical Heaps that have equivalent performance: Brodal Heaps (Brodal) and Strict Fibonacci Heaps (Brodal, Lagogiannis, and Tarjan) each support insert in O(1) time, delete in O(log n) time, decrease_key in O(1) time, and meld in O(1) time. However, both types of Heaps are rather complicated and, in practice, run very slowly.

Pairing Heaps (Fredman et al) are extremely easy to describe, have extremely good experimental performance and have been proven by Iacono to support insert and meld in O(1) time, delete in O(log n) amortized time, and decrease_key in o(log n) amortized time. The exact (amortized) complexity of the decrease_key operation is an open problem. Fredman showed a lower bound of $\Omega(\log\log n)$, and a disputed result of Pettie states that insert, meld, and decrease_key can be shown to run in O(4ˆ{sqrt(loglogn)}) amortized time. However, experiments indicate that in practice, Pairing Heaps are consistently quite efficient in many settings, though on small inputs ImplicitDaryHeaps tend to perform better.

In summary, for theoretical purposes, when analyzing the running time of an algorithm using a PriorityQueue, assume a (Strict) Fibonacci Heap; in practice prefer a Pairing Heap.

5 Randomization

5.1 Random Search Trees and Treaps

TODO Write

5.2 Hashing

TODO Write

Appendix: Tail Inequalities

TODO Write