CS 240: Data Structures and Data Management

Chris Thomson

Winter 2013, University of Waterloo

Notes written from Alejandro López-Ortiz's lectures.

Contents

1 Introduction & Code Optimization

← January 8, 2013

1.1 Course Structure

The grading scheme is 50% final, 25% midterm, and 25% assignments. There are five assignments, usually due on Wednesday mornings at 9:30 am. There are several textbooks for the course, all of which are optional but recommended. The textbooks cover roughly 80% of the course content. There are also some course notes published online from previous terms, however the lectures will not necessarily follow those notes strictly.

See the course syllabus for more information.

1.2 CS as the Science of Information

So far in our undergraduate careers, computer science has meant programming. However, programming is only a subset of computer science. Computer science is the **science of information**.

What do we want to do with information? We want to:

- **Process it**. Programs ≡ algorithms.
- **Store it**. We want to encode it. This also leads to information theory. Storing information involves data structures, databases, (file) systems, etc., all of which are searchable in some way.
- **Transmit it**. We want to transmit information over networks. This process involves coding theory.
- **Search it**. First, we have to structure it with data structures and/or SQL databases. Information retrieval is the process of searching for textual information instead of numerical information.
- **Mine it**. This involves artificial intelligence and machine learning.
- **Display it**. Information needs to be displayed using computer graphics (CG) and user interfaces (UI, partially related to psychology).
- **Secure it**. Encryption and cryptography are important. Information should also be stored redundantly to prevent harm from catastrophic events.

1.3 Objectives of the Course

- **Study efficient methods of storing, accessing, and performing operations on large collections of data.**

  "Large" is subjective – your data needs to be large enough to justify the additional mental complexity.

  Typical operations:

  – Insert new item.

  - "Deleting" data (flagging data as "deleted", not actually deleting the data).
  - Searching for data.
  - Sorting the data.

  Examples of "large data":

  - The web.
  - Facebook data.
  - DNA data.
  - LHC (Large Hadron Collider) measurements (terabytes per day).
  - All the music from iTunes, for finding duplicate songs, etc.

- **There is a strong emphasis on mathematical analysis.**

  The performance of algorithms will be analyzed using order notation.

- **The course will involve abstract data types (objects) and data structures.**

  We will see examples that have to do with:

  - Databases.
  - Geographic databases.
  - Text searching.
  - Text compression.
  - Web searching.

1.4 Code Optimization & Measuring Performance

Richard Feynman was in charge of the computer group on the Manhattan Project. He made his code run 10x faster.

The initial method to measure performance was to use a **wall clock**. Initial studies looked like this:

Data size	A	B
3	1	3
5	2	9
10	4	10
20	16	11

However, computers were getting better, so we couldn't use wall clocks anymore. Results were not robust enough due to the quick progression of performance increases in terms of computing power. Moreover, because of the differences in architecture, the same program may have a different execution time on two different computer models.

Idea: rather than comparing algorithms using seconds, we should compare algorithms using the number of operations required for the algorithm to run.

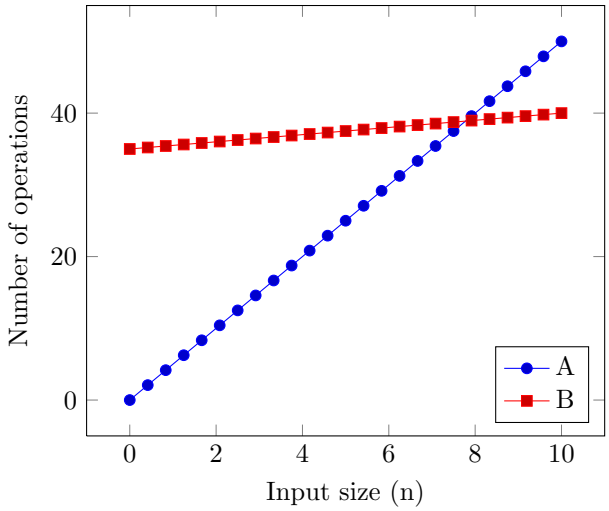


Figure 1: A comparison of two algorithms A and B.

- Express algorithm using pseudocode.
- Count the number of primitive operations.
- Plot the number of operations vs. the input size.

Note that A and B are plotted as continuous functions, however they are not actually continuous – we join all of the points for readability purposes only.

In the long run, you may want to use algorithm B even if algorithm A is better in some cases, because the benefits of A will be short lived as n grows.

Hence, comparing the programs has been transformed into comparing functions. We use order notation to measure the long-term growth of functions, allowing us to choose the smaller function, which corresponds to the faster algorithm.

2 Order Notation

← January 10, 2013

The time for algorithm A on input of size n is:

$$\underbrace{2n \log n}_{\text{sort}} + \underbrace{1.5n^3}_{\text{mult}} + \underbrace{22n^2 - 3n}_{\text{additions}} + \underbrace{7}_{\text{setup}}$$

On a different machine, the same algorithm A may be $2n \log n + 9n^3 + 10n^2 - 3n + 7$. These give a **false sense of precision**. These specifics can also be quite costly to determine. In the 1960s, Don Knuth proposed **order notation** as a way to analyze the general quality of algorithms in an accurate, cost- and time-efficient way by ignoring precise runtimes.

2.1 Formal Definitions

Order notation represents algorithms as functions. We say a function $f(n)$ relates to a certain order $g(n)$ using different notation depending on which relation we're interested in.

Relation	Functions
$3 \leq 7$	$f(n) = O(g(n))$
$8 \geq 7$	$f(n) = \Omega(g(n))$
$7 = 7$	$f(n) = \Theta(g(n))$
$3 < 7$	$f(n) = o(g(n))$
$7 > 3$	$f(n) = \omega(g(n))$

Here's an easy way to remember the correspondence between a relation and its order notation symbol: the greedy operators, $\leq$ and $\geq$, are uppercase letters O and Ω. The less greedy operators, $<$ and $>$, are the same letters but in lowercase (o and ω).

Order notation only cares about the long run. An algorithm can violate the relation early – we're interested in its asymptotic behavior.

Definition ($\leq$)**.** $f(n) = O(g(n))$ if there exist constants $c > 0, n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Definition ($\geq$)**.** $f(n) = \Omega(g(n))$ if there exist constants $c > 0, n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Definition ($=$)**.** $f(n) = \Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

The equality relation is effectively sandwiching the algorithm $f(n)$ between two other functions (the squeeze theorem). If it's possible to sandwich the algorithm between two multiples of the same $g(n)$, then $f(x)$ is said to be equal to the order $g(n)$.

Definition ($<$)**.** $f(n) = o(g(n))$ if for all $c > 0$ there exists constant $n_0 > 0$ such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

Definition ($>$)**.** $f(n) = \omega(g(n))$ if for all $c > 0$ there exists constant $n_0 > 0$ such that $f(n) > c \cdot g(n)$ for all $n \geq n_0$.

2.2 Order Notation Examples

Example 2.1. $2n^2 + 3n + 11 = O(n^2)$

We need to show that there exists constants $c > 0$ and $n_0 > 0$ such that $2n^2 + 3n + 11 \leq cn^2$ for all $n \geq n_0$.

Let $c = 3$. Simplifying gets us $3n + 11 \leq n^2$. This holds for $n_0 = 1000$, for instance.

Example 2.2. $2n^2 + 3n + 111 = \Theta(n^2)$

In order to show equality (Θ), we need to show both the $\leq$ and $\geq$ cases. In the previous example, the $\leq$ case was shown. For the $\geq$ case, we need to show $n^2 \leq c \cdot (2n^2 + 3n + 11)$.

Let $c = 1$. Simplifying gets us $n^2 \leq 2n^2 + 3n + 11$, which gives us $0 \leq n^2 + 3n + 11$. This holds for $n_0 = 0$.

Example 2.3. $2010n^2 + 1388 = o(n^3)$

We must show that for all $c > 0$, there exists $n_0 > 0$ such that $2010n^2 + 1388 \leq c \cdot n^3$ for all $n \geq n_0$.

$$\frac{2010n^2 + 1388}{n^3} \overset{?}{\leq} \frac{c \cdot n^3}{n^3} \implies \frac{2010}{n} + \frac{1388}{n^3} \overset{?}{\leq} c$$

There is a **trick to prove f(n) = o(g(n))**: show that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

2.3 A note about the use of =

$$\underbrace{2n^2}_{\text{specific function}} \quad \in \quad \underbrace{O(n^2))}_{\text{set of functions}}$$

The use of the equality operator (=) in order notation is not the same as in most other areas of mathematics, since a single element of a set cannot strictly be equal to the set itself (even $a \neq \{a\}$). The = in order notation is not a true equality – it's just notation. The $\in$ operator is semantically more correct.

2.4 Performance of Algorithms

We have made some progression in how we determine the performance of algorithms.

- Comparing single runs.
- Comparing measured curves (multiple runs).
- Produce analytical expressions for the curves.
- Simplify using order notation.

Let's say we have the problem of integrating a mathematical function. Our program numerically integrates the function, and we're trying to analyze the algorithm behind that process. The timing would vary depending on the mathematical function and preciseness required.

If we conduct multiple runs of our algorithm, we'll get different times. Plotting all of the results (time vs. input size) will result in data that is not a function, because the data fails the vertical line test, since for a given input size we have multiple times.

We need to decide on a convention for these cases where we have multiple time values. In this course, we'll usually look at the worst case, however in some situations examining the best or average cases could be useful.

Finding the average case is hard. To produce an appropriate average, we need an input distribution.

3 Formalism

← January 15, 2013

Definition. A **problem** is a collection of questions and (correct) answer pairs.

Example 3.1. Informally, the problem is "multiplying two numbers." More formally, the problem could be represented as: (2 x 3, 6), (3 x 4, 12), (1 x 5, 5), etc.

Definition. An **instance** of a problem is a specific question and answer pair, (Q, A).

Definition. The **input** of a problem is the question. Example: 3 x 4.

Definition. The **output** is the only correct answer. Note: there is only one correct answer under this definition. "Multiple" correct answers can always be reduced to some canonical form that represents the one correct answer.

Definition. An **algorithm** is a mechanical method to produce the answer to a given question of a problem.

Definition. The **size of the input** is the number of bits, characters, or elements in the question. This definition will vary depending on what's most appropriate for the given question.

Long division in elementary school was the first time a problem's complexity was directly related to the input size. That's not always the case, however.

Definition. We say an algorithm **solves** a problem if for every question Q it produces the correct answer, A.

Definition. A **program** is an implementation of an algorithm using a specified computer language.

Example 3.2. We want to sort n numbers. One instance of this problem Q is:

$$(\underbrace{(5,1,3,2,4)}_{\text{Q}}, \underbrace{(1,2,3,4,5)}_{\text{A}})$$

For a problem in this form, we'll say the size of Q is $|I| = 5$. Why? Counting the number of elements is the most logical definition in this case.

This course emphasizes algorithms rather than programs. We're computer scientists, so we care about the algorithm and the speed/efficiency of that algorithm.

A problem Π can have several correct algorithms that solve it. Our goal is to find efficient solutions to Π. How do we do that?

1. **Algorithm design**. Find a solution(s) to Π.
2. **Algorithm analysis**. Assess the correctness and efficiency of the algorithm.

In this course, we're mostly interested in algorithm analysis. Algorithm design will be covered in more detail in CS 341.

3.1 Timing Functions

A timing function is a function $T_{\mathcal{A}}$ such that:

$$T_{\mathcal{A}} : \{Q\} \to \mathbb{R}^+$$

where $\mathbb{R}^+$ is the set of positive real numbers.

$T_{\mathcal{A}}(Q)$ is the time taken by algorithm $\mathcal{A}$ to compute the answer to Q.

We also want to define a general timing function for a particular problem, regardless of algorithm:

$$\begin{aligned} T(n) &= max\{T_{\mathcal{A}}(Q)\} \\ T_{avg}(n) &= avg\{T_{\mathcal{A}}(Q)\} \\ T_{min}(n) &= min\{T_{\mathcal{A}}(Q)\} \\ &\vdots \end{aligned}$$

If we had two solutions (algorithms) $T_A(n)$ and $T_B(n)$, we could use order notation to compare the quality of A and B.

Note that some problem instances are easier than others to solve, even with the same input size. Most people find it easier to multiply 0×7 than 8×7, for instance, despite those two problems having the same input size.

4 Analysis of Algorithms

In order to analyze an algorithm, we count the number of basic operations that the algorithm performs.

Example 4.1. Let's say our algorithm is to compute $x^2 + 4x$ and assign it to a variable called `result`. That involves seven basic operations:

1. Read x from memory.
2. Compute $x \cdot x$.
3. Assign the result of $x \cdot x$ to a variable, `pr1`.
4. Compute $4 \cdot x$.
5. Assign the result of $4 \cdot x$ to a variable, `pr2`.
6. Compute `pr1` + `pr2`.
7. Assign the result of `pr1` + `pr2` to a variable, `result`.

The number of operations remains the same across machines, but the actual running time of an algorithm will differ from machine to machine (which one of the reasons why we use order notation).

We only count the number of basic operations. There are various definitions of what a "basic" operation actually is (especially with regards to variable assignments), but these are some common operations that are considered to be basic:

- Add numbers of reasonable size (numbers that can be added primitively).
- Multiply numbers of reasonable size (numbers that can multiplied primitively).
- Access an index in an array.
- Store a value in memory (variable assignments).
- Read a value from memory.

Be careful. Some languages disguise complexity well. Just because the syntax is simple doesn't necessarily mean it's a basic operation, and doesn't guarantee that the operation runs in constant time.

4.1 Techniques for Algorithm Analysis

- Straight line programs (no loops). Simply tally up the basic operation count.
- Loops (`for`/`while`). You need to add the number of operations for each pass on the body of the loop.

Example 4.2. Analyze the following algorithm.

for $i = a$ *to* b **do**
| $<$ straight line program $> T_L$
end

This program will run with $\sum_{i=a}^{b} T_L(i)$ operations.

Whenever you have a loop in your algorithm, you should expect to get a summation in your timing function.

Example 4.3. Analyze the following algorithm.

for $x = 0$ *to* 10 **do**
| pr1 = x * x;
| pr2 = 4 * x;
| result = pr1 + pr2;
| print result;
end

This program will run with $\sum_{i=0}^{10} 4 = 44$ operations (depending on your definition of "basic" operations).

Operations like addition and multiplication are primitive for numbers of a reasonable size. Once numbers get large enough to exceed certain limits, we have to resort to some trickery to perform those operations, which adds additional complexity, making them no longer "basic" operations.

We can be a bit sloppy and use the worst case, then say the actual algorithm is $\leq$ what we calculated.

Example 4.4. Analyze the following algorithm. Test 1 (n).

```
sum = 0;
for i = 1 to n do
|  sum = sum + i
end
return sum
```

This program will run with $1 + (\sum_{i=1}^{n} 1) + 1 = 2 + n = \Theta(n)$ operations.

Example 4.5. Analyze the following algorithm.

```
sum = 0;
for i = 1 to n do
|  for j = 1 to i do
|  |  sum = sum + (i − j)^2;
|  |  sum = sum^2;
|  end
end
return sum
```

This program will run with time:

$$\begin{aligned}
&1 + \sum_{i=1}^{n} \left[\sum_{j=1}^{i} 4 \right] + 1 \\
&= 2 + \sum_{i=1}^{n} 4i \\
&= 2 + 4 \sum_{i=1}^{n} i \\
&= 2 + 4 \cdot \frac{n(n+1)}{2} \text{ (Gauss)} \\
&= 2 + 2n^2 + 2n \\
&= \Theta(n^2)
\end{aligned}$$

We can be pessimistic, assume worst-case scenario, and say that it runs with time:

$$\begin{aligned}
&2 + \sum_{i=1}^{n} n \\
&= O(n^2)
\end{aligned}$$

Note that $O(n^2)$ is a less precise result than $\Theta(n^2)$, but in some cases that's good enough.

Order notation helps make algorithm analysis easier by allowing us to throw away any specific constants, as long as the order remains untouched. The entire analysis process happens within the context of order notation, so you can just start dropping constants immediately.

Keep in mind, it is possible to create a bad over-estimation. You have to be smart about it.

Example 4.6. Analyze the following algorithm. Test 2 (A, n).

```
max = 0;
for i = 1 to n do
    for j = 1 to n do
        sum = 0;
        for k = i to j do
            sum = A[k] + sum;
            if sum > max then
                max = sum;
            end
        end
    end
end
return max
```

This is the **maximum subsequence problem**. The input is an array of integers $A[1 \dots n]$, and the output is the consecutive run with the largest sum.

Sample sequence:

2	-4	1	3	-2	8	-1

The running time of this program is $max\left\{ \sum_{i=1}^{j} A[k] \middle| 1 \leq i \leq j \leq n \right\}$.

← January 17, 2013

Example 4.7. Analyze the following algorithm.

```
for i = 1 to n do
    for j = 1 to i do L1[i]
        for k = 1 to j do L2[i, j]
            x = x + 1
        end
    end
end
```

$$
\begin{aligned}
\sum_{i=1}^{n} L_1(i) &= \sum_{i=1}^{n} \left[\sum_{j=1}^{i} L_2(i,j) \right] \\
&= \sum_{i=1}^{n} \left[\sum_{j=1}^{i} \left[\sum_{k=1}^{j} \Theta(1) \right] \right] \\
&= \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1 \\
&= \sum_{i=1}^{n} \sum_{j=1}^{i} j \\
&= \sum_{i=1}^{n} \frac{i(i+1)}{2} \\
&= \frac{1}{2} \sum_{i=1}^{n} (i^2 + i) \\
&= \frac{1}{2} \left[\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i \right] \\
&= \frac{1}{2} \left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \right] \\
&= \Theta(n^3)
\end{aligned}
$$

Alternatively, we could use a lazier process to determine that this algorithm is $O(n^3)$, wihch is less precise than saying the algorithm is $\Theta(n^3)$. The lazy way is to say each of the nested loops will run in $\leq n$ operations in the worst case, which can be multiplied out to n^3.

4.2 Math Review

4.2.1 Exponentiation

Exponents have a number of important properties, including:

$$
\begin{aligned}
b^0 &= 1 \\
b^1 &= b \\
b^{1/2} &= b^{0.5} = \sqrt{b} \\
b^{-1} &= \frac{1}{b} \\
b^a \cdot b^c &= b^{a+c} \\
(b^a)^c &= b^{ac}
\end{aligned}
$$

4.2.2 Logarithms

Definition. $\log_b a = c$ if and only if $a = b^c$. If $b = 2$, we write $\lg a$.

There are a number of log identities you should be aware of:

$$\log_b(a \cdot c) = \log_b a + \log_b c \tag{1}$$

$$\log\left(\frac{a}{c}\right) = \log_b a - \log_b c \tag{2}$$

$$\log(a^c) = c \cdot \log_b a \tag{3}$$

$$b^{\log_c a} = a^{\log_c b} \tag{4}$$

$$\log_b a = \frac{\log_c a}{\log_c b} \tag{5}$$

Identity (1) is useful because it allows you to go from multiplication to addition. You can avoid nasty multiplication operations by using this identity.

Identity (4) is the prof's favorite. If b and c are constants, $\log_b n = \Theta(\log_c n)$ – that is, we don't care about the base of the logarithm in the context of order notation.

4.2.3 Recursive Definitions

Definition. The **factorial** of a number n is represented by $n!$. Informally, $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$. More formally:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

Definition. The **fibonacci numbers** are defined by:

$$F_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ F_{i-2} + F_{i-1} & i > 1 \end{cases}$$

4.2.4 Summations

There are a few common summations you should be aware of:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1} = \Theta(n^{k+1})$$

You should also be familiar with the **geometric series**:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ where } a < 1$$

4.3 Useful Properties of Order Notation

1. $f(n) = \Theta(a \cdot f(n))$ for $a > 0$.
2. Transitivity. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
3. $[f(n) + g(n)] = \Theta(\max\{f(n), g(n)\})$
4. $a_0 + a_1 x^1 + a_2 x^2 + \ldots + a_n x^n = \Theta(x^n)$, where a_i are constants, $x > 1$, and $a_n > 0$. This is just a common case of (3).
5. $n^k = O(a^n)$ for $a > 1$.
6. $\log^k n = o(n^b)$ for $k > 0$ and $b > 0$ (where $b \in \mathbb{R}$, not necessarily the base of the logarithm).

You can use these properties in proofs *unless* you're requested to write a proof from first principles.

4.4 Orders We Aim For

When we are designing an algorithm, we aim for:

1. Constant time: $\Theta(1)$
2. Logarithmic time: $\Theta(\lg n)$
3. Poly log: $\Theta((\lg n)^k)$
4. Linear complexity: $\Theta(n)$
5. $n \log n$ complexity: $\Theta(n \lg n)$
6. Quadratic: $\Theta(n^2)$
7. Cubic: $\Theta(n^3)$
8. Polynomial: $\Theta(n^k)$
9. Exponential: $\Theta(b^n)$

4.5 Maximum Subsequence Problem

Recall the maximum subsequence problem from earlier:

```
max = −∞;
for i = 1 to n do
    for j = 1 to n do
        total = 0;
        for k = i to j do
            total = total + A[k];
            if total > max then
                max = total;
            end
        end
    end
end
return max
```

Also, recall our sample sequence A:

2	-4	1	3	-2	8	-1

Note that the longest run (the subsequence with the largest sum) in this sample sequence is from 1 to 8, which is a subsequence that sums to 10.

The maximum subsequence problem has many applications. It is used for DNA matching, for instance. There is a score for accuracy / closeness of the match. You get a score for various runs within a DNA sequence. If there is a long enough run(s) then you have probably identified the correct person. Google may also use the maximum subsequence problem as part of its algorithm for determining typos or other terms to match a particular search query (such as the word's plural form).

Our naïve solution is not good if n is sufficiently large, such as $10,000$ or $1,000,000,000,000$. Notice that the innermost loop recalculates the sum unnecessarily every time j is incremented. You really just need to add the new element to the previous result. That gives us this more efficient algorithm:

```
max = −∞;
for i = 1 to n do
    total = 0;
    for j = i to n do
        total = total + A[j];
        if total > max then
            max = total;
        end
    end
end
return max
```

This algorithm runs in $O(n^2)$ time. We could do better, but that's okay for now. Our code is now 10,000 times faster on an input of size 10,000.

5 Recursive Analysis

We are going to analyze **mergesort**. Recall: mergesort involves sorting the two halves separately, then merging the sorted items into a single list again. We merge items $(1 \ldots n/2)$ and $(n/2 + 1 \ldots n)$ separately, then merge the two lists.

The number of operations mergesort requires is:

$$\begin{aligned} T_{ms}(n) &= T_{ms}(\tfrac{n}{2}) + T_{ms}(\tfrac{n}{2}) + n \\ &= 2T_{ms}(\tfrac{n}{2}) + n \end{aligned}$$

$$T_{ms}(1) = 1 \text{ (base case)}$$

The method you use here is **guess and check**. For example, for $n = 4$, we have:

$$\begin{aligned} T_{ms}(4) &= 2T_{ms}(2) + 4 \\ &= 2(2T_{ms}(1) + 2) + 4 \\ &= 4T_{ms}(1) + 4 + 4 \\ &= 4 + 4 + 4 \end{aligned}$$

$$T_{ms}(8) = 8 + 8 + 8 + 8$$

$$T_{ms}(n) = n \cdot \lg n$$

You can check this with induction. That is, assuming this holds for $n = 2$, show that it holds for any n. If $T_{ms}(n) = n \lg n$, then:

$$\begin{aligned} T_{ms}(n) &= 2T_{ms}(\tfrac{n}{2}) + n \\ &= 2 \cdot \frac{n}{2} \cdot \lg \frac{n}{2} + n \\ &= n \cdot \lg \frac{n}{2} + n \\ &= n \cdot (\lg n - \log 2) + n \text{ by log identity} \\ &= n(\lg n - 1) + n \\ &= n \lg n - n + n \\ &= n \lg n \end{aligned}$$

Therefore, mergesort has time $\Theta(n \lg n)$.

6 Abstract Data Types

6.1 Stack

A **stack** is a collection of items, which supports the operations `push` (insert an item), `pop` (remove most recently inserted item), `peek` (look at the last item), and `isEmpty`. `push` and `pop` are the most commonly supported operations.

6.1.1 Stack Implementations

Common ways to implement a stack are:

- **Linked List**. You maintain a pointer to the top of the stack. When you `push` or `pop`, you change your pointer to the next or previously item as appropriate. This method is more flexible, but pointers can be a pain.
- **Array**. You keep track of where the last item is in the array. You may need to re-size your array, but at least you don't have to worry about pointers.

6.2 Queue

A **queue** is a data structure where you `insert` at the end and remove (`dequeue`) from the front, just like the way a queue (a lineup) works in real life.

Implementations are similar to implementing a stack. You maintain pointers to the start and end of the array or linked list.

6.3 Priority Queue

← January 22, 2013

A priority queue is a collection of elements similar to a queue, except each element has a priority assigned to it and elements are popped in order of priority (not in order of arrival).

A priority queue supports these operations:

- `insert(x, p)` – inserts item x with priority p.
- `deleteMin()` – deletes. Used when the queue is defined to be such that lower numbers indicate higher priority. This deletes the element with the lowest p.
- `deleteMax()` – deletes. Used when the queue is defined to be such that higher numbers indicate higher priority. This deletes the element with the highest p.
- `peek()` – view the top element without deleting it.

6.3.1 Applications of Priority Queues

- To-do lists in real life.
- To-do lists in the operating system (used for multitasking; the processor can only really do one thing at a time, so it does a bit of each task as determined by priority).
- Sorting. Insert the elements from an array A and then when you delete them you'll get them back in sorted order.

Priority queues can be used for sorting purposes. It allows us to sort without worrying about the underlying sorting implementation. However, if we want to know how good of a sorting algorithm it is, we would need to examine the implementation of `insert` and `deleteMin`.

Here's some pseudocode for sorting with a priority queue:

```
for i = 0 to n - 1 do
 |  PQ.insert(A[i], A[i]); // key and priority are the same
end
for i = 0 to n - 1 do
 |  A[i] = PQ.deleteMin();
end
```

6.3.2 Implementations of Priority Queues

1. Use **unsorted arrays**.

   - `insert(x, p)` takes O(1) time because you simply place the element at the end of the array.
   - `deleteMin()` takes O(n) time since you need to walk the array (keeping track of the current min you've seen so far), then replace the deleted element with the last element of the array. O(n) + O(1) = O(n) time.

   Sorting with unsorted arrays takes O(n^2) time, since for each element you're deleting you need to walk the entire array. Clearly, this is not a good sorting algorithm.

2. Use **heaps**. A heap is a complete binary tree that has the heap property.

   Recall: a **complete (or perfect) binary tree** is defined as a binary tree such that:

   - Every node has 0 or 2 children, except the rightmost leaf in the bottom level, which may have one only child.
   - All the leafs appear consecutively in the bottom two levels.
   - Deeper leaves are leftmost in the tree.

   You can think of a complete binary tree as trying to fill the tree from left-to-right, going deeper and deeper as necessary.

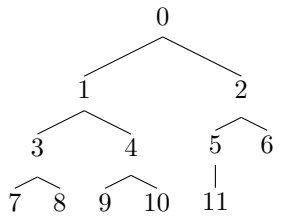


Figure 2: A complete binary tree.

The **heap property** is the idea that the priority of a node is higher than that of all of its children (if any). In a min-PQ, this means the p value of all children is a larger number than that of the parent. In a max-PQ, the childrens' p values must all be smaller than that of their parent.

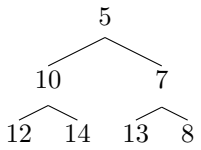


Figure 3: A min-PQ tree that satisfies the heap property.

You can have multiple elements with the same priority. You settle those disputes arbitrarily.

This is not a binary search tree, so the larger element does not need to be on the right. There is no order among siblings, as seen in Figure 3. There is also no relationship with the parent's siblings – it's only a relationship with the parent.

Next, we'll look into **filling holes**. If you delete an element from the tree, you delete the element at the top, but then you have a hole that needs to be filled. But how? We take the last element in the tree (the rightmost element at the bottom level) and swap it into the top position, then we get it to **boogie down**.

The process of **boogieing down** is where an element (in this case, the top element) looks at its children and swaps with a child that has lower priority than itself (if any). It does this recursively. It continues to recurse until the child is of higher priority than its parent, or until you reach a leaf node. Swap places with the child with the highest priority.

Insertion uses the process of **bubbling up**. We place the new element in the first available spot, which will be in the bottom level. The new element exchanges with their parent if the new element has higher priority than the parent, and that continues as far as needed, until it doesn't swap or until it reaches the top.

Implementing a heap as an array (also known as a **l33t hacking trick**):

You can actually implement a heap without pointers, using an array instead. The array indices follow the pattern as indicated in Figure 2 by inserting at the end of the array.

The parent of a node at index m is $\lfloor \frac{m-1}{2} \rfloor$. The children for a node at index m are located at index $2 \cdot m + 1$ and $2 \cdot m + 2$.

To perform a `deleteMin`, we erase element 0 and replace it with the last element in the array. We then perform a boogie down on element 0. Before computing children, we must perform boundary checking to ensure there actually are children (that is, we haven't run off the end of the array).

Here's the pseudocode for the bubble up process:

```
while ⌊(v−1)/2⌋ ≥ 0 and A[⌊(v−1)/2⌋].p > A[v].p do  [parent exists and has higher priority]
    swap(v, ⌊(v−1)/2⌋);
    v = ⌊(v−1)/2⌋;
end
```

Here's the pseudocode for the boogie down process:

```
while 2v + 1 ≤ last do  [v has at least one child]
    u = argmin{A[2v + 1], A[2v + 2]}; // argmin returns the array index of the min
    if A[u].p < A[v].p then
        swap(u, v);
        v = u;
    else
        break;
    end
end
```

When implementing boogieing down, you'll have to handle boundary cases where there is only one child, etc., which were not handled in this pseudocode.

7 Sorting

7.1 PQ Sort

We can sort data using a priority queue.

Both `insert` and `deleteMin` take O(lg n) time where n is the number of elements in the tree. Bubble up and boogie down each take O(h) = O(lg n) time, where h is the height of the tree.

Note that the number of elements in a tree is between 2^{h-1} and 2^h. PQ sort takes O(n lg n) time, which is the same as mergesort.

7.2 Heap Sort

← January 24, 2013

Heap sort is a specialization of PQ sort. We can actually insert things even faster into the heap, though.

Claim: if all the elements to be inserted are given at once we can build the heap in $\Theta(n)$ time, which is faster than $\Theta(n \lg n)$.

Proof. We're going to show that we can run this algorithm in $\Theta(n)$ time.

```
Heapify(A); // A is an array
n = size(A) - 1;
for i = ⌊(n−1)/2⌋ down to 0 do
    boogie_down(A, i);
end
```

The number of comparisons for boogie down is:

$$\frac{n}{2} + 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \ldots + 1 \cdot \lg n$$

$$\sum_{i=1}^{\lg n} \frac{n}{2^i} \cdot i$$

The cost of boogieing down n elements is:

$$T(n) = T\left(\frac{n}{2}\right) + \frac{n}{4} = T\left(\frac{n}{4}\right) + \frac{n}{8} + \frac{n}{4}$$

This is the case because there are $\frac{n}{4}$ elements that can go one level further down. So we have:

$$\begin{aligned} T\left(\frac{n}{2}\right) &= T\left(\frac{\frac{n}{2}}{2}\right) + \frac{\frac{n}{2}}{4} \\ &= T\left(\frac{n}{4}\right) + \frac{n}{8} \\ &= \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \frac{n}{32} + \ldots \\ &= n\left(\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \ldots\right) \end{aligned}$$

This is true because $\sum_{i=0}^{\infty} \frac{1}{2^i} = 1 + \frac{1}{2} + \frac{1}{4} + \ldots = 2$ was proven by Zeno.

Aside: Zeno's proof of this involved a human and a turtle walking. The human walked at a pace of 1 metre per hour and the turtle moved at a pace of a $\frac{1}{2}$ metre per hour. The turtle was given a head start. The human would catch up with the turtle at point 2, but not any sooner because both the human and the turtle will have made progress on their race by the time the human reaches a point where the turtle was previously. This is known as **Zeno's paradox**. Unrelated: someone has called the dust that you can never sweep into a dust pan zenodust.

An alternate way to prove this is as follows:

$$\begin{aligned} \sum_{i=0}^{\lg n} \frac{n}{x^i} \cdot i &= n \sum_{i=0}^{\lg n} \frac{1}{x^i} \cdot i \\ &= n \sum_{i=0}^{\lg n} \tau^i \cdot i \text{ where } \tau = \frac{1}{x} \end{aligned}$$

What we now have looks a bit like a derivative. Differentiating gives:

$$n \cdot \tau \sum_{i=0}^{\lg n} i \cdot 2\tau^{i-1}$$

We can integrate and then differentiate.

Therefore, we can heapify in $\Theta(n)$ time, so heap sort is $\Theta(n \lg n)$ and the constant will be lower than other methods, which is preferable. □

In practice, it's better to insert one by one, because our analysis is in the worst-case. When comparing the average case, inserting one by one is faster than heapifying. The worst case is really rare, so you should insert one by one (and pay more attention to the average case analysis in this situation).

A heap implemented using an array is called an **in-place implicit data structure**. You avoid pointers by using trickery in the organization of the data itself.

7.3 Quick Sort

Selection: suppose we want to find the minimum or maximum element in an unsorted array. The easy solution is to scan the array and report the answer in $\Theta(n)$ time. However, if you're asked to find the k-th largest element? The naïve solution would be to sort the entire array, which would take $\Theta(n \lg n)$ time.

We can do better. We can find the k-th largest element in an array in linear time in the worst case, using a method called quick select.

7.3.1 Quick Select

Given an array A, we want to find the k-th smallest (or k-th largest) element. For example, let's say we're looking for the 3rd smallest element and we're given this array A:

20	8	9	2	1	7	15	12

We now want to partition the array into two smaller sets of elements. We will first sort the elements onto the proper side of 20 (the first element):

8	9	2	1	7	15	12	20

Then, we'll continue on the side of 20 that happens to contain the element we're looking for. This time, we'll sort elements onto the proper side of 8 (which is the first element).

2	1	7	8	9	15	12	20

If we were to continue further, we would proceed on the left side of 8 because there are fewer (or equal) to 3 elements on that side.

We use a random element from the array called the **pivot** to partition the elements into those smaller than the pivot and those larger than the pivot. In general:

$<$ p	p	$>$ p

Here's some pseudocode for quick select:

```
p = choose random pivot in A;
partition (A, p) obtaining position i of p;
if k < i then
    quick_select(A[0, i - 1], k);
else if k > 1 then
    quick_select(A[i + 1, ..., n - 1], k - i - 1);
else
    return p;
end
```

In quick select, you only continue to recurse in the "half" that contains what we're looking for (unlike in quick sort where we continue to recurse on both sides).

In the worst case, this runs in $\Theta(n^2)$ time, since the random element chosen is always $\max(A)$, and we are searching for $k =$ first element:

...	p_3	p_2	p_1

Partitioning with two arrays is easy (i.e. creating a new array and inserting). Partitioning within the existing array is a bit more complicated, however that is a key advantage to quick select. Here's the pseudocode for partitioning within the same array:
A sample run of this algorithm is as follows:

```
swap(A[0], A[p]);
i = 1;
j = n - 1;
while true do
    while i < n and A[i] < A[0] do
        i = i + 1;
    end
    while j ≥ 1 and A[j] > A[0] do
        j = j - 1;
    end
    if j < i then
        break;
    else
        swap(A[i], A[j]);
    end
end
```

← January 29, 2013

```
44 4 10 52 86 57 97 19 18 88 31 82 59 39
19 i -->>           44           <<-- j
19 4 10 18 86 57 97 44 52 88 31 82 59 39
         i -->>-<>-<<--  j
4 10 18 19 86 57 97 44 52 88 31 82 59 39 (final output)
```

In-place partitioning is one the reasons why quick sort (and quick select) is so good. Recall the selection problem, where we have to find a way to return the k-th element in the sorted order. Quick select is one solution to this problem.

Worst case: quick select runs in $T(n) = n + T(n-1) = n + (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$ time.

Average case: assume all permutations are equally likely. This is a probabilistic assumption on the input. The probability of choosing a "good" pivot is $\frac{1}{2}$. After r recursive calls, about half the pivots are good, which means the array was halved approximately $\frac{r}{2}$ times. After $4 \lg n$ recursive calls, we must be left with an array of size 1. Note that this is $4 \lg n$ and not $2 \lg n$ because we aren't dividing perfectly in half each time. We're also assuming that randomness is somehow preserved on recursive calls.

Expected case: probabilities come from the algorithm itself. For example: select p at random in A[0...n - 1], which preserves randomness. Regardless of the input distribution or partition algorithm, a pivot is "good" with probability $\approx \frac{1}{2}$. The expected case is that this algorithm runs in $T(n,k)$ time as follows:

$$\begin{aligned} T(n,k) &= cn + \frac{1}{n}T(n-1,k) + \frac{1}{n}T(n-2,k) + \frac{1}{n}T(n-3,k) + \ldots + \frac{1}{n}T(n+1,k-1) + \frac{1}{n}T(n+2,k-2) + \ldots \\ &= cn + \frac{1}{n}\left[\sum_{i=0}^{k-1} T(n-i-1,k-i-1) + \sum_{i=k+1}^{n-1} T(i,k)\right] \end{aligned}$$

We could hand-wave and say:

$$\begin{aligned} T(n) &\le \frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + cn \\ &\le 2cn + 2cn\left(\frac{3n}{4}\right) + 2c\left(\frac{9n}{16}\right) + \ldots + T(1) \\ &\le \frac{1}{2}\left[\frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + cn\right] + \frac{1}{2}\left[\frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}T\left(\frac{9n}{16}\right) + c\frac{3n}{4}\right] \\ &\le \frac{1}{4}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}cn + \frac{1}{4}T\left(\frac{9n}{16}\right) + c\frac{3n}{8} + cn \\ &\le \underbrace{T(1)}_{\text{constant}} + 2cn \underbrace{\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i}_{\text{constant}} \\ &= \Theta(n) \end{aligned}$$

$\therefore$ We can quick select in linear time in the expected case.

We really don't need highly random bits for quick select – at least nowhere near as random as in cryptography. You generally need a good random source, however for this application (since security is not an issue), the current time may suffice as a random seed. If we can *easily* produce a sequence mechanically with a program, then it is not considered random. True randomness cannot be compressed.

7.3.2 Quick Sort Algorithm and Analysis

The general algorithm for quick sort is as follows.

if *n = 0* **then return**;
p = choose random_pivot(0 . . . n - 1);
i = partition(A, p);
QS(A[0 . . . i - 1]);
QS(A[i + 1 . . . n]);

In quick select, we only recursed on the side containing the target, but in quick sort, we want to recurse continuously on both sides until we every side contains only one element.

Worst case: $T(n) = n + T(n-1) = n + (n-1) + T(n-2) + \ldots = \Theta(n^2)$.

Best case: $T(n) = n + 2T\left(\frac{n}{2}\right) = \Theta(n \lg n)$.

Average case:

$$\begin{aligned} T(n) &= cn + \frac{1}{n}(T(n-1) + T(1)) + \frac{1}{n}(T(n-2) + T(2)) + \ldots + \frac{1}{n}(T(1) + T(n-1)) \\ &= \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1)) + \underbrace{\Theta(n)}_{\text{partition step}} \\ &\leq \frac{1}{n}\sum_{i=0}^{n-1} 2\max\{T(i), T(n-i-1)\} + \Theta(n) \\ &= \frac{1}{n} \cdot \frac{1}{2}(T(\frac{3i}{4})) + \frac{1}{n} \cdot \frac{1}{2}T(i) + cn \end{aligned}$$

Take note of two important observations here:

1. At each level of the recursion, the total work adds up to n (roughly).
2. This continues until we reach the end of the recursion at $n = 1$.

Every two levels, every block is half of its original size or less.

The worst case of quick sort is *extremely* rare. Alejandro ran an experiment, and out of one billion runs, the worst case took just twice the time of the average case.

Bubblesort is really bad. It's only taught because it has a cool name.

Mergesort cannot be improved upon, but quick sort can be improved by optimizing the algorithms in selecting pivots.

← January 31, 2013

To review, quick sort operates with the following runtimes:

- Average case: $\Theta(n \lg n)$, with a constant ≈ 1.4, on a uniform input distribution. However, we know the input is *not* uniform. One type of typical distribution would be an array containing the old customer list (sorted) followed by a new customer list, and we'd want to merge the two lists. That is not a uniform distribution.

- Expected case: $\Theta(n \lg n)$ if we choose the pivot position at random. This is the best option in general.
- Worst case: $\Theta(n \lg n)$, with a constant $\approx\in [7, 10]$, if you select the median in $\Theta(n)$ time. This constant is relatively large, which is why we typically avoid this algorithm.

Note that the constants only matter when we can't improve the order any further, as is the case here.

Can we do better than $\Theta(n \lg n)$? It depends on what you're sorting. But in general, no, regardless of the algorithm (even considering other sorting algorithms other than quick sort). There is a lower bound for the performance of *all* sorting algorithms by $\Omega(n \lg n)$.

Finding the minimum (or maximum) element in a list or unsorted array of numbers has a lower bound of $\Omega(n)$ because you need to look at all n numbers. We can find a similar lower bound (and rationale) for sorting.

7.4 Sorting in the Comparison Model

Elements to be sorted can be compared but no other operations are allowed.

Theorem. In the comparison model, it takes $\Omega(n \lg n)$ operations to sort n objects.

Recall the 20 questions game from childhood. If you're unaware, 20 questions is a game where:

- A mineral, plant, or animal is selected.
- The other person asks 20 yes/no questions.
- The other person aims to guess the mineral/plant/animal that was selected, within those 20 questions.

For the guess to be correct, it can only be made in nodes where the set of candidates is of size 1. That is, you must narrow the number of possibilities down to 1 prior to making a guess in order for your guess to be correct.

If the tree of questions is balanced, we could distinguish 2^{20} objects since there are 2^{20} leaves (maximum) on a tree of height 20. If you have a completely unbalanced tree, you can only distinguish between 20 objects in a tree of height 20. This is one of the reasons why mergesort is good.

Fundamentally, shuffling and sorting are the same operation, just backwards. Every sorting sequence corresponds to a unique shuffle sequence. How many different shuffles do we have for a set that contains n elements? $n!$.

Let's consider a comparison-based sorting algorithm on a set where all the keys are distinct. We get a tree where the root is $a_i : a_j$ and the two child nodes are $a_i < a_j$ and $a_i > a_j$. As we get further down the tree, these comparisons all compound to narrow down the set. The algorithm determines the next question to ask.

$a_i : a_j$ is just one way of representing that we're doing a general comparison. Sometimes we also use the notation $a_i ? a_j$ or $a_i \gtreqless a_j$.

At the end of the comparisons, we *must* have a single unique candidate shuffle, namely the sequence that is unshuffled by the algorithm as a result of those comparisons.

We need a tree with $n!$ leaves that is as shallow as possible. More specifically:

$$h = \log(n!) = \log\left(\frac{n^n}{e^n}\right) = \log n^n - \log e^n = n \log n - n \log e = \Theta\left(\left(\frac{n}{e}\right)^n\right)$$

This is true because of Stirling's approximation.

$\therefore$ As long as your sorting algorithm only uses comparison operators, $\Omega(n \lg n)$ is the best you can do.

When making an argument like this, you need to make arguments that apply across *all* algorithms, even ones that no one has seen or thought of yet. That's what we did here. When you're discussing lower bounds, you need to be extremely clear which operations are and aren't allowed.

7.5 Counting Sort

Counting sort is an example of a sorting algorithm that runs faster than $n \lg n$, however it only applies in certain circumstances.

- You are given a permutation of the numbers $1 \ldots n$. This runs in $\Theta(n)$ or $\Theta(1)$ time, depending on which operations you're allowed to use. If we're given the correct operations, we don't even need to look at the input in this case.
- Keys given belong to a set of size $O(n)$, allowing possible repetition.

Here's an algorithm for `counting_sort(A, k)`, where A is an array and k is the size of the universe (not the number of elements).

```
c[0 ... k - 1] = 0⃗; // Θ(n) = O(k)
for i = 0 to n - 1 do
    // Θ(n)
    c[A[i]]++;
end
for i = 1 to k - 1 do
    // Θ(n) < Θ(k)
    c[i] = c[i] + c[i - 1];
end
B = (copy A); // Θ(n)
for i = n - 1 down to 0 do
    // Θ(n)
    decrement C[B[i]];
    A[c[B[i]]] = B[i]
end
```

This works well when you have a dense set such as student IDs or street addresses. It would not work well for student names because there are too many possible values. ← February 5, 2013

You may be interested in this webpage from Kent State University about counting sort.

Counting sort works by walking the array A, and for each element increment $c[A[i]]$, where c is an array of counters. c is then updated to be the number of elements that are less than or equal to the index of c.

Counting sort is stable. That means that when there is a tie between two keys, they appear in the order of the input. This intuitively makes sense in a lot of circumstances because if you have two people with the same priority in a queue (a lineup), you want to process them in the order they arrived. That's the motivation for the concept of stability. Straight-forward quick sort algorithms, as seen in class, are not stable.

Counting sort takes time $\Theta(n+k) = \Theta(n)$ if $k = O(n)$.

7.6 Radix Sort

There are situations where you want to sort larger numbers. For example: you have 20 students, but student IDs are ten digits long. Counting sort would be a terrible application in this case because you'd need a huge array, so we'll use **radix sort** instead. Radix sort is good when you have a fixed-length key space.

7.7 MSD Radix Sort

Most significant digit radix sort, also known as RSD radix sort, is a sorting algorithm where the elements are sorted into buckets based on their most significant digit first. That is, elements are sorted into buckets (using bucket sort) containing elements starting with 1, elements starting with 2, and so on, up until the bucket with elements starting with 9.

You can recursively apply bucket sort to complete the sort.

Example 7.1. Let's say you have the following student IDs:

15	25	12	45	99	66	21

Using bucket sort on the most significant digit, these IDs will be sorted into the following buckets:

12 15	25 21		45		66		99

7.8 LSD Radix Sort

Least significant digit radix sort, also known as LSD radix sort, is a sorting algorithm where the elements are sorted into buckets based on their least significant digit first. That is, elements are sorted into buckets (using bucket sort) containing elements ending with 1, elements ending with 2, and so on, up until the bucket with elements ending with 9. LSD radix sort is so common that it's often referred to simply as "radix sort".

You can recursively apply bucket sort to complete the sort.

You may be interested in this webpage about radix sort from Kent State University.

Example 7.2. Once again, let's say you have the following student IDs:

15	45	12	25	99	66	21

Using bucket sort on the least significant digit, these IDs will be sorted into the following buckets:

Pass 1:	21	12			15 45 25	66		99
Pass 2:	12 15	21 25		45		66		99

With LSD radix sort, each subsequent application occurs on the entire input in the order of the previous pass of bucket sort.

The time to sort d-digit numbers numbers using LSD is $\Theta(n \cdot d)$.

LSD radix sort only works when used in conjunction with a stable sort like counting sort. The reason this works is because at the last step, the first digits are tied. Ties are settled by the order of the next-most significant digit, since counting sort is stable and preserved the order from the previous pass.

Example 7.3. Here's another example of LSD radix sort.

INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

MSD radix sort operates at approximately the same complexity as LSD radix sort, but LSD is easier to implement, so it's more widely used.

LSD radix sort requires fixed key lengths. It's not always safe to assume that key lengths will be fixed. For example, phone numbers were created under the assumption that there would be one per household. But now, everyone has at least one cell phone (maybe multiple cell phones for different countries), and multiple mobile data devices like tablets or mobile data USB sticks. Similarly, it's not safe to assume that surnames can only be n characters long.

Earlier, we said LSD radix sort takes $\Theta(n \cdot d)$ time to sort. If you have n distinct numbers, $d = \Omega(\log_b n)$. So, if your numbers are distinct, you end up complexity $\Theta(n \cdot d) =$

$\Theta(n \log_{10} n) = \Theta(n \lg n)$.

It would appear as if no progress has been made, at least if all numbers are distinct. However, note that $\Theta(n \lg n)$ is the *best case* for quick sort, but it's the worst case here. So, we have made some progress.

Aside: if you think about comparing two or more elements at a time (like a bitwise comparison), it could be possible to sort in $\Theta(n \lg \lg n)$ time.

8 Dictionaries

The dictionary abstract data type is the most important and most commonly used abstract data type of all time.

A dictionary is a collection of items, each consisting of a key and some data. The `(key, data)` construct is called a **key-value pair**. Keys can be compared and are usually unique (unless explicitly stated otherwise). A database is essentially a collection of dictionaries.

8.1 Supported Operations of Dictionaries

A dictionary must support these operations:

- `search(key)`. Returns the data associated with the `key`, if the `key` is present in the dictionary.
- `insert(key, data)`. Inserts the `(key, data)` key-value pair into the dictionary.
- `delete(key)`. Deletes the `key` from the dictionary.

8.2 Implementations of Dictionaries

There are many different ways that dictionaries could be implemented.

1. **Unordered array**.
   - `search` takes $\Theta(n)$ time.
   - `insert` takes $\Theta(1)$ time, by inserting at the end of the array.
   - `delete` takes $\Theta(n)$ time, by searching, removing, then compacting the array.
2. **Ordered array**.
   - `search` takes $\Theta(\lg n)$ time by using binary search.
   - `insert` takes $\Theta(n)$ time.
   - `delete` takes $\Theta(n)$ time, by compacting and possibly searching, too.
3. **Linked list**.
   - `search` takes $\Theta(n)$ time.
   - `insert` takes $\Theta(1)$ time.

- `delete` takes $\Theta(n)$ time if it needs to find and delete the element, or $\Theta(1)$ time if it already has a pointer to the element.

There is a trade-off with all of these implementations. If one operation is slow, the other is fast.

People thought it should be possible to achieve:

- `search` in $\Theta(\lg n)$ time (with a bigger constant than with an ordered array).
- `insert` in $\Theta(\lg n)$ time.
- `delete` in $\Theta(\lg n)$ time.

These operations are possible in those complexities with binary search trees in the worst case. The main property of binary search trees is that if k_1 is the root/parent element of k_2 and k_3, then $k_2 \leq k_1 \leq k_3$.

9 Trees

← February 7, 2013

9.1 Binary Search Trees and AVL Trees

Binary search trees take $\Theta(\lg n)$ time for `insert` and `delete` in the average case. However, we can achieve those complexities in the worst case if we get nicely balanced trees as seen in Figure 4.

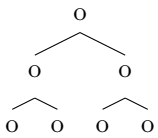


Figure 4: A balanced binary tree.

An example of an unbalanced tree was if we inserted 7, then 6, then 5, then 4, then 3, then 2, then 1, which would produce a single path of length 7.

In 1962, a group of Russian mathematicians came up with **AVL trees**. AVL stood for Adelson-Velski and Landis, the two Russian guys who came up with AVL trees. They were trying to build a chess program and had a need for balanced binary search trees.

They succeeded because they were tolerable of slight imperfections in their balanced trees. They had a relaxed attitude.

Definition. The **height of a tree** is the length of the longest path from the root to a leaf.

They demanded height-balanced trees.

Definition. AVL trees were binary search trees where the difference between the height of the left and right subtrees is at most 1.

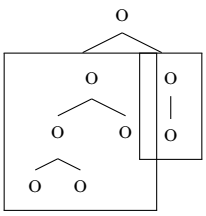


Figure 5: A tree with the left subtree having $h = 2$ and the right subtree having $h = 1$

The people in the West would have freaked out at the tree as seen in Figure 5.

In an AVL tree, at each non-empty node we store the result of the expression height(R) − height(L), so that:

- -1 means the tree is left-heavy.
- 0 means the tree is "balanced".
- +1 means the tree is right-heavy.

If the difference is some other value (± 2, etc.), then the tree is skewed enough that the tree is no longer an AVL tree. It needs to be re-balanced slightly in order to become an AVL tree again.

Insertion on an AVL tree: first, do a standard BST insertion. Then, recompute all of the height (balance) expressions. If the expressions are all still $\in \{-1, 0, 1\}$, then do nothing. Otherwise, we need to rebalance.

9.1.1 Right Rotation

If the balance of a particular subtree is -2, then a right rotation should be applied to that subtree.

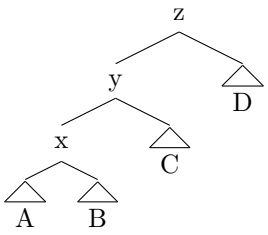


Figure 6: A right rotation needs to be applied to this entire tree.

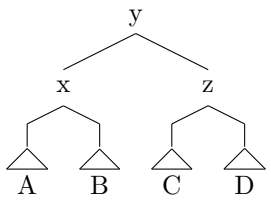


Figure 7: After a right rotation has been applied to the tree in Figure 6.

Note that the BST property has been maintained in Figure 7. Imagine a pulley sitting under z in Figure 6, and C got caught in the pulley and landed on the other side (under z).

We can use a constant number of operations (a few pointer changes) to rebalance the tree.

A right rotation of a tree has the following pseudocode:

```
new_root = z.left;
z.left = new_root.right;
new_root.right = z;
return new_root;
```

Similarly, we have left rotations.

9.1.2 Left Rotation

If the balance of a particular subtree is +2, then a left rotation should be applied to that subtree.

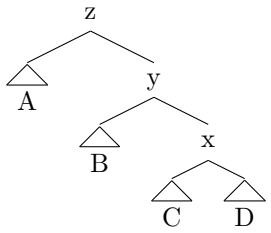


Figure 8: A left rotation needs to be applied to this entire tree.

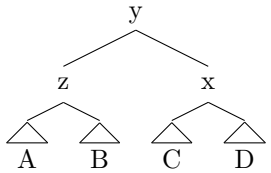


Figure 9: After a left rotation has been applied to the tree in Figure 8.

.

9.1.3 Double-right Rotation

If the inner-left side is heavy (and the balance for the tree is -2), a double-right rotation should be applied to that subtree. A normal right rotation does not property rebalance the tree.

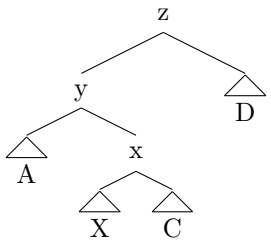


Figure 10: A double-right rotation needs to be applied to this entire tree.

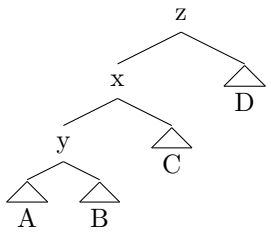


Figure 11: After a left rotation has been applied on the left subtree of the tree in Figure 10.

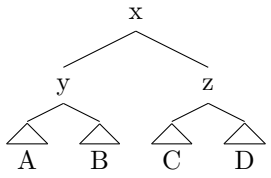


Figure 12: After right rotation has been applied to the tree in Figure 11. The double-right rotation is now complete.

Essentially, a double-right rotation involves a left rotation of the left subtree, then a right rotation of the entire tree.

9.1.4 Double-left Rotation

If the inner-right side is heavy (and the balance for the tree is +2), a double-left rotation should be applied to that subtree. A normal left rotation does not properly rebalance the tree.

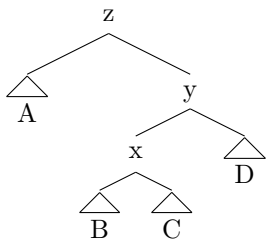


Figure 13: A double-left rotation needs to be applied to this entire tree.

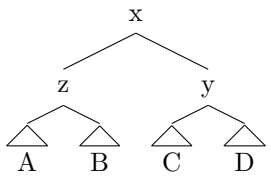


Figure 14: After a double-left rotation has been applied to the tree in Figure 13.

Note that for all of these rotations, you only perform the rotation at the node level where the balance has been broken.

Example 9.1. Let's say you just did a BST insertion of element 4.5, and now need to rebalance. A double-left rotation must be used because the tree is inner-right heavy.

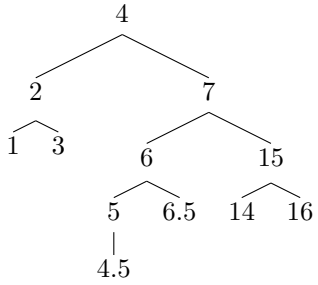


Figure 15: A double-left rotation needs to be applied to this entire tree.

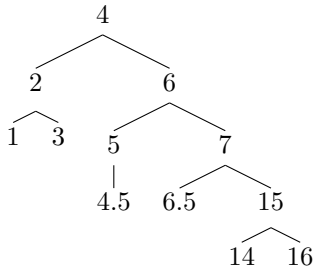


Figure 16: A right rotation was applied on the right subtree of the tree in Figure 15.

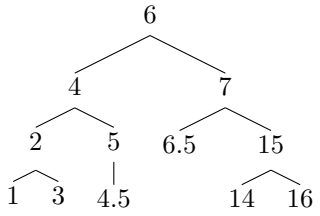
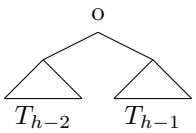


Figure 17: A left rotation was applied on the entire tree in Figure 16. The double-left rotation is now complete.

9.1.5 Other Tidbits about AVL Trees

What is the relationship between the longest path in an AVL tree and the number of elements in it, in the worst case? We would like the path length to be $\lg n$ for a tree of n elements. We want to avoid long paths with few elements (where height = number of elements, for instance).

Observation: the skinniest AVL tree T_h of depth h looks like this:



That is, the left subtree has depth $h-2$ and the right subtree has depth $h-1$. We get that $T_h = 1 + T_{h-2} + T_{h-1} = 1 + T_{h-1} + T_{h-2}$, which looks similar to the definition of the fibonacci numbers ($F_n = 1 + F_{n-1} + F_{n-2}$).

The smallest item in the left subtree is T_{h-2} because it satisfies the AVL property. T_{h-3} would be smaller, but it would not satisfy the AVL property as necessary.

Mathematicians discovered that $F_n = (\varphi)^n$ (the golden ratio φ to the n-th power), so we can say that $T_h \approx (\varphi)^n$.

The number of nodes $n = c^h$ (where c is some constant) in the skinniest AVL tree is:

$$\lg n = \lg c^h = h \lg c$$
$$h = \frac{\lg n}{\lg c} = \Theta(\lg n)$$

In practice, you probably won't implement an AVL tree. There are easier implementations, such as left red black trees and AA trees, however we look at AVL trees because they are more interesting theoretically. People also don't use any sort of binary tree in practice. It's more common to use a tree with more than two children.

← February 12, 2013

You may be interested in looking at this webpage on the cases of rotation for AVL trees. Note that after performing a BST insertion on the leftmost subtree then rotating right, the height will not change. Its ancestors will not see any change.

Upon insertion, after the first rebalancing operation the entire AVL tree is once again balanced. In contrast, after a deletion you might have to rotate every node in the path up to the root node in the worst case. In the best case, just a few rotations may suffice. This can be proven by the diagrams on the webpage about the cases of rotation for AVL trees.

9.2 Handling RAM Constraints

To many, a computer is a screen and a big box. To us, a computer is really an array M, which is RAM, plus a CPU and a hard drive.

When we write programs in a language like C++, our code is converted to machine instructions that operate on that array M.

```
C++:        Machine instructions:
int x;
x = 8;      MOVE 8, 435
x++;        INC 1, 435
```

Reading from a hard drive (or any secondary storage device) is much slower than reading from RAM. Hard drives operate at approximately 14,400 rpm, which is 240 rps. If we are looking for some data on a hard drive and just missed it as it was rotating, we have to wait for a full revolution to get the data. Each revolution takes $\approx 4ms$. The head moves even slower than that.

Hard drives are made up of several platters on an axle, plus a head. Each platter is divided into tracks (rings) and sectors (pie slices). When reading from a hard drive, we must read an *entire* sector at once, even if we aren't interested in all of the data on that sector.

As a rule of thumb, hard drives are 10,000x slower than RAM. This ratio has remained fairly constant because as hard drives have been improving at a similar progression as RAM. So, you want to avoid interacting with the hard drive as much as possible. There is some good news: solid state drives (SSDs). SSDs are "only" 100s of times slower than RAM, so dropping down to a SSD isn't as tragic as a traditional hard drive, but it still isn't as efficient as RAM.

If you're building a binary search tree over a large amount of data, a hard drive must come into play. We define "large" here as being an amount of data that is large enough that it cannot fit into RAM in its entirety.

Mergesort has efficiency $O(n \lg n)$. In reality, the performance is $n \lg n$ until the input size n hits the RAM limit, then the performance spikes. After this performance spike, mergesort continues to operate in $O(n \lg n)$ time, but with a much larger constant like $10000n \lg n$. This performance hit is a result of having to drop down to the performance of a hard drive. Some machines will have multiple spikes, between L1 and L2 cache, L2 cache and RAM, and RAM and a hard drive (from the fastest storage to the slowest storage).

When defining data structures, it's important to think about what will happen in terms of RAM and hard drive usage.

Let's say you have a binary search tree, stored in memory. When you insert a new element, let's say no more RAM is available, so the element needs to be placed on disk instead. This new element may contain pointers to other elements which are stored on other sectors of the hard drive. In order to follow this path, we would have to read several entire sectors, despite only being interested in a very small portion of each of those sectors.

For reference: the size of a sector is typically between 512 bytes and 4,000 bytes, and this size has remained fairly constant as time has progressed.

Idea: make the nodes exactly the size of the sector by changing the binary search tree into a multi-way tree (to utilize the extra space).

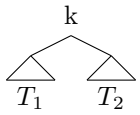


Figure 18: A typical binary search tree.

In Figure 18, $T_1 \in (-\infty, k)$ and $T_2 \in (k, +\infty)$. In any given BST, we need 1 key and 1 or 2 children. We could extend this to a multi-way tree.

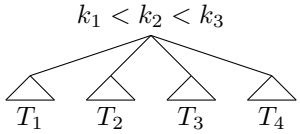


Figure 19: A multi-way tree.

In Figure 19, there are 3 keys and 4 children. However, to generalize, a multi-way tree with k keys needs $k + 1$ or $k + 2$ children.

In a binary search tree, we have $O(\log_2 n)$ complexity. In a multi-way tree (with k keys), we have $O(\log_{k+1})$ complexity. For a 20-way tree, for example, $\log_{20} n \geq \log_2 n$, but the constant will be much smaller for $\log_{20} n$. This smaller constant means we will hit the hard drive fewer times, which is a net win.

CD-ROMs used to take as long to read as music CDs did. They implemented an indexing system using a binary search tree, which was too slow. It was relatively easy to make CD-ROMs quicker (2x, 4x, ...) by overclocking the spinning mechanism. However, the queries were not being processed any faster because the head was slow. For music CD players, heads didn't need to be very fast. In fact, heads really haven't improved much since then.

The index was being developed for the Oxford English Dictionary (OED). They wanted OED on a CD-ROM and they wanted it to be fast. They aimed to be able to store all levels of the tree in RAM except for the deepest level, which meant you'd only need to hit the hard drive once for each query ($\approx$ 250 ms hard drive delay per query). A full dictionary lookup could occur in < 1 second, whereas before it took one and a half minutes.

To be cute, they also added "queried in 0.3 seconds" to their interface. Frank Tompa, a UW professor, worked on this project. The OED technology effectively enabled the first search engines.

9.3 2-3 Trees

2-3 trees were based on several key principles:

- Every node contains one or two keys and two or three children.
- All the leaves are at the same level (like a heap).
- The sloppiness permitted was the variable number of keys.

If we were to perform a BST insertion into the deepest leftmost position, we'd end up with the tree in Figure 20.

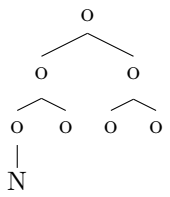


Figure 20: A binary search tree with leaves at different levels.

Performing the same insertion on a 2-3 tree would result in the tree as seen in Figure 21.

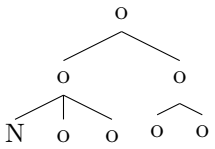


Figure 21: A 2-3 tree.

Insertion:

1. Follow the path down the tree according to key ranges until you find the parent of the leaves.
2. Insert there.
3. Fix your free, if needed.

Let's try inserting two elements. After step 1, we end up with the tree as seen in Figure 22.

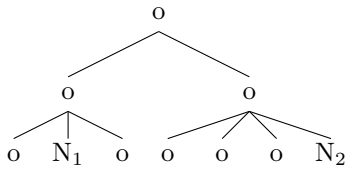


Figure 22: A 2-3 tree that needs fixing after two insertions.

Notice that the right subtree has four children, which is not allowed. We can fix this by splitting that node into two, each with two children. Then the root node will have three children (which is acceptable), and each subtree will have 2 or 3 children. The resulting tree is shown in Figure 23.

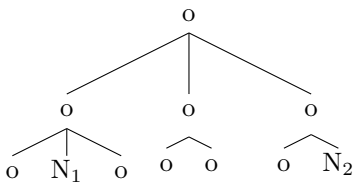


Figure 23: The 2-3 tree from Figure 22 after its right subtree was fixed.

We continue to split our parents up as necessary, and create a root node if necessary (if the root node was split).

Deletion: If we remove a node that causes one subtree to only have one child, we ask the sibling subtrees how many children they have. If the sibling we ask has just two children, we shove our single child onto the sibling. Otherwise, the sibling must have three children, in which case we take one of those children.

Let's look at an example where we delete an element and the sibling has just two children. Immediately after deleting the node, we're left with Figure 24, whose right subtree needs adjusting.

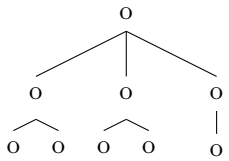


Figure 24: A 2-3 tree that needs fixing after the deepest rightmost element was deleted.

The right subtree of Figure 24 needs adjusting because no subtree in a 2-3 tree is allowed to only have one child. So, we shove this child onto our sibling, as seen in Figure 25.

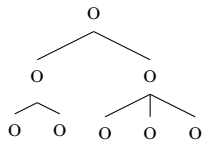


Figure 25: The 2-3 tree from Figure 24 after its right subtree has been fixed.

After a deletion, you need to recurse upwards ensuring that are all parents still satisfy all requirements of 2-3 trees. At the very top of the tree, you may have a case like in Figure 26.

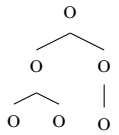


Figure 26: A 2-3 tree with the deepest rightmost node removed (needs fixing).

We can push the child of the right subtree onto the left subtree, as seen in Figure 27.

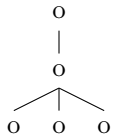


Figure 27: The 2-3 tree as seen in Figure 26 with its right subtree merged into the left subtree (still needs fixing).

We still need to adjust this further, however. The root node now only has one child, so we can get rid of it, as seen in Figure 28.

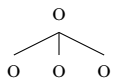


Figure 28: The 2-3 tree as seen in Figure 26 and Figure 27, after it has been fully adjusted.

We want to conserve RAM by storing only the keys in the internal nodes, instead of a key-value pair. All data resides in the leaves, which is the only layer that will be accessed on the hard drive. We're doing this to avoid the hard drive by storing as many levels as possible in RAM. In order to do this, we would have a many-way tree like the one seen in Figure 29.

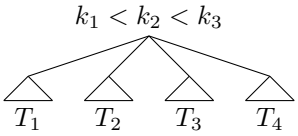


Figure 29: A many-way tree that stores all data in its leaves.

In Figure 29, T_1 is a subtree with range $(-\infty, k_1]$, T_2 has range $(k_1, k_2]$, T_3 has range $(k_2, k_3]$, and T_4 has range $(k_3, +\infty)$. Note that these ranges are all inclusive now, whereas before they were exclusive.

In a tree like the one in Figure 29, if you find the key you're looking for, you know you're on the right path. The data lies further down the path – not at the node you already found.

9.4 Comparison of Trees

← February 14, 2013

Binary Search Trees:

- Each node has 0, 1, or 2 children.
- Every node contains one key and some data, called a key-value pair.
- If the BST is an AVL tree, then the children are balanced (their height does not differ by more than one).

2-3 Trees:

- Each node has 0, 2, or 3 children.
- Every node has one or two key-value pairs.
- All leaves are at the same level.

B-Trees:

- Similar to 2-3 trees, but they contain between $\frac{m}{2}$ and $m-1$ key-value pairs per node.
- Each node can contain up to m children.

B^+-Trees:

- The internal nodes only have keys.
- Key-value pairs are present in the leaf nodes only.
- The root node of a subtree is a marker for which subtree to navigate to.
- Pointers to siblings are just for convenience.
- This is an example of an "algorithm for external memory" (hard drives and other secondary storage).

In a B-tree of height h, what is the minimum number of nodes in the tree? $2+2\cdot2+2\cdot2\cdot2+\ldots$, since the root has two children, and each of those has two children, and so on. This leaves

us with the minimum number of nodes in the tree being:

$$2 + 2 \cdot 2 + 2 \cdot 2 \cdot 2 + \ldots = \sum_{i=1}^{h} \left(\frac{m}{2}\right)^i \approx \left(\frac{m}{2}\right)^{h+1}$$

$$\left(\frac{m}{2}\right)^{h+1} \geq n$$

$$(h+1) \lg \frac{m}{2} \leq \lg n$$

$$h + 1 = \frac{\lg n}{\lg \frac{m}{2}}$$

$$h \approx \frac{\lg n}{\lg m - 1} - 1$$

In a B-tree of height h, what is the maximum number of nodes in the tree? $4{+}4{\cdot}4{+}4{\cdot}4{\cdot}4{+}\ldots$, following the same logic as before. This leaves us with:

$$4 + 4 \cdot 4 + 4 \cdot 4 \cdot 4 + \ldots = \sum_{i=0}^{h} m^i \approx m^{h+1}$$

$$m^{h+1} = n$$

$$(h+1) \lg m = \lg n$$

$$h \approx \frac{\lg n}{\lg m} - 1 \approx \log_m n$$

10 Hashing

Hashing is another dictionary implementation. It also has the same operations as a typical dictionary, including `search(k)`, `insert(k, data)`, and `delete(k)`.

Theorem. In the comparison model for keys, $\Omega(\lg n)$ comparisons are required to search a dictionary of size n.

Assumption: every key k is an integer k with $0 \leq k < m$ for some fixed constant m.

We implement this dictionary using an array A as our data structure, containing all of the dictionary's data values.

The naïve approach is to use direct addressing by making the array of size m. The operations would be implemented in the expected way:

- `insert(k, data)` would simply set the k-th index of the array to the `data`.
- `search(k)` would check the k-th index of the array, returning empty if the array's k-th index is empty (NULL), or returning the k-th element if it's found.
- `delete(k)` would set the k-th index of the array to be empty.

This approach is okay if $n \approx m$, but if n is sufficiently less than m, we should use hashing instead.

We define a hash function $h : \{0, 1, \ldots, m-1\} \to \{1, 2, \ldots, \Theta(n)\}$. That is, we define $h : U \to 0 \ldots |A| = 0 \ldots m-1$, where U is the key universe.

Using the hashing method, we use the same code as before except instead of using the k-th index of the array, we use the $h(k)$-th index for insertion, searching, and deletion.

We say there is a **collision** in a hash function if two different keys k_1 and k_2 exist such that $k_1 \neq k_2$ but $h(k_1) = h(k_2)$. Even if you have a nice hash function, collisions *will* happen, so we need to deal with them in some way. For example, with the birthday paradox, $\sqrt{m}$ keys have a collision.

Choosing a good hash function is hard. Let's look for an $h : \{ \text{ keys } \} \to \{0, 1, \ldots, m\}$ (where m is the size of the hash table). There are two main approaches. ← February 26, 2013

1. **h(k) = k mod m**. We know that $\mod m$ behaves a lot nicer if m is prime, so we're interested in choosing a prime m that is large enough to be the size of the table.

   When you need to resize the hash table, you will often choose to double the hash table size. However, that would mean we'd have to double m and then find the next prime that is larger than m. In practice, you may see pre-computed primes hard-coded into an array in a program for this purpose.

2. **Multiplication Method**. Choose $h(k)$ to be defined by:

   $$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

   Mathematicians have found that choosing $A = \frac{\sqrt{5}-1}{2} \approx 0.618$ is an ideal choice.

It's possible to choose better hash functions, but these two methods are good enough for our purposes.

Collisions are unavoidable (in the expected case). The first collision will occur after $\sqrt{\frac{\pi m}{2}} \approx 1.25\sqrt{m}$ items have been inserted, in the expected case. With the birthday paradox, that means you're likely to expect a collision after 24.

Speaking of the birthday paradox, buying baseball cards is a sort of reverse birthday paradox. The card company wants you to get many duplicates, so you'll keep buying more cards until you're lucky enough to get one of every card. That'll take n^2 cards (in the expected case) to get a full deck of n unique cards.

10.1 Collision Resolution Strategies

10.1.1 Seperate Chaining

The table contains n pointers, each pointing to an unsorted linked list.

When you want to insert a student whose hashed student ID is 5, you'll go to the 5th pointer and set the linked list element. If another student with the same hashed ID is inserted, you append that student to the linked list pointed to by the 5th pointer. In practice,

the new student is usually prepended for simplicity – to avoid walking down the list unnecessarily.

These linked lists would affect the run time if they become large. You could always make the linked list a sorted linked list if you care less about insert complexity and more about search complexity. In general, this solution is okay as long as the linked lists don't become too large.

The worst case linked list chains are $O(\lg n)$ in length, and many pointers would remain NULL. However, having a lot of pointers to NULL wastes memory, which isn't good.

The load factor of the hash table is $\alpha = \frac{n}{m}$, where n is the number of elements and m is the size of the hash table.

Searches are performed in $\Theta(1 + \alpha)$ complexity in the average case and in $\Theta(n)$ in the worst case.

Inserting runs in constant $\Theta(1)$ complexity because we're simply inserting into an unsorted linked list.

Deletion takes the same time as searching plus constant time $\Theta(1)$. Deletion requires finding the key in the linked list and then using one constant operation to remove it.

If we ensure that $m = \Theta(n)$, then $\alpha = \Theta(1)$. That is, the run times will be constant in the average case. But how do we keep α constant? We'd have to re-size the hash table, but that takes time.

10.1.2 Open Addressing

This is a more space efficient method because the table now contains the records themselves.

When we try to insert a second element with the same hashed key, we check the next element in the table to see if it's free. We continue to do this until we find a free element, and then we insert the new element there. This is called **linear probing**. That is, key k lives in position $h(k)$ if it's available, otherwise it might live in the $h(k + 1)$-th position if that's available, or $h(k+2)$ if that's available, and so on, until we find the first index greater or equal to k that is free.

When performing a search, we check the hashed index, and continue checking until we find an empty (free) element.

Be careful when your table fills up. The hash table object should maintain a counter of how full it is and silently increase (or prepare itself to throw an error). The hash table should do this without any operation (that is, *before* the next `insert` operation is even called). The hash table should never be left idle and full at the same time.

Linear probing has two main advantages: it uses space more efficiently, and it's quite easy to implement.

In reality, you'll get clusters of full elements – many elements with no free elements between them. You want to keep the clusters small to keep things (both `insert` and `search` operations) speedy. The bigger the cluster gets, the quicker it will continue to grow. Why is this? The bigger the cluster, the larger the area, the larger chance that a new insertion will lie within that cluster. It's a vicious cycle.

Theorem. The average number of probes required to search in a hash table of size m with n keys ($\alpha = \frac{n}{m}$) is approximately $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ (in the case of a successful search) or $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ (in the case of an unsuccessful search) for α not too close to 1.

Why is this? If we have two clusters each growing larger and larger, they will eventually run into each other, causing longer search times when looking for elements whose hashed keys lie in what was previously the first cluster. You'd be forced to search both the first and second clusters (since there are no free elements between them anymore), causing delays.

With different load factors α, the number of elements that need to be checked for a hit (successful search) and a miss (unsuccessful search) are as follows:

α:	1/2	2/3	3/4	9/10
hit:	1.5	2.0	3.0	5.5
miss:	2.5	5.0	8.5	55.5

So, if your hash table is very close to being full, misses become very costly.

10.1.3 Double Hashing

Idea: stop the formation of clusters by using two hash functions h_1 and h_2.

The key k now lives in position $h_1(k)$ or $h_1(k) + h_2(k)$. Observe that in general, even if $h_1(k_1) = h_1(k_2)$ (that is, you have a collision), $h_2(k_1) \neq h_2(k_2)$, since they are different hash functions. It's unlikely that you'll have a collision in both the primary and secondary hashes, although it is possible.

More formally, we define $h_1 : \{ \text{ set of keys } \} \to \{0, 1, \ldots, m-1\}$ and $h_2 : \{ \text{ set of keys } \} \to \{1, \ldots, m-1\}$. Notice that the definition of h_2 excludes zero in the range because h_2 is a displacement and we don't want to cause a displacement of zero (since that element is already full).

What happens if the secondary ($h_1(k) + h_2(k)$-th position) is already occupied as well? We continue to add $h_2(k)$ until an empty position is found. Searching works the same way – you keep adding h_2 until you find an empty element.

If we wanted to support deletion, we wouldn't actually delete the element – we would simply mark it as deleted. In general, hash tables like these don't support deletion because it's inconvenient.

Theorem. The average number of probes required to search successfully using double hashing is approximately $\frac{1}{2}\ln\left(\frac{1}{1-\alpha}\right)$. The average number of probes required to search unsuccessfully using double hashing is $\frac{1}{1-\alpha}$.

Let's examine the number of elements that need to be checked for a hit or miss using double hashing:

α:	1/2	2/3	3/4	9/10
hit:	1.4	1.6	1.8	2.6
miss:	1.5	2.0	3.0	5.5

Notice that a miss on a load factor of $\frac{9}{10}$ was improved from 55.5 to 5.5 by using double hashing (a 10x increase in performance!).

10.2 Hashing Applications

Back in the day, Apple had a Pascal compiler that took an hour to compile programs. Another company made a compiler that could do the same job in just one and a half minutes. They achieved that by using hashing for all identifiers/tokens, while Apple used a straight-line table.

Network routers need performance greater than $O(\lg n)$ for passing packets to their appropriate destinations. A solution was found that ran in $O(\lg \lg n)$ time, which was nice. But even that wasn't fast enough. Now, hashing is used to achieve constant $O(1)$ time. Cisco continues to fund research to make hashing as efficient as possible.

A hash table may also be good for implementing a tree structure if the worst case complexity doesn't concern you.

10.3 Cuckoo Hashing

Cuckoo hashing is a different way of handling the insertion of a second element with the same hash key. The first element (that's already in the hash table) is the one that gets moved by calculating its h_2 displacement value. The latest element to be inserted will always be in the h_1-th position for that hash key.

More on this next time.