Comprehensive CS

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

Theory of Computation

1.1 Natural Languages, Formal languages: Definitions and theorems

- **Gödel's incompleteness theorem**: Gödel's Incompleteness Theorems are two fundamental results in mathematical logic that state:
  - Proved that for some axiomatic systems that there is no algorithm that will generate all true statements from those axioms.
  - No such system can prove its own consistency.

  This was the first indication that there are inherent limts on algorithms
- **Turing**: Alan Turing later provided formalism to the concepts of an "algorithm" and "computation", he Invented definition for an abstract machine called the "universal algorithm machine", he Provided means to formally (i.e., with mathematical rigor) explore the boundaries of what algorithms could, and could not, accomplish. Turing's model for a universal abstract machine was the basis for the first computer – in fact, Turing was involved in the construction of the first computer.
- **Natural languages**: We communicate via a *natural language*, Although we don't often think about it, our language is guided by rules; spelling, grammar, punctuation
- **Formal language**: Formal languages, which are not intended for human-to- human communication, are similar to natural languages in that they too have rules that define "correct" words and statements, but they are also different than natural languages in two key ways;
  - The rules that define a formal language are strictly enforced. There is no tolerance for misspellings, bad grammar, etc.
  - For the purpose of determining if a word or statement is acceptable in a formal language, meaning is ignored. Determining if something is (or is not) part of a language is determined by the language's defining rules which do not attach meaning (i.e., no definitions of words like in natural languages)

  In short, formal languages is a game of symbols, not meaning
- **Formal Languge terminology**:
  - **Symbol**: it is an abstract entity that is not formally defined – like a point or a line in geometry – but think of it as a single character like a letter, numerical digit, punctuation mark, or emoticon
  - **String (or Word)**: A finite sequence (i.e., order matters) of zero or more symbols
  - **Length**: The length of a string w is denoted by $length(w)$ or $|w|$ and is the number of symbols composing the string. Because strings, by definition, are finite then a string's lengths is always defined (sometimes zero).
  - **Prefix, suffix**: Any number of leading/trailing symbols of the string.
  - **Concatenation**: The concatenation of two strings w and x is formed by writing the first string w then the second string x

Note: For any string w, $\Lambda w = w\Lambda = w$

- **Alphabet**: A finite set of symbols, typically denoted by the Greek capital letter sigma Σ, for example

$$\Sigma = \{a, b, c\} \quad \Sigma = \{0, 1\} \quad \Sigma = \varnothing \quad \text{(special case)}.$$

- **The empty string**: A string with zero symbols is called the empty string and is denoted by the capital Greek letter lambda Λ, or sometimes lower case Greek letter epsilon ϵ, where Λ and ϵ are **not** symbols

  Thus,

$$|\Lambda| = 0.$$

- **Formal language definition**: A formal language is a et of strings from some **one** alphabet. Given an alphabet we generally define a formal language over that alphabet by specifying rules that either;

  1. Tell us how to test a candidate word, or
  2. Tell us how to construct all words.

  For example, Given $\Sigma_1 = \{x\}$, we can define languages

  $L_1 =$ any non empty string $= \{x, xx, xxx, ...\}$
  $L_2 = \{X^n : \ x = 2k + 1, \ k \in \mathbb{Z}\} = \{x, xxx, xxxxx, xxxxxxx, ...\}$ $L_3 \quad = \{x, xxxxxxxx\}$.

- **The empty language**: The empty language $L = \varnothing$ is typically denoted with the capital greek letter phi Φ. Thus, $L = \varnothing = \Phi$

- **Notes on formal languages**:

  - All languages are defined over some alphabet; cannot define a language without an alphabet.
  - Some languages are finite, some languages are infinite (remember, alphabets are always finite).
  - Some languages include the empty string Λ, some do not.
  - Some languages are defined by rules, some are simply written completely (e.g., $\Sigma_1 = \{x\}$, $L_3 = \{x, \text{xxxxxxxxxx}\}$).
  - No matter what the alphabet Σ (even $\Sigma = \emptyset$), you can always define at least two languages; $L_1 = \{\Lambda\}$ and $L_2 = \emptyset$.

- **Closure of an alphabet (closure of Σ) (Kleene closure)**: The language defined by the set of all strings (including the empty string Λ) over a fixed alphabet Σ.

  - **Examples:**

$$\begin{aligned} \Sigma &= \{a\} & \Sigma^* &= \{\Lambda, a, aa, aaa, aaaa, \dots\} \\ \Sigma &= \{0, 1\} & \Sigma^* &= \{\Lambda, 0, 1, 00, 01, 10, 11, 000, \dots\} \\ \Sigma &= \emptyset & \Sigma^* &= \{\Lambda\} \end{aligned}$$

    Note: If $\Sigma = \emptyset$ then Σ^* is finite and $\Sigma^* = \{\Lambda\}$, otherwise Σ^* is infinite.

- **Positive closure**: $\Sigma^+ = \Sigma^* - \{\Lambda\}$, you just take the empty string out of the kleene closure

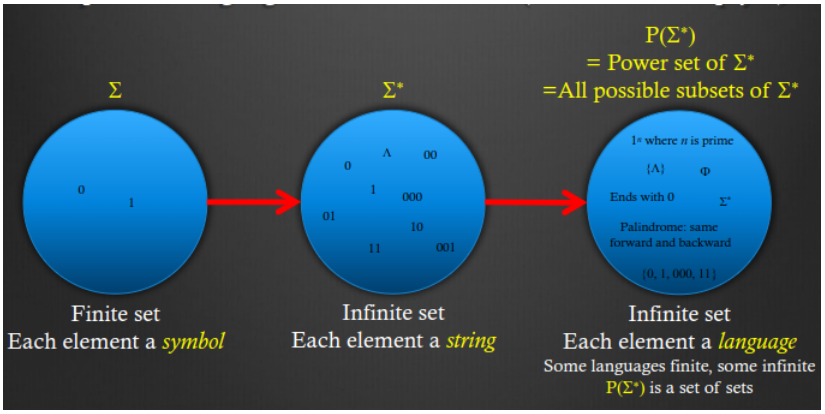
- **Recall: Power set**: The power set of any set S, written $\mathcal{P}(S)$ is the set of all subsets of S, including the empty set and the set S itself.

In other words, given a set S, then its power set $\mathcal{P}(S)$ is a set of sets

- **Note:**
  - If $S = \emptyset$, then then $\mathcal{P}(S) = \mathcal{P}(\emptyset) = \{\emptyset\} = \{\emptyset\}$ = a set with one element = $\emptyset$.
  - If S is non-empty and finite with n elements, then $\mathcal{P}(S)$ will be finite with 2^n elements.
  - If S is infinite, then $\mathcal{P}(S)$ will be infinite.
- **Example:**
  If $S = \{x, y, z\}$, then $\mathcal{P}(S)$ will have the following $2^3 = 8$ elements (each a set):

$$\mathcal{P}(S) = \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$$

- **Power set of the kleene closure** $\mathcal{P}(\Sigma^\star)$: Given some alphabet Σ we can construct the set of all possible languages from Σ as follows (assume non-empty Σ):



- **From formal languages to computers**:
  - Given an alphabet Σ we can define many formal languages – the range of which is captured by $\mathcal{P}(\Sigma^*)$.
  - We can define many formal languages verbally, but is there a way to define/express every language in any $\mathcal{P}(\Sigma^*)$ with some formal system or abstract machine?
  - We search for a formal system or abstract machine with enough "power" to define any language in any $\mathcal{P}(\Sigma^*)$.
  - **KEY POINT**
    The abstract machines we discover along our search to cover $\mathcal{P}(\Sigma^*)$ turn out to be *the theoretical basis for all computing.*
  - In other words, by understanding the power (and limitations) of abstract machines that cover $\mathcal{P}(\Sigma^*)$, we are simultaneously discovering the same limits about all computing.

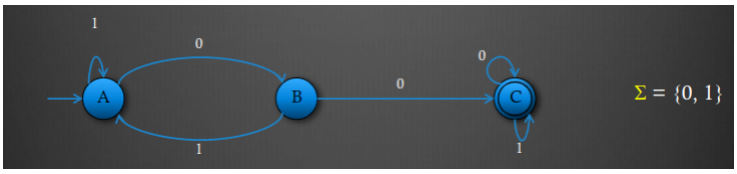
1.2 Regular languages

Preface. The first few subsubsections will be in the world of regular languages. In the context of computation theory, regular languages are a class of formal languages that can be recognized by finite automata. These languages are important because they are the simplest class of languages that can be described by a computational model. The characteristics of regular languages are as follows,

- **Finite Automata:** Regular languages can be recognized by deterministic or nondeterministic finite automata (DFA or NFA).
- **Regular Expressions:** Regular languages can be described using regular expressions.
- **Closure Properties:** Regular languages are closed under several operations, including:
  - **Union:** The union of two regular languages is also regular.
  - **Concatenation:** The concatenation of two regular languages is also regular.
  - **Kleene Star:** The Kleene star operation, which involves repeating a regular language any number of times (including zero), results in a regular language.
  - **Intersection and Difference:** Regular languages are also closed under intersection and difference.
- **Decision Problems:** Certain decision problems are decidable for regular languages. For example, it is possible to determine whether a given string belongs to a regular language (membership problem), whether two regular languages are equivalent, or whether a regular language is empty.

1.2.1 Finite Automata

- **Informal definition**: Described informally, a finite automaton (FA) is always associated with some alphabet Σ and is an abstract machine which has
  1. A non-empty finite number of states, exactly one of which is designated as the "start state" and some number (possibly zero) of which are designated as "accepting states".
  2. A transition table that shows how to move from one state to another based on symbols in the alphabet Σ
- **A simple example of a FA**:



  - Defined over alphabet $\Sigma = \{0, 1\}$.
  - States are circles; transitions are directed edges (i.e., arrows) between states.
  - Has exactly three states; **A**, **B**, and **C**.
  - Every FA must have exactly one start state. In this example, the start state is **A** and denoted as the only state that has an edge coming to it from no other state.

- There is only one accepting state, **C**, and it is denoted by its *double circle*. (We could have more than one but in this case we only have one)
- **Very important:**
  - Each symbol in the alphabet has exactly one associated edge leaving every state.
  - In other words, every state must have exactly one edge leaving it for each symbol in the alphabet.

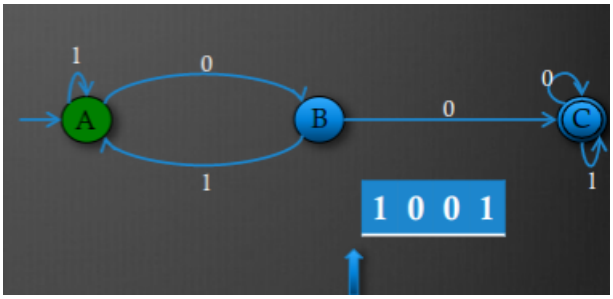
- **How to use an FA**: The purpose of a FA is to define a language over its alphabet Σ. The FA provides the means by which to test a candidate string from Σ and determine whether or not the string is in the language. It does this by "writing" the candidate string on an fictitious input tape and proceeding as follows:

  1. Set the FA to the start state.
  2. If end-of-string then halt.
  3. Read next symbol on tape.
  4. Update the state according to the current state and the last symbol read.
  5. Goto step 2.

  When the process halts check which state the FA is in. If it is in any accepting state, then the string is in the language defined by the FA, otherwise the string is not in the language

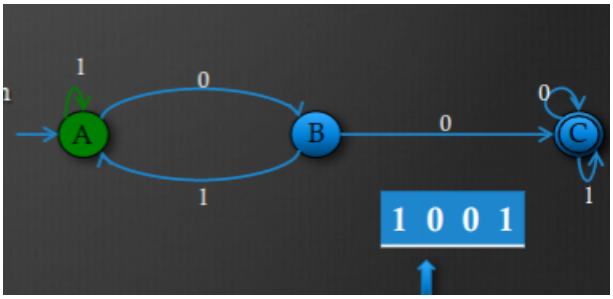
- **Using the previous FA**: Let's now try to use our FA to test whether or not the string 1001 is in the language

  We start by writing the string on an input tape, placing the read head at the beginning of the tape, and placing the FA in its initial state, *A*



  Since the tape head is not at the end of the tape we

  1. Read the next symbol from the tape.
  2. Follow the edge from the state we are currently in that corresponds to the symbol we just read to transition to the next state.
  3. Move the tape head

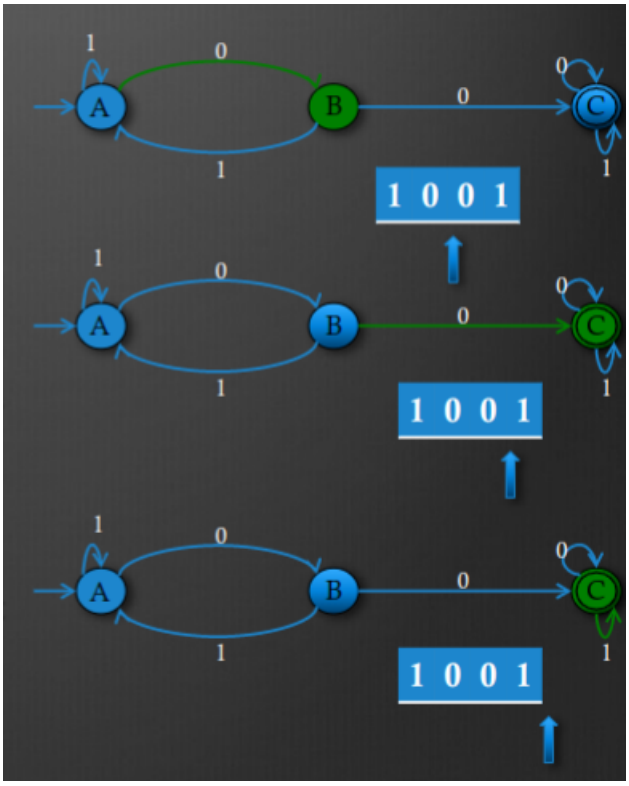


In this case, we started in state A, read symbol 1, and followed the edge labeled 1 from A which brought us back to A

We proceed in this way, read, change state, move tape head until we reach the end of the tape

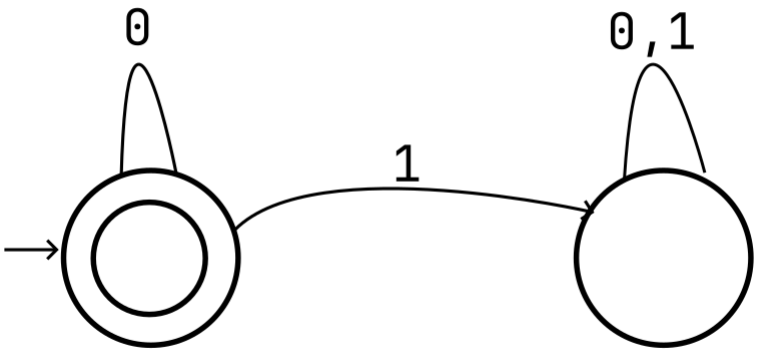
Once the tape head reaches the end of the tape we simply look to see whether or not the FA ended in an accepting state.

In this case it ended in state C, which is an accepting state, which means that string 1001 is in the language.



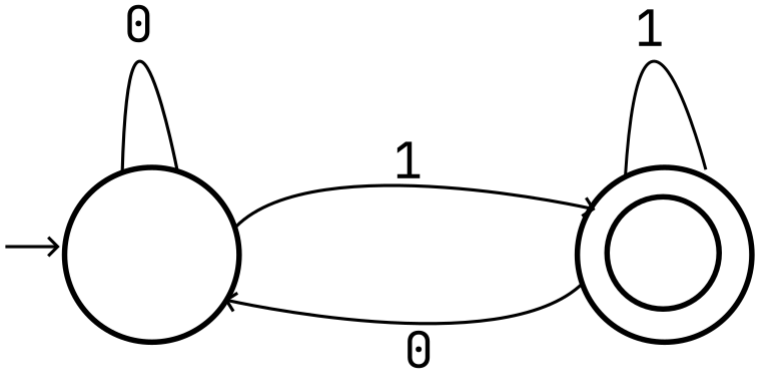
We deduce that the language has only strings with two consecutive zeroes somewhere.

- **FA Example Two: The set of all strings that do not contain a one** ($\Sigma = \{0, 1\}$):

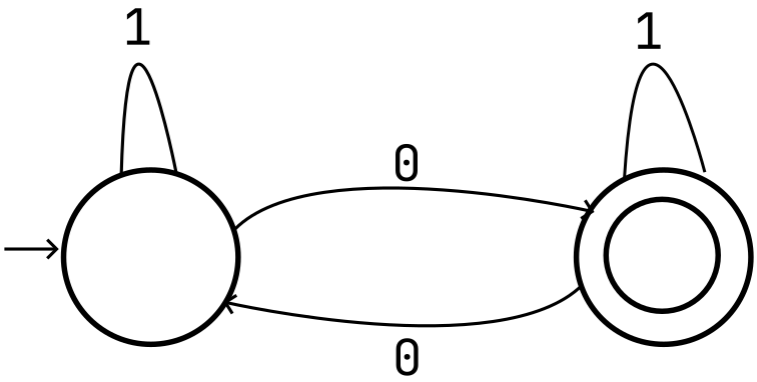


This one is pretty simple. If we have a zero, stay in the accepting state, if we see a one, toss it to the other non-accepting state, its not coming back.

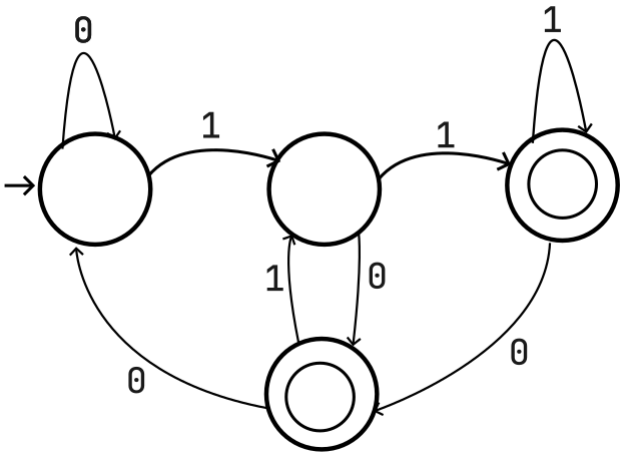
- **FA Example Three: The set of all strings that end in one** ($\Sigma = \{0, 1\}$):



- **FA Example Four: The set of all strings with an odd number of zeros** ($\Sigma = \{0, 1\}$):



- **FA Example Five: The set of all strings where the second to last symbol is one** ($\Sigma = \{0, 1\}$):



- **States are "memory"**: Consider the four FA we just created, in each instance the solution required us to design an FA that remembered at least part of what it had already read from the input tape. The type of memory that an FA has is very different than the RAM we find in contemporary computers, but the FA does have memory. Each time the FA enters a different state it is, in effect, redefining the memory of the entire FA. The FA can only be in a finite number of states, and that number can be arbitrarily large, but (as we will see) that difference in memory has a profound limiting effect in what FAs can compute.

- **Limits of a FA**:

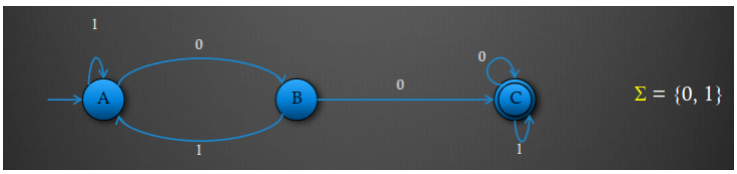
  Limited Memory:

- **Finite State:** A finite automaton has a finite number of states. This means it can only "remember" a limited amount of information about the input it has processed. Once a finite automaton transitions to a new state, it forgets all previous information except for the current state.
- **No Stack or Tape:** Unlike more powerful models such as pushdown automata (which have a stack) or Turing machines (which have an infinite tape), finite automata cannot use any form of auxiliary memory to keep track of an unbounded number of items or to perform operations that require more complex memory management.

Inability to Count Unboundedly:

- **No Arbitrary Counting**: Finite automata cannot count occurrences of symbols beyond the number of states they have. For example, a DFA with n states can only count up to $n-1$ occurrences of a symbol reliably. Thus, they cannot recognize languages that require matching counts of different symbols if those counts are unbounded, such as $\{a^n b^n \mid n \geqslant 1\}$, where the number of 'a's must match the number of 'b's.

- **FA Formal Definition**: We formally denote a *finite automaton* by a 5-tuple $(Q, \Sigma, q_0, T, \delta)$, where
  - Q is a finite set of *states*.
  - Σ is an alphabet of *input symbols*.
  - $q_0 \in Q$, is the *start state*.
  - $T \subseteq Q$, is the set of *accepting states*.
  - δ is the *transition function* that maps a state in Q and a symbol in Σ to some state in Q. In mathematical notation, we say that $\delta : Q \times \Sigma \to Q$. With:
    - $Q \times \Sigma$: The Cartesian product of the set of states Q and the alphabet Σ. This represents all possible pairs of a state and an input symbol.
    - $\to Q$: Indicates that the transition function maps each pair (q, σ) (where $q \in Q$ and $\sigma \in \Sigma$) to a single state in Q.
- **Formally Specifying Our First FA**:



Recall our first FA that accepts any string with two consecutive zeros somewhere.

We drew it as a Finite State diagram, but to formally define this FA we must specify each of the elements from the 5-tuple $(Q, \Sigma, q_0, T, \delta)$.

- Q is a finite set of *states*: $Q = \{A, B, C\}$
- Σ is an alphabet of *input symbols*: $\Sigma = \{0, 1\}$
- $q_0 \in Q$, is the *start state*: $q_0 = A$
- $T \subseteq Q$, is the set of *accepting states*: $T = \{C\}$

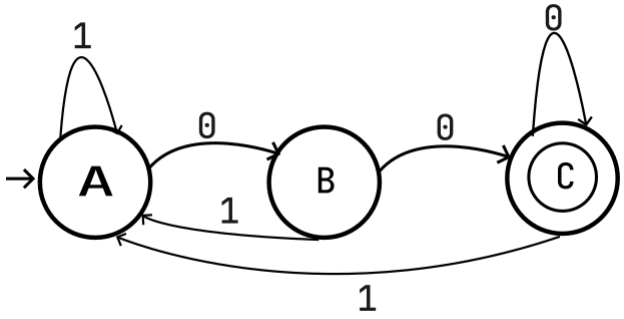
– δ is the *transition function* $\delta : Q \times \Sigma \to Q$

δ	0	1
A	B	C
B	C	A
C	C	C

- **Unary**: consisting of or involving a single component or element.
- **Unary language**: One where the alphabet has only one symbol.
- **Binary**: Relating to, composed of, or involving two things.
- **Ternary**: Composed of three parts.
- **Dead state (trap state)**: This is a state that once entered, can never be left.
- **Deterministic finite automaton (DFA)**: The FA's we have looked at thus far have been DFA's. A DFA is a finite automaton where, for each state and each input symbol, there is exactly one transition to a new state. This means that given a current state and an input symbol, the next state is uniquely determined. In the future we will look at nondeterministic finite automaton (NFA). An NFA is a finite automaton where, for each state and input symbol, there can be multiple possible transitions to different states. Additionally, an NFA can have transitions that do not consume any input symbol (ϵ-transitions).

1.2.2 Finite Automata: More examples

- $\Sigma = \{0, 1\}$**, all strings that start with 00**
- $\Sigma = \{0, 1\}$**, all strings that end with 00**



With:

- $Q = \{A, B, C\}$
- $\Sigma = \{0, 1\}$
- $q_0 = A$
- $T = C$
- $\delta :\ Q \times \Sigma \to Q$ defined by

δ	0	1
A	B	A
B	C	A
C	C	A

1.2.3 Regular expressions

- **RE**: A RE corresponds to a set of strings; that is, a RE describes a language
- **RE three operations**:
  1. Union (+)
  2. concatenation (xy)
  3. star (zero or more copies)
- **RE special symbols**

$$+ \quad * \quad (\quad).$$

- **Grouping**: The parenthesis are used for grouping,
- **Union**: the plus sign means **union**. Thus, writing

$$0 + 1.$$

  Means zero or one, we refer to + as "or"
- **Concatenation**: We concatenate simply by writing one expression after the other, with no spaces

$$(0 + 1)0.$$

  Is the pair of strings 00 and 10
- **Empty string**: We can also use the empty string ϵ

$$(0 + 1)(0 + \epsilon).$$

  corresponds to 00, 0, 10, and 1
- **Zero or more copies (star)**: Using the start indicates zero or more copies, thus

$$a*.$$

  corresponds to any string of a's: $\{\epsilon, a, aa, aaa, ...\}$
- **More on union**: If you form an RE by the or of two REs, call them R and S, then the resulting language is the union of the languages of R and S.

  Suppose $R = (0 + 1) = \{0, 1\}$, and $S = \{01(0 + 1)\} = \{010, 011\}$, then $R + S = (0 + 1) + (01(0 + 1)) = \{0, 1, 010, 011\}$
- **More on concatenation**: If you form an RE by the or of two REs, call them R and S, then the resulting language consists of all strings that can be formed by taking one string from the language of R and one string from the language of S and concatenating them.

  Suppose $R = (0 + 1) = \{0, 1\}$, and $S = \{01(0 + 1)\} = \{010, 011\}$, then $RS = (0 + 1)01(0 + 1) = \{0010, 0011, 1010, 1011\}$
- **More on star**: If you for man RE by taking the star of an RE R, then the resulting language consists of all strings that can be formed by taking any number of strings from the language of R (they need not be the same and they need not be different), and concatenating them.

Suppose $R = 01(0+1) = \{010, 011\}$, then $R^* = 01(0+1)*\{010, 010010, ..., 011, 011011, ...010011, ...\}$

- **Precedence of the operations**
  1. Star (*)
  2. Concatenation
  3. Union (+)
- **Recursive definition of the kleene star (closure) (L^*)**:
  1. $\epsilon \in L^*$
  2. If $x \in L^*$ and $y \in L$, then $xy \in L^*$

  Base case: The first rule provides a starting point by ensuring that the empty string ϵ is in L^*.

  Recursive step: The second rule allows you to take any string x already in L^* and concatenate it with a string $y \in L$ to produce a new string $xy \in L^*$.

  After using the second rule once to generate a new string $xy \in L^*$, you can apply the rule again by concatenating this new string with another string from L. This recursive process can continue indefinitely, generating all possible strings that can be formed by concatenating zero or more strings from L.
- **Recursive definition of the kleene star (other)**
  1. $L^0 = \{\epsilon\}$ (Start with the empty string, always in the closure)
  2. $L^i = LL^{i-1}$ for $i > 0$ (Start recursively building strings)
  3. $L^* = \bigcup_{i=0}^{\infty} L^i$ (the whole thing)

  Note: We also define the positive closure of L, denoted L^+, as $L^* - \{\epsilon\}$ or

$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

- **Closure of the empty language**: $\Phi^* = \{\epsilon\}$
- **Regular expression for the empty language**: $\Phi = \varnothing$ is the regular expression for the empty language (empty set)
- **More on language composition operators**: The language composition operators were defined over any language and, in turn, generate new languages. As such, composition operators take any one or two languages from $P(\Sigma^*)$ and can produce any language in $P(\Sigma^*)$.

$$\in \quad \epsilon.$$

- **Regular languages (regular sets), regular expression limits**: Although regular expressions are based on language composition operators, their recursive definition (i.e., only regular expressions, therefore only languages defined by regular expressions) limits the languages that they can define.

  Note: Regular expressions cannot produce all languages in $P(\Sigma^*)$.

In fact, the set of languages that regular expressions can define have a special name – they are called regular languages (or sometimes regular sets).

- **Kleene's theorem**: There is an FA for a language if and only if there is an RE for the language
- **Regular expressions order of operations**: From highest to lowest precedence
    1. Parenthesis
    2. Kleene star
    3. Concatenation
4. Union (+ or ||)

- **Properties of regular expressions**:

  Note: Intersection is a operation not defined for regular expressions

  Union

  – **Commutative**:
  $$R_1 \cup R_2 = R_2 \cup R_1$$
  – **Associative**:
  $$(R_1 \cup R_2) \cup R_3 = R_1 \cup (R_2 \cup R_3)$$
  – **Identity Element**:
  $$R_1 \cup \emptyset = R_1$$
  – **Idempotent**:
  $$R_1 \cup R_1 = R_1$$

  2. Concatenation ($\cdot$)

  – **Non-commutative**:
  $$R_1 \cdot R_2 \neq R_2 \cdot R_1$$
  – **Associative**:
  $$(R_1 \cdot R_2) \cdot R_3 = R_1 \cdot (R_2 \cdot R_3)$$
  – **Identity Element**:
  $$R_1 \cdot \epsilon = \epsilon \cdot R_1 = R_1$$
  – **Concatenation with $\emptyset$**:
  $$R_1 \cdot \emptyset = \emptyset \cdot R_1 = \emptyset$$

  Kleene Star ($*$)

  – **Kleene Star of ϵ**:
  $$\epsilon^* = \{\epsilon\}$$
  – **Kleene Star of $\emptyset$**:
  $$\emptyset^* = \{\epsilon\}$$
  – **Idempotent**:
  $$(R^*)^* = R^*$$

Distributive Properties

- **Union over Concatenation**:

$$R_1 \cdot (R_2 \cup R_3) = (R_1 \cdot R_2) \cup (R_1 \cdot R_3)$$

- **Concatenation over Union**:

$$(R_1 \cup R_2) \cdot R_3 = (R_1 \cdot R_3) \cup (R_2 \cdot R_3)$$

- **Language of a RE notation**: $L(RE)$ is the language defined by the regular expression RE, If we have an RE R, then the language $L(R)$ is the language defined by the RE R
- **When a regular expression is the empty set** $\varnothing$: When a regular expression (RE) represents the empty set it means that the RE matches no strings at all, not even the empty string.

  The language is then

$$L(\varnothing) = \Phi.$$

  Where Φ denotes the empty language
- **One or more occurences** RR^*: We denote this by plus instead of star, ie $RR^* = R^+$, but you also must redefine union as $|$ instead of $+$
- **Simplifying regular expressions (Some can also be found above in properties)**:
  - **Concatenation of stars**: $(R^*)^* = R^*$
  - **Concatenation of Repeated Expressions**: $R^*R^* = R^*$
  - **Idempotence of Union**: $R \mid R = R$
  - **Empty Set in Union and Concatenation:** $R \mid \varnothing = R$, $R\varnothing = \varnothing$
  - **Empty string in concatenation**: $\epsilon R = R\epsilon = R$
  - **Union with the kleene star**: $R^* \mid R = R^*$
  - **Distributive Property:** $R_1(R_2 \mid R_3) = R_1R_2 \mid R_1R_3$
  - **Absorption:** $R \mid (RR^*) = RR^* = R^+$
- **The RE operators with the empty language** Φ:
  1. $\varnothing r = r\varnothing = \varnothing\varnothing = \varnothing$ for any regular expression r
  2. $r + \varnothing = \varnothing + r = r$
  3. $\varnothing + \varnothing = \varnothing$
  4. $\varnothing^* = \{\epsilon\}$

  These cases can also be represented with language notation

  1. $\Phi L = L\Phi = \Phi\Phi = \Phi \ \forall L$
  2. $L + \Phi = \Phi + L = L$
  3. $\Phi + \Phi = \Phi$
  4. $\Phi^* = \{\epsilon\}$
- **Convert RE to NFA-**ϵ: The conversion algorithm starts by defining an NFA with ϵ-moves for each of the three base cases from the recursive definition of a regular expressions over an alphabet Σ
  1. $\varnothing$ is a regular expression and denotes the empty set (i.e., the empty language Φ)
  2. ϵ is a regular expression and denotes the set $\{\epsilon\}$
  3. For each symbol $x \in \Sigma$, x is a regular expression and denotes the set $\{x\}$.

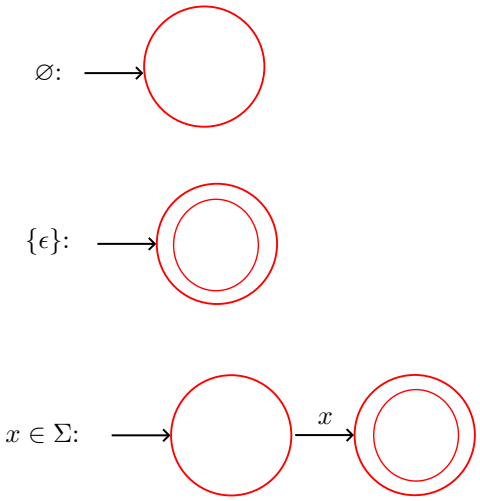
Conditions on the NFAs with ϵ-moves for This Algorithm

1. There must be exactly one accepting state.
2. No transitions (not even ϵ-moves) may leave the one accepting state.

Note: If faced with an NFA with ϵ-moves that has more than one accepting state and/or accepting states with transitions leaving it then simply modify the NFA with ϵ-moves by

1. Adding a new accepting state.
2. Add an ϵ-move from each of the original accepting states to the newly added accepting state.
3. Convert all of the original accepting states to non-accepting states.

The three base cases have the following nfa that satisfy the above criteria



We use those NFAs as the basic building blocks to iteratively build more complex NFA's with ϵ-moves (all the while honoring the accepting state conditions for this algorithm) as we apply the recursive part of the regular expression definition. Recall:

If r and s are regular expressions denoting the sets R and S, respectively, then

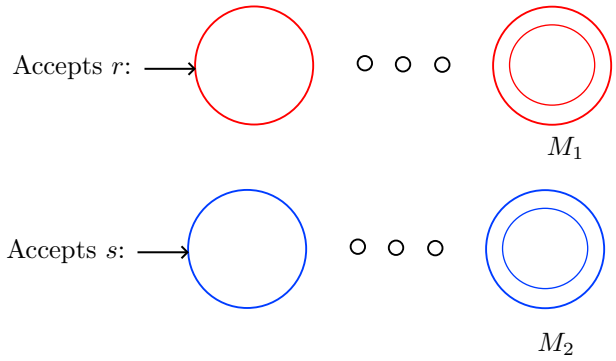
1. $r + s$ is a regular expression denoting the set $R + S$, (i.e,. union of languages),
2. rs is a regular expression denoting the set RS (i.e., concatenating languages), and
3. $r*$ is a regular expression denoting the set R^* (i.e., Kleene closure of a language).

Once we define an NFA with ϵ-moves for each of the base cases (which we have done) then when we address each recursive part of the definition (e.g., union above)

1. We may assume that there already exists NFAs with ϵ-moves for each of the regular expressions r and s (and that each also satisfies the acceptance state conditions of this algorithm) and
2. Then our job is to use those NFAs with ϵ-moves to create a new NFA with ϵ-moves that accepts $r + s$ and that also satisfies the acceptance state conditions of this algorithm.

The algorithm:

- **Handling union:** We start by assuming there already exists NFAs with ϵ-moves M_1 and M_2 that accept regular expressions r and s, respectively, and that both M_1 and M_2 satisfy the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.



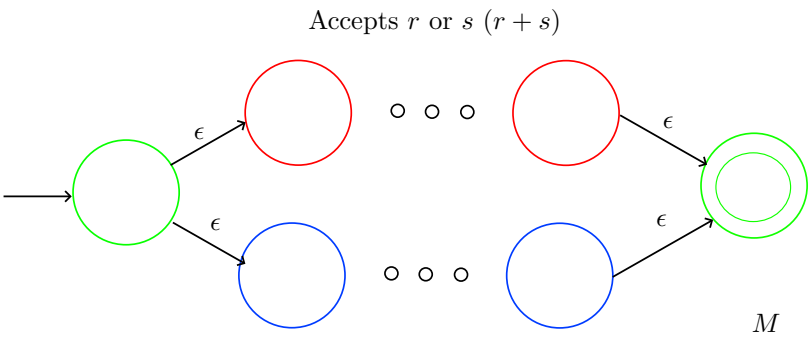
Note: The details of the machine arn't important here, all we know is the machine has a start, does whatever else it needs to (repesented by the elipsis), and then accepts strings represented by r in the top machine and s in the bottom

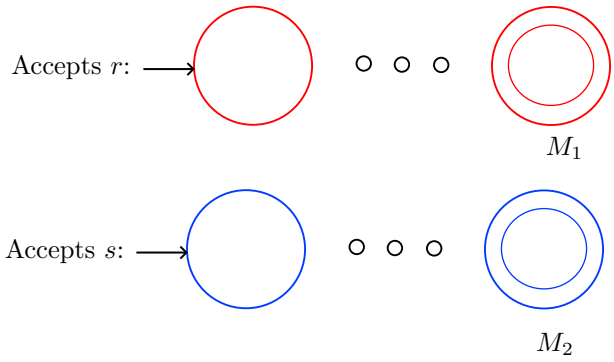
We then use these machines M_1 and M_2 to create a new machine M that accepts $r + s$

So what did we do here

1. Create new start state and add ϵ-moves to the original start states
2. Create new accepting state and add emoves from all the original accepting states.
3. Change the original accepting states to non-accepting states.

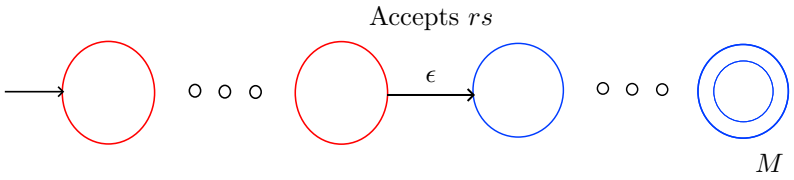
- **Handle Concatenation**: We again start by assuming there already exists NFAs with ϵ-moves M_1 and M_2 that accept regular expressions r and s, respectively, and that both M_1 and M_2 satisfy the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.



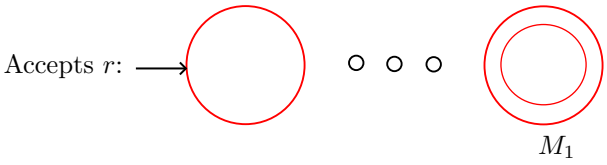


We use M_1 and M_2 to construct new NFA with ϵ-move M that accepts rs.

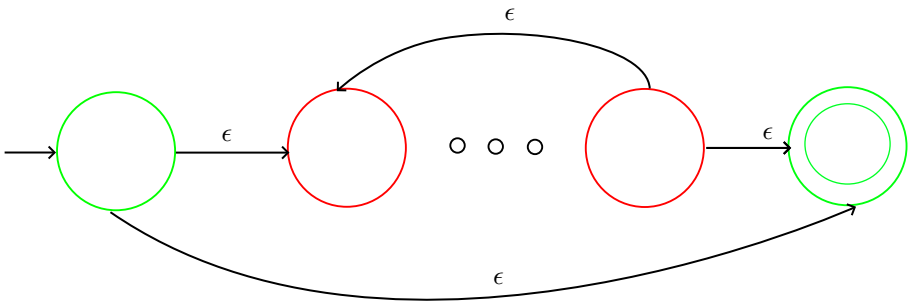
1. Add an ϵ-move from M_1's accepting state to M_2's start state.
2. Change M_1's accepting state to a nonaccepting state.



– **Handle Kleene closure**: We again start by assuming there already exists an NFA with ϵ-moves M_1 that accepts regular expressions r and that satisfies the acceptance state conditions (i.e., one accepting state, no exit) of this algorithm.



We use M_1 to construct new NFA with ϵ-move M that accepts r^*

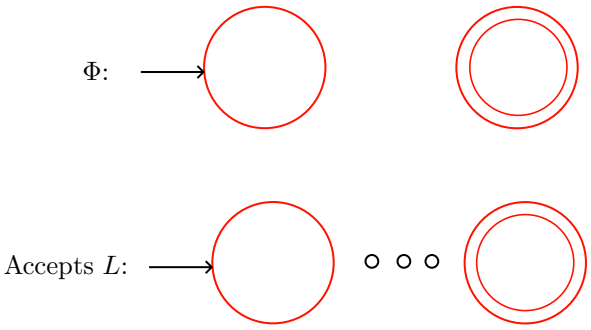


1. Create new start and accepting states.
2. Add ϵ-move from new start to M_1 start, M_1 accepting to new accepting, and new start to new accepting.
3. Add ϵ-move from M_1 accepting to M_1 start.
4. Change M_1's accepting state to a non accepting state.

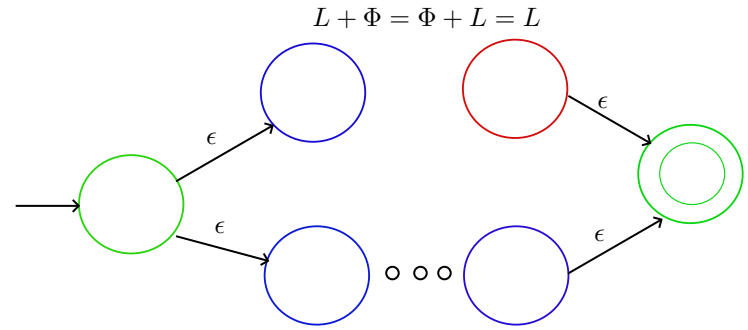
- **RE to NFA conversion: Special cases**: Recall the special case with regular expressions, the empty language Φ – and how it behaved with the three regular expression operators;

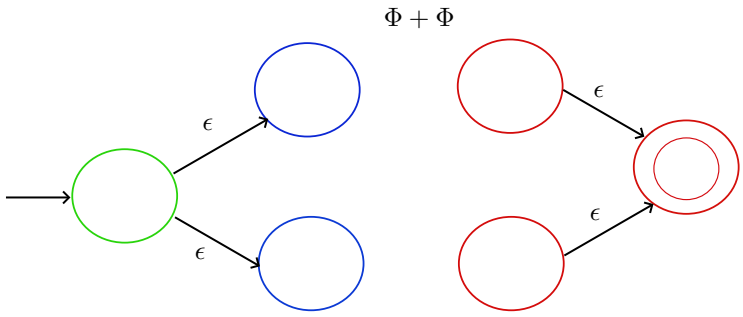
1. $\Phi L = L\Phi = \Phi\Phi = \Phi \ \forall L$
2. $L + \Phi = \Phi + L = L$
3. $\Phi + \Phi = \Phi$
4. $\Phi^* = \{\epsilon\}$

We can now check these operations using the algorithm with Φ. First, we define the base case NFA's

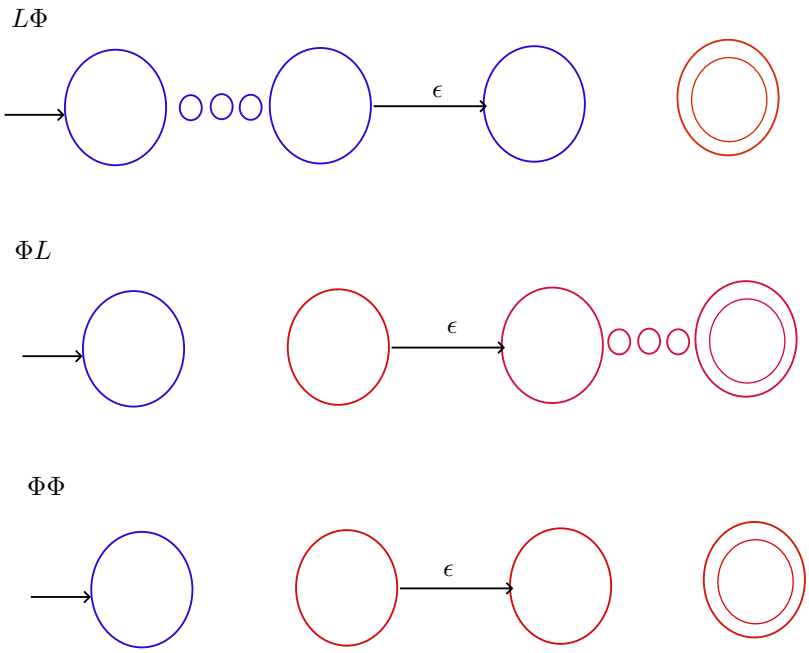


1. Confirming $L + \Phi = \Phi + L = L$ and $\Phi + \Phi = \Phi$:

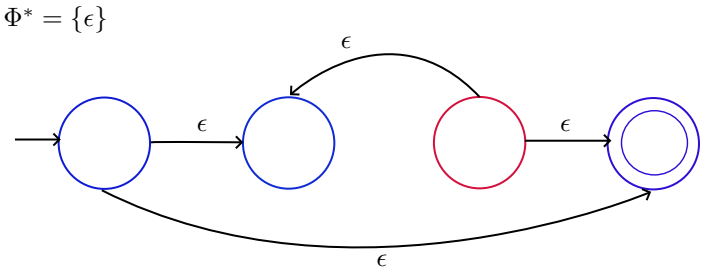




2. Confirming $\Phi L = L\Phi = \Phi\Phi = \Phi$



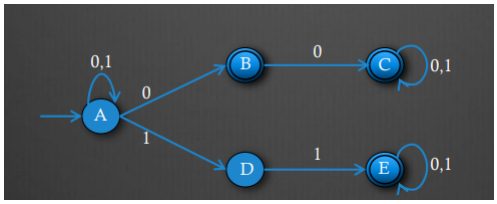
3. Confirming $\Phi^* = \{\epsilon\}$



1.2.4 nondeterministic Finite automata (NFA)

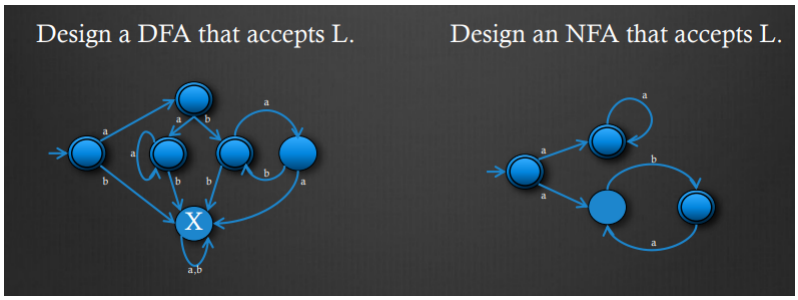
- **NFA definition**:
  - If an automaton gets to a state where there is more than one possible transition corresponding to the symbol read from the tape, the automaton may choose any of those paths. (nondeterminism) We say it **branches**
  - if an automaton gets to a state where there is no transition for the symbol read from the tape, then that path of the automaton halts and rejects the string. We say it **dies**
  - the automaton accepts the input string if and only if there exists a choice of transitions that ends in an accept state.

  Example: Consider this nondeterministic FA (NFA) over $\Sigma = \{0, 1\}$

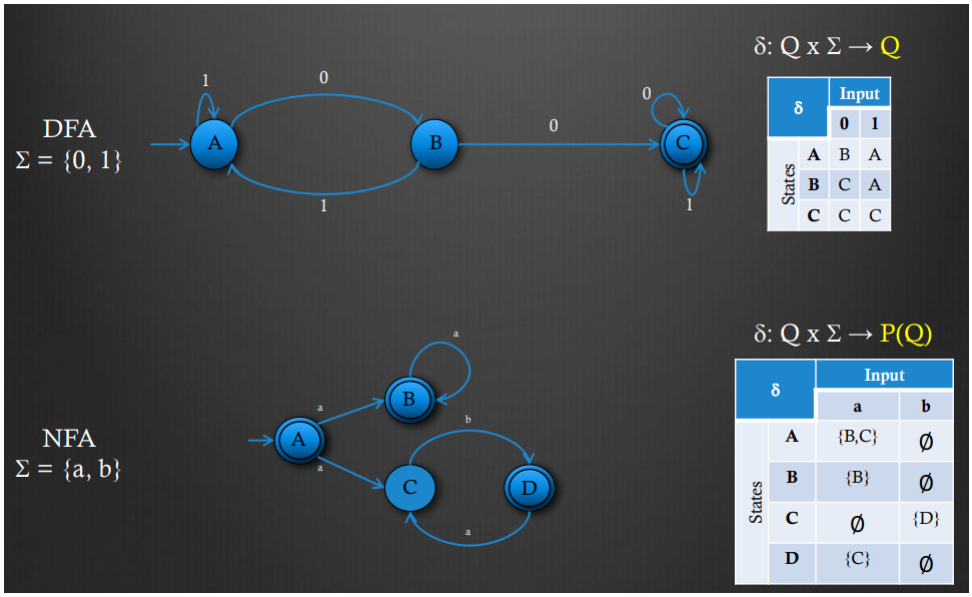


- **DFA or NFA?**: Consider the language L over $\Sigma = \{a, b\}$ which is defined by

$$L = (a^*) + (ab)^*.$$



- **NFA Formal definition**: We define an NFA $M(Q, \Sigma, q_0, T, \delta)$
  - Q is a finite set of states
  - Σ is an alphabet of input symbols
  - $q_0 \in Q$ is the start state
  - $T \subseteq Q$ is the set of accepting states
  - δ is the transition function $\delta : Q \times \Sigma \to P(Q)$
- **Transition function, DFA vs NFA**:

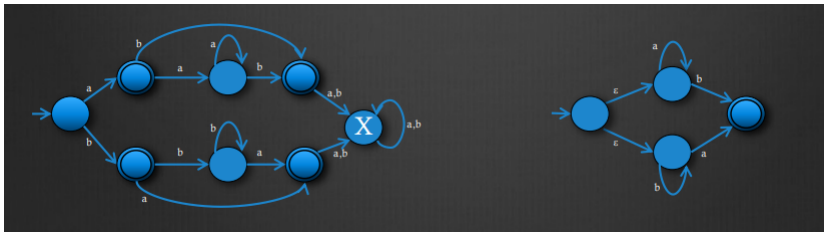


- **NFA with ϵ-transitions**: ϵ-transitions allow the automaton to change state without consuming an input symbol

  Changing states without consuming input symbols can go on arbitrarily long as there are ϵ-transitions to traverse.

- **DFA or NFA with ϵ-moves?**: Consider the language L over $\Sigma = \{a, b\}$ which is

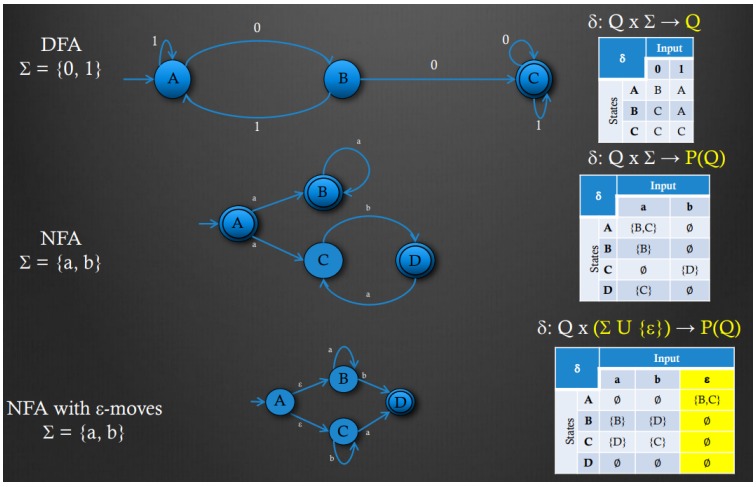
$$L = (b^*a) + (a^*b).$$



- **NFA with ϵ-transitions formal definition**: Everything is the same except for the transition function, we now have

$$\delta : \ Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q).$$

- δ – **DFA, NFA, and NFA with ϵ-moves**:



- **DFA, NFA, or NFA with ϵ moves, who can define the most languages?**: We begin by noting, by definition, every DFA is an NFA. This means that any language you can define with a DFA can also be defined by an NFA. Thus,

$$\text{Languages defined by DFA} \subseteq \text{ Languages defined by NFA.}$$

  Also, by definition, every DFA is an NFA with ϵ-moves, an NFA is an NFA with ϵ moves, even if it doesnt have any. Thus,

$$\text{Languages defined by DFA} \subseteq \text{ Languages defined by NFA with } \epsilon\text{-moves.}$$

  But, by definition, every NFA is an NFA with ϵ-moves. Thus,

$$\text{Languages defined by NFA} \subseteq \text{ Languages defined by NFA with } \epsilon\text{-moves.}$$

  This tells us that

  - NFAs are at least as powerful in defining languages as DFAs
  - NFAs with ϵ-moves are at least as powerful in defining languages as DFAs and NFAs.

  It turns out that these three are **equally** as powerful. We assert

$$\begin{aligned}&\text{Languages defined by DFA's}\\ &= \text{Languages defined by NFA's}\\ &= \text{Languages defined by NFA's with } \epsilon\text{-moves.}\end{aligned}$$

  We prove this by showing an algorithm that converts any NFA with ϵ-moves (or any NFA) to a DFA that accepts the exact same language

  This means that there does not exist a language that can be defined by an NFA with ϵ-moves (or NFA) that cannot also be defined by a DFA.

- **ϵ-closure**: Before we can look at the algorithm we must first define the ϵ-closure of a set of states

Given:

- an NFA with ϵ-moves $M(Q, \Sigma, q_0, T, \delta)$
- Some set of states $S \subseteq Q$

We define the ε-closure(S) as the set of states that are reachable from the set of states S using only zero or more ε-moves in δ.

Note: it is always the case that S $\subseteq \varepsilon$-closure(S)

The formal definition is

$$\epsilon - \text{closure}(q) = \{q\} \cup \{p : \ q \xrightarrow{\epsilon} p\}.$$

- **ϵ-closure alternate notation**.

$$\epsilon\text{-closure}(\{A\}) = \epsilon(\{A\}) = E(\{A\}).$$

- **ϵ-closure of the empty set $\varnothing$**: The epsilon closure of the empty set is $\epsilon(\varnothing) = \varnothing$
- **Algorithm: Converting NFA with ϵ-moves to DFA**: The algorithm constructs a new DFA $M'(Q', \Sigma, q_0', T', \delta')$ From an NFA with ϵ-moves $M(Q, \Sigma, q_0, T, \delta)$. Σ will remain the same

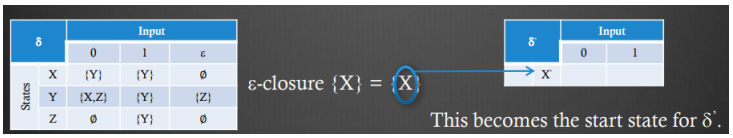
Things to note about the conversion:

- Same alphabet Σ
- Lose column ϵ
- Lose all nondeterminism
- Lose all empty sets
- Cell values change from sets of states to states

Example: **Consider the following NFA with ε-moves M(Q, Σ, q_0, T, δ) over $\Sigma = \{0, 1\}$ and its associated transition table δ: Q $\times$ ($\Sigma \cup \{\varepsilon\}$) $\rightarrow$ P(Q)**

	0	1	ε
X	$\{Y\}$	$\{Y\}$	$\emptyset$
Y	$\{X, Z\}$	$\{Z\}$	$\{Z\}$
Z	$\emptyset$	$\{Y\}$	$\emptyset$

Start by computing the ϵ-closure of the start state in δ.



There is a subtle - but very important - point to be made here ...

we cannot simply take the ϵ-closure (a set) and use it to create a row in δ' (which needs to be a state). What we do is create a label for the new state in δ' that represents the set of states from δ and then add that new state to δ'

In this instance we represented the set of states $\{X\}$ by a single state whose label is X'

We continue by filling the columns of the start state for each symbol $\Sigma = \{0, 1\}$

Processing δ' state X' which represents the set of states $\{X\}$ in M:

- Processing input symbol 0 (process each state in $\{X\}$ using δ):
  - Process X

$$\delta(X, 0) = \{Y\}$$

$$\varepsilon\text{-closure}(\{Y\}) = \{Y, Z\}$$

Since there are no more states in $\{X\}$ to process, we have finished processing the symbol 0 and have produced the set of states $\{Y, Z\}$.

We create a new state with label $Y'Z'$ (or $Z'Y'$, order does not matter) for δ' that represents $\{Y, Z\}$ in M and define:

$$\delta'(X', 0) = Y'Z'$$

We note that $Y'Z'$ is a new state in δ' and so we create a new row for it in δ'.

We continue this until we reach

δ'	Input	
	0	1
X'	Y'Z'	Y'Z'
Y'Z'		

Processing δ' state $Y'Z'$ which represents the set of states $\{Y, Z\}$ in M:

- Processing 0:
  - Process Y

$$\delta(Y, 0) = \{X, Z\}, \quad \varepsilon\text{-closure}(\{X, Z\}) = \{X, Z\}$$

  - Process Z

$$\delta(Z, 0) = \emptyset, \quad \varepsilon\text{-closure}(\emptyset) = \emptyset$$

Here is our first instance of processing a state and symbol where the state in δ' represents multiple states in NFA M. When this happens, the set of states in NFA M is computed by *taking the union of the ε-closures*: $\{X, Z\} \cup \emptyset = \{X, Z\}$.

This produces a new label $X'Z'$ which we use to define:

$$\delta'(X'Y', 0) = X'Z'$$

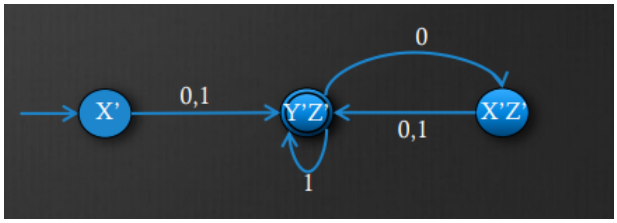
and since $X'Z'$ is a new state, we add it to δ'.

We continue this until we reach

δ'	Input	
	0	1
X'	Y'Z'	Y'Z'
Y'Z'	X'Z'	Y'Z'
X'Z'	Y'Z'	Y'Z'

A state in M' is an accepting state iff at least one of the states that it represents in M is an accepting state ... in this case $T' = \{Y'Z'\}$.

We can now draw the new DFA



Note: If the closure or union of closures is the empty set, we do this

δ'	Input	
	0	1
A'	B'C'	*empty*
B'C'		
empty	*empty*	*empty*

This "emtpy" is a state and represents a garbage state, what goes does not leave.

- **Kleene's theorem revisited**: The following are equivalent for a language L
  1. There is a DFA for L
  2. There is an NFA for L
  3. There is an RE for L

- **Union of two DFA's (cartesian product construction)**: The process of finding the union of two deterministic finite automata (DFAs) involves creating a new DFA that accepts the union of the languages accepted by the original DFAs. This is done using a product construction (also called the Cartesian product construction), where you combine the states of both DFAs in a systematic way to ensure the resulting DFA accepts strings from either of the original DFAs.

  Let's say we have two DFAs:

  $$D_1 = (Q_1, \Sigma, \delta_1, q_1^{\text{start}}, F_1)$$

  that recognizes language L_1.

  $$D_2 = (Q_2, \Sigma, \delta_2, q_2^{\text{start}}, F_2)$$

  that recognizes language L_2.

  Create a New DFA State Set:

  - The states of the new DFA are pairs of states, one from each of the original DFAs. The new state set will be the Cartesian product $Q_1 \times Q_2$, meaning every possible combination of a state from D_1 and a state from D_2.
  - If D_1 has n states and D_2 has m states, the new DFA will have $n \times m$ states.

  Define the New Start State:

  - The new start state is $(q_1^{\text{start}}, q_2^{\text{start}})$, where q_1^{start} is the start state of D_1 and q_2^{start} is the start state of D_2.

  Define the New Transition Function:

  - The transition function δ for the new DFA operates by taking an input symbol and applying the transition functions of both original DFAs in parallel.
  - For each input symbol $a \in \Sigma$, the new DFA transitions from state (q_1, q_2) to state $(\delta_1(q_1, a), \delta_2(q_2, a))$.
  - In other words, if q_1 moves to q_1' on input a in D_1, and q_2 moves to q_2' on input a in D_2, the new DFA will move from (q_1, q_2) to (q_1', q_2').

  Define the New Set of Accepting (Final) States: The new DFA will accept a string if either of the original DFAs would accept it. Therefore, the set of final states F in the new DFA is defined as:

  $$F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$$

  This means that if either q_1 is a final state in D_1, or q_2 is a final state in D_2, the pair (q_1, q_2) is a final state in the new DFA.

  Note: It is possible in the new DFA (constructed as the union of two DFAs) to have states that are unreachable—meaning there are states in the DFA that cannot be reached from the start state. This typically happens because, in the product construction, we generate all possible pairs of states from the two original DFAs, but not all of these pairs are necessarily reachable.

The union of two finite automata (FAs) is useful for constructing a new automaton that recognizes any string accepted by either of the two original automata. This has several practical applications in theoretical computer science and programming:

- **Finding the intersection of two DFA's**: The process is basically the same as finding the union, but it differs in how we define the accepting states in the new machine, the accepting states will be

$$T = \{(q_1, q_2) : \ q_1 \in T_1 \text{ and } q_2 \in T_2\}.$$

Note: The intersection of two DFAs is useful in various practical applications where you need to accept only the strings that satisfy the conditions or rules of both automata

- **Concatenation of two DFA's**: The process is simple

For two machines $M_1(Q_1, \Sigma, q_{0_1}, T_1, \delta_1)$, and $M_2(Q_2, \Sigma, q_{0_2}, T_2, \delta_2)$

1. Connect the final states of the first machine to the start state of the second machine (With ϵ-transitions)
2. Clear T_1, There are no more final states in the first machine
3. Convert ϵ-NFA to DFA

Note: The concatenation of two DFAs has practical uses in many scenarios where the language of interest is the concatenation of two sublanguages. Concatenating two DFAs allows you to recognize strings that can be divided into two parts, where the first part is recognized by one DFA and the second part is recognized by the other.

- **Finding the union of two NFA's**: taking the union of two nondeterministic finite automata (NFAs) involves constructing a new NFA that accepts any string that is accepted by either of the original NFAs. This process can be done by creating a new NFA that combines the two original NFAs.

Given $M_1(Q_1, \Sigma, q_{0_1}, T_1, \delta_1)$, and $M_2(Q_2, \Sigma, q_{0_2}, T_2, \delta_2)$

1. **New start state**: Start by defining a new start state q_0', this state will have ϵ transitions to the start states of both machines.
2. **Define Q', the new set of states**: The new set of states will be the set of all states in M_1, and it will include all the states in M_2, along with the new start state. Thus,

$$Q' = Q_1 \cup Q_2 \cup \{q_0'\}.$$

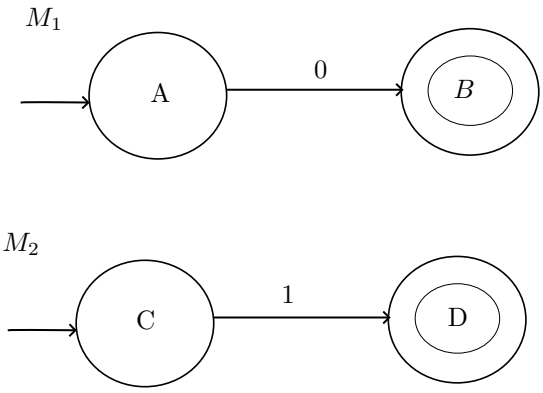
3. **Define the transition function**: The transition function δ' of the new NFA will include:
   - All the transitions of M_1 and M_2
   - Two ϵ transitions from the new start state to the start states of the two original machines q_{0_1} and q_{0_2}. Thus,

$$\delta'(q_0', \epsilon) = \{q_{0_1}, q_{0_2}\}.$$

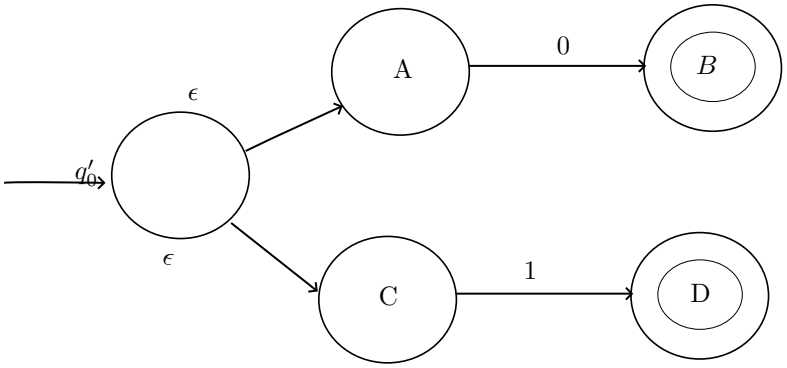
4. **Define the set of accepting states**: The set of accepting states will be

$$T' = T_1 \cup T_2.$$

Example:



$M_1 \cup M_2$ is then



- **Finding the intersection of two NFA's**: For NFAs, intersection is more complex because NFAs are nondeterministic and don't handle intersection naturally. Typically, you convert the NFAs to DFAs and then apply the DFA product construction
- **Concatenation of two NFA's**: The process is the same as with two DFA's (see above), but you don't need to convert to a DFA at the end.
- **Convert DFA into RE**: To convert a DFA (Deterministic Finite Automaton) into a Regular Expression (RE), you can use the state elimination method or generalized transition automaton method. This process works by gradually reducing the DFA's states and transitions until only a regular expression representing the entire language remains.

  Given a DFA $M(Q, \Sigma, \delta, q_0, F)$, the goal is to find a regular expression that represents the language recognized by this DFA.

  Process:

  1. **Add a new Start and accept state**:
     - Add a new start state q_s with an ϵ-transition (empty string) to the original start state q_0.

       Note: Once you add the new start state q_s with an ϵ-transition to the original start state q_0, q_0 is no longer considered the start state. Instead, q_0 becomes just another intermediate state in the automaton. The new start state is q_s, and it immediately transitions to q_0 without consuming any input (via the ϵ-transition).
     - Add a new accept state q_f and add ϵ-transitions from each of the original accept states to this new accept state q_f.

       Note: Similarly, when you add the new accept state q_f and connect it via ϵ-transitions from the original final states in F, the original final states are no longer considered final states in the sense of marking the end of a string's acceptance. Now, the new final state q_f serves as the sole final state, and the automaton reaches q_f via ϵ-transitions from the original final states.

     These new states simplify the process because now there's exactly one start state and one accept state.
  2. **Eliminate States One by One**:
     - The idea is to progressively eliminate states from the DFA while updating the transitions between the remaining states with regular expressions.
     - Every time you eliminate a state r, you need to update the regular expressions on the transitions between the remaining states to account for the paths that go through r.

     For any three states p, r, and q, if there is a path from p to q that goes through r, the new transition after eliminating r will include the regular expression:

$$R(p \to q) = R(p \to q) + R(p \to r)R(r \to r)^* R(r \to q)$$

Where:

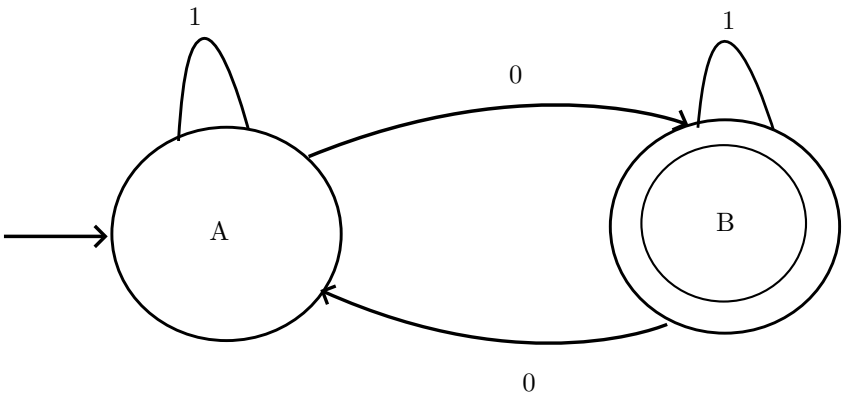
- $R(p \to q)$ is the regular expression for the direct transition from p to q.
- $R(p \to r)$ is the regular expression for the transition from p to r.
- $R(r \to r)$ is the regular expression for the loop on state r.
- $R(r \to q)$ is the regular expression for the transition from r to q.
- $+$ represents union, and $*$ represents the Kleene star (zero or more repetitions).

After updating the transitions, remove the state r.

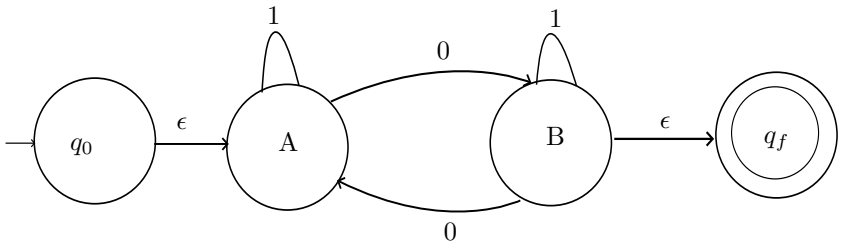
3. **Repeat the Elimination Until Only Two States Remain**: Continue eliminating states and updating the transitions until only two states remain: the start state q_s and the new accept state q_f.

   At this point, the regular expression on the transition from q_s to q_f represents the language of the DFA.

Example: For the alphabet $\Sigma = \{0, 1\}$, let's take the machine that accepts the strings with any number of ones, but the total number of zero's must be odd, and convert it to a RE.

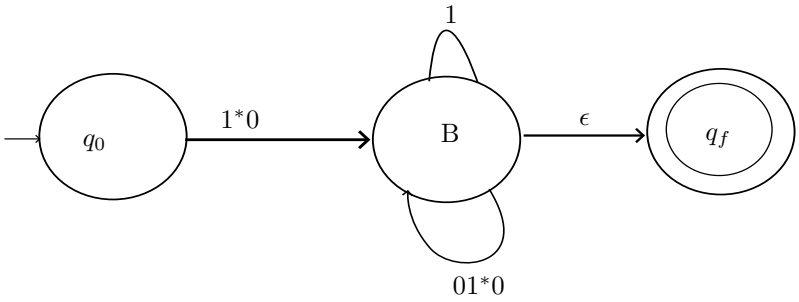


Let's start by making the new start and end states

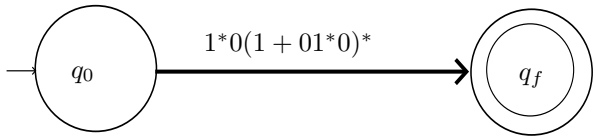


Now we start eliminating states, note that it does not matter in which order we eliminate the states, but for this example we will begin by eliminating state A. To get from the start state q_0 to state B, we need to pass through A, to get from A to B, we can have any number of $1's$ followed by a zero which takes us to be. Thus, the transition from q_0 to B is the regular expression 1^*0

We also have to consider the original transition from B to A, and then back to B, for this we have the RE 01^*0. Thus the machine becomes



To eliminate B, we need to consider the transitions through B ie from q_0 to q_f. We know to get from q_0 to B we have the RE 1^*0, then from B to q_f we have $(1+01^*0)^*$. Thus, the transition for q_0 to q_f is $1^*0(1+1^*01^*0)^*$. And the final machine with only one regular expression is



- **Properties of union, intersect, and concatenation for two FA's**: the properties of union, intersection, and concatenation for finite automata (FAs) are directly tied to the properties of regular languages.

  Union of two FA

  - **Closure**: The class of regular languages (those recognized by FA) is closed under union. This means the union of two regular languages is also regular, and there exists an FA that recognizes the union of the languages.
  - **Commutative:** Union is commutative for FA, meaning the order of combining automata does not matter.
  - **Associative:** Union is associative, so it doesn't matter how automata are grouped when performing multiple unions.
  - **Distributive over Intersection** Union distributes over intersection for regular languages, just as with sets.

  Intersection of two FA

  - **Closure:** The class of regular languages is also closed under intersection, meaning there is always an FA (typically constructed as a DFA) that recognizes the intersection of two regular languages.
  - **Commutative:** Intersection is commutative, meaning the order of combining automata doesn't matter.
  - **Associative:** Intersection is associative, so the grouping doesn't matter.
  - **Distributive over Union:** Intersection distributes over union for regular languages, just as with sets.

  Concatenation

  - **Closure:** Regular languages are closed under concatenation.
  - **Associativity:** Concatenation is associative. This means that the way in which you group the automata when performing concatenation doesn't matter.
  - **Identity Element:** The identity element for concatenation is the language that contains only the empty string,

$$L(A) \cdot \{\epsilon\} = L(A).$$

- **Distributivity Over Union:** Concatenation distributes over union. This means:

$$L(A) \cdot (L(B) \cup (L(C)) = L(A) \cdot L(B)) \cup (L(A) \cdot L(C)).$$

- **Concatenation with the Empty Set:** Concatenating any language with the empty set results in the empty set. This is because there are no strings to concatenate if one of the languages is empty:

Not commutative

DSA

2.1 C++ Stuff

2.1.1 Type declarations: Definitions and theorems

- **Discern any type**: Some rules,

  1. Start with the variable name, we read from inside to out
  2. const, %, *, and basic types go on the left
  3. const refers to what is immediately on the left (except for `const int*`), but the standard form of this is actually `int const*`. Thus, the exception to this is const is at the very left, then it refers to what is immediately right.
  4. arrays and functions go on the right, function args are type declaration subproblems

  The Algorithm:

  - Start with the variable name, or the implied name position
  - Read right until end or)
  - Read left until end or (
  - If something still left to read, move out one level of parenthesis and go to 2, else done.

  Thus, using parenthesis allows us to change direction, this will come in handy.

  Examples:

  - a is an int $\implies$ `int a`
  - a is a pointer to an int $\implies$ `int * a`
  - a is a pointer to a constant int $\implies$ `int const * a` (also `const int * a`)
  - a is a constant pointer to an int $\implies$ `int * const a`
  - a is a constant pointer to a constant int $\implies$ `int const * const a` (also `const int * const a`)
  - a is an array of 5 ints $\implies$ `int a[5]`
  - a is an array of 5 pointers to constant ints $\implies$ `int const * a[5]`
  - a is a pointer to an array of 5 constant ints $\implies$ `int const (* a)[5]`

- **Multi dimensional arrays (matrices)**: Think of multi-dimensional arrays as arrays of arrays. More indicative of what's happening internally. `float dat [3][4];` can be read as: "dat is an array of 3 arrays of 4 floats" (Using the algorithm from above).

  Examples:

  - `arg1` is a reference to an array of 25 constant pointers to arrays of 8 strings. $\implies$ `string (* const (& arg1)[25])[8]`

Note: Notice how we use parenthesis to change direction

- **Function Pointers**: Pointers point to bytes, which can be interpreted different ways. Pointers can point to bytes that can be interpreted as code, i.e. a function pointer.

  Examples:

  – f is a pointer to a function which takes an int and returns void. $\implies$ `void (* f) (int)`

2.1.2 G++

- **Compliation and linking**: Compilers turn source code into executable code.
  - **Source code → object code (Compilation)**: Object code is almost executable. It contains pieces that it provides to other objects, and holes to be filled in. It is a slow process
  - **Object code → executable (Linking)**: Connects pieces of object files together. This is a fast process

  Note: Many "compilers" do both compiling and linking. Most programs are built in two stages:

  1. Compile all the source code files
  2. Link the object code file into an executable

  This is the most efficient way to compile large projects. Changing a single source code file requires a small number of compilations (slow), followed by linking (fast).

- **Standard unix c compiler**: The standard is GNU gcc
- **Standard unix cpp compiler**: The standard is GNU g++
- g++ Options: With no options, g++ will go from source to an executable named a.out
  - **-o**: The -o option gives the name of the output file
  - **-c**: The -c option makes the compiler stop after the compilation stage. No linking is done. The name of the object code file is the same as the source with the extension replaced with .o
  - **-W**[***warning***]: Tell the complier to look for a specific warning
  - **-Wall (Warning all)**: There are many -W*warning* options, which warn of various conditions. -Wall warns about all of them. The compiler keeps going through warnings

    Note: A compiler warning is usually a bug waiting to happen. Do all you can to get rid of all warnings.
  - **-Werror**: The -Werror option turns all warnings into errors. The compiler aborts on an error.
  - **-g**: The -g option turns on debugging, and leaves much extra information in an object file. Executable is much larger, possibly slower.
  - **-0**: The -O option turns on optimization. There are several different levels of optimization, e.g. -O0, -O1, -O2, -O3.

    Note: Optimization may break your code, and -O and -g don't always work well together
  - **-I**[***directory***]: The -I option specifies an additional directory to search for include files. No space between -I and directory

    Thus,

    ```
    1 #include "./dir/headerfile" // Without -I
    2 #include "headerfile" // With -I : g++ -I./dir ...
    ```

  - **-L**[***directory***]: The -L option specifies an additional directory to search for libraries. No space between -L and directory.

Note: This option is meant for linking only. It has no effect in compilation.

- **-l[*librarymame*]**: The -l option specifies a library for linking. No space between -l and library name. The library name is related to the libray file name, but it is not identical. Library names start with "lib" and end with ".so.*" or ".a". These are removed. For example
  - The math library /lib/x86_64-linux/gnu/libm.so.6 is linked as -lm
  - The X11 graphics library /usr/lib/x86_64-linux-gnu/libX11.so is linked as -lX11

  Note: This option is for linking only. It has no effect in compilation. Libraries are the last things listed in a linking command.

  If you're linking against a library that is located in a non-standard directory (a directory that is not automatically searched by the linker, such as ./libs), then you need to tell the linker where to find that library using the -L option. Thus, -L tells the compiler where to look, -l specifies which one to grab.

2.1.3 STL Vectors

2.1.3.1 Implementation A vector models a dynamic array. Thus, a vector is an abstraction that manages its elements with a dynamic C-style arra

A vector copies its elements into its internal dynamic array. The elements always have a certain order. Thus, a vector is a kind of ordered collection. A vector provides random access. Thus, you can access every element directly in constant time, provided that you know its position. The iterators are random-access iterators, so you can use any algorithm of the STL.

2.1.3.2 Performance in operations on the end Vectors provide good performance if you append or delete elements at the end. If you insert or delete in the middle or at the beginning, performance gets worse. This is because every element behind has to be moved to another position. In fact, the assignment operator would be called for every following element.

2.1.3.3 Size and capacity Part of the way in which vectors give good performance is by allocating more memory than they need to contain all their elements. To use vectors effectively and correctly, you should understand how size and capacity cooperate in a vector.

Vectors provide the usual size operations size(), empty(), and max_size(). An additional "size" operation is the capacity() function, which returns the number of elements a vector could contain in its actual memory. If you exceed the capacity(), the vector has to reallocate its internal memory.

The capacity of a vector is important for two reasons:

1. Reallocation invalidates all references, pointers, and iterators for elements of the vector
2. Reallocation takes time.

Thus, if a program manages pointers, references, or iterators into a vector, or if speed is a goal, it is important to take the capacity into account

To avoid reallocation, you can use reserve() to ensure a certain capacity before you really need it. In this way, you can ensure that references remain valid as long as the capacity is not exceeded:

Another way to avoid reallocation is to initialize a vector with enough elements by passing additional arguments to the constructor. For example, if you pass a numeric value as parameter, it is taken as the starting size of the vector:

```
1 std::vector<T> v(5);
```

Note: If the only reason for initialization is to reserve memory, you should use reserve()

Unlike for strings, it is not possible to call reserve() for vectors to shrink the capacity. Calling reserve() with an argument that is less than the current capacity is a no-op

Because the capacity of vectors never shrinks, it is guaranteed that references, pointers, and iterators remain valid even when elements are deleted, provided that they refer to a position before the manipulated elements. However, insertions invalidate all references, pointers, and iterators when the capacity gets exceeded

2.1.3.4 Constructors

- `vector<Elem> c`
  Default constructor; creates an empty vector without any elements.
- `vector<Elem> c(c2)`
  Copy constructor; creates a new vector as a copy of `c2` (all elements are copied).
- `vector<Elem> c = c2`
  Copy constructor; creates a new vector as a copy of `c2` (all elements are copied).
- `vector<Elem> c(rv)`
  Move constructor; creates a new vector, taking the contents of the rvalue `rv` (since C++11).
- `vector<Elem> c = rv`
  Move constructor; creates a new vector, taking the contents of the rvalue `rv` (since C++11).
- `vector<Elem> c(n)`
  Creates a vector with `n` elements created by the default constructor.
- `vector<Elem> c(n, elem)`
  Creates a vector initialized with `n` copies of element `elem`.
- `vector<Elem> c(beg, end)`
  Creates a vector initialized with the elements of the range [`beg`, `end`].
- `vector<Elem> c{initlist}`
  Creates a vector initialized with the elements of the initializer list `initlist` (since C++11).
- `vector<Elem> c = {initlist}`
  Creates a vector initialized with the elements of the initializer list `initlist` (since C++11).
- `c. vector()`
  Destroys all elements and frees the memory.

2.1.3.5 Note about at() Out of all the element access operators and methods: [], at(), front(), back(), only at() performs range checking. If the index is out of range, at() throws an out_of_range

All other functions do not check. A range error results in undefined behavior. Calling operator [ ], front(), and back() for an empty container always results in undefined behavior:

2.1.3.6 Iterator methods

We have

- begin()
- end()
- rbegin()
- cbegin()
- cend()
- crbegin()
- crend()

2.1.4 STL Deque

2.1.4.1 Implementation A deque (pronounced "deck") is very similar to a vector. It manages its elements with a dynamic array, provides random access, and has almost the same interface as a vector. The difference is that with a deque, the dynamic array is open at both ends. Thus, a deque is fast for insertions and deletions at both the end and the beginning

To provide this ability, the deque is typically implemented as a bunch of individual blocks, with the first block growing in one direction and the last block growing in the opposite direction

2.1.4.2 Abilities, performance, uses The abilities of deques differ from those of vectors as follows:

- Inserting and removing elements is fast at both the beginning and the end (for vectors, it is fast only at the end). These operations are done in amortized constant time.
- The internal structure has one more indirection to access the elements, so with deques, element access and iterator movement are usually a bit slower.
- Iterators must be smart pointers of a special type rather than ordinary pointers because they must jump between different blocks.
- In systems that have size limitations for blocks of memory (for example, some PC systems), a deque might contain more elements because it uses more than one block of memory. Thus, max_size() might be larger for deques.
- Deques provide no support to control the capacity and the moment of reallocation. In particular, any insertion or deletion of elements other than at the beginning or end invalidates all pointers, references, and iterators that refer to elements of the deque. However, reallocation may perform better than for vectors because according to their typical internal structure, deques don't have to copy all elements on reallocation.
- Blocks of memory might get freed when they are no longer used, so the memory size of a deque might shrink (however, whether and how this happens is implementation specific).

2.1.4.3 When to use deques

- You insert and remove elements at both ends (this is the classic case for a queue).
- You don't refer to elements of the container.
- It is important that the container frees memory when it is no longer used (however, the standard does not guarantee that this happens).

2.1.4.4 Constructors

- `deque<Elem> c`
  Default constructor; creates an empty deque without any elements.
- `deque<Elem> c(c2)`
  Copy constructor; creates a new deque as a copy of `c2` (all elements are copied).
- `deque<Elem> c = c2`
  Copy assignment operator; creates a new deque as a copy of `c2` (all elements are copied).
- `deque<Elem> c(rv)`
  Move constructor; creates a new deque, taking the contents of the rvalue `rv` (since C++11).
- `deque<Elem> c = rv`
  Move assignment operator; creates a new deque, taking the contents of the rvalue `rv` (since C++11).
- `deque<Elem> c(n)`
  Creates a deque with `n` elements created by the default constructor.
- `deque<Elem> c(n, elem)`
  Creates a deque initialized with `n` copies of element `elem`.
- `deque<Elem> c(beg, end)`
  Creates a deque initialized with the elements of the range `[beg, end]`.
- `deque<Elem> c {inilist}`
  Creates a deque initialized with the elements of initializer list `inilist` (since C++11).
- `deque<Elem> c = {inilist}`
  Creates a deque initialized with the elements of initializer list `inilist` (since C++11).
- `c.~deque()`
  Destroys all elements and frees the memory.

Deque operations differ from vector operations in only two ways:

1. Deques do not provide the functions for capacity (capacity() and reserve()).
2. Deques do provide direct functions to insert and to delete the first element (push_front() and pop_front()).

2.1.5 STL Lists

2.1.5.1 Implementation Manages its elements as a doubly linked list. As usual, the C++ standard library does not specify the kind of the implementation, but it follows from the list's name, constraints, and specifications.

2.1.5.2 Abilities The internal structure of a list is totally different from that of an array, a vector, or a deque. The list object itself provides two pointers, the so-called anchors, which refer to the first and last elements. Each element has pointers to the previous and next elements (or back to the anchor). To insert a new element, you just manipulate the corresponding pointers

Thus, a list differs in several major ways from arrays, vectors, and deques:

- A list does not provide random access. For example, to access the fifth element, you must navigate the first four elements, following the chain of links. Thus, accessing an arbitrary element using a list is slow. However, you can navigate through the list from both end. So accessing both the first and the last elements is fast.
- Inserting and removing elements is fast at each position (provided you are there), and not only at one or both ends. You can always insert and delete an element in constant time, because no other elements have to be moved. Internally, only some pointer values are manipulated.
- Inserting and deleting elements does not invalidate pointers, references, and iterators to other elements.
- A list supports exception handling in such a way that almost every operation succeeds or is a no-op. Thus, you can't get into an intermediate state in which only half of the operation is complete.

2.1.5.3 Differencs in the methods The member functions provided for lists reflect these differences from arrays, vectors, and deques as follows:

- Lists provide front(), push_front(), and pop_front(), as well as back(), push_back(), and pop_back().
- Lists provide neither a subscript operator nor at(), because no random access is provided.
- Lists don't provide operations for capacity or reallocation, because neither is needed. Each element has its own memory that stays valid until the element is deleted.
- Lists provide many special member functions for moving and removing elements. These member functions are faster versions of general algorithms that have the same names. They are faster because they only redirect pointers rather than copy and move the values.

2.1.5.4 Constructors

- `list<Elem> c`
  Default constructor; creates an empty list without any elements.
- `list<Elem> c(c2)`
  Copy constructor; creates a new list as a copy of `c2` (all elements are copied).
- `list<Elem> c = c2`
  Copy assignment operator; creates a new list as a copy of `c2` (all elements are copied).
- `list<Elem> c(rv)`
  Move constructor; creates a new list, taking the contents of the rvalue `rv` (since C++11).
- `list<Elem> c = rv`
  Move assignment operator; creates a new list, taking the contents of the rvalue `rv` (since C++11).
- `list<Elem> c(n)`
  Creates a list with `n` elements created by the default constructor.
- `list<Elem> c(n, elem)`
  Creates a list initialized with `n` copies of element `elem`.
- `list<Elem> c(beg, end)`
  Creates a list initialized with the elements of the range `[beg, end]`.
- `list<Elem> c{inilist}`
  Creates a list initialized with the elements of initializer list `inilist` (since C++11).
- `list<Elem> c = {inilist}`
  Creates a list initialized with the elements of initializer list `inilist` (since C++11).
- `c.Iist()`
  Destroys all elements and frees the memory.

2.1.5.5 Element access With lists, we only have front and back methods. However, these methods do not check for existence. Calling these methods on empty containers results in undefined behavior

Thus, the caller must ensure that the container contains at least one element

2.1.5.6 Iterator functions To access all elements of a list, you must use iterators. Lists provide the usual iterator functions. However, because a list has no random access, these iterators are only bidirectional. Thus, you can't call algorithms that require random-access iterators. All algorithms that manipulate the order of elements a lot, especially sorting algorithms, are in this category. However, for sorting the elements, lists provide the special member function sort()

2.1.5.7 Splice Functions and Functions to Change the Order of Elements Linked lists have the advantage that you can remove and insert elements at any position in constant time. If you move elements from one container to another, this advantage doubles in that you need only redirect some internal pointers

To support this ability, lists provide not only remove() but also additional modifying member functions to change the order of and relink elements and ranges.

2.1.6 STL Forward lists

2.1.6.1 Implementation

A forward list (an instance of the container class forward_list<>), which was introduced with C++11, manages its elements as a singly linked list

Conceptionally, a forward list is a list (object of class list<>) restricted such that it is not able to iterate backward. It provides no functionality that is not also provided by lists. As benefits, it uses less memory and provides slightly better runtime behavior. The standard states: "It is intended that forward_list have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted.

2.1.6.2 Abilities, limitations

Forward lists have the following limitations compared to lists:

- A forward list provides only forward iterators, not bidirectional iterators. As a consequence, no reverse iterator support is provided, which means that types, such as reverse_iterator, and member functions, such as rbegin(), rend(), crbegin(), and crend(), are not provided.
- A forward list does not provide a size() member function. This is a consequence of omitting features that create time or space overhead relative to a handwritten singly linked list.
- The anchor of a forward list has no pointer to the last element. For this reason, a forward list does not provide the special member functions to deal with the last element, back(), push_back(), and pop_back().
- For all member functions that modify forward lists in a way that elements are inserted or deleted at a specific position, special versions for forward lists are provided. The reason is that you have to pass the position of the element before the first element that gets manipulated, because there you have to assign a new successor element. Because you can't navigate backwards (at least not in constant time), for all these member functions you have to pass the position of the preceding element. Because of this difference, these member functions have a _after suffix in their name. For example, instead of insert(), insert_after() is provided, which inserts new elements after the element passed as first argument; that is, it appends an element at that position.

- For this reason, forward lists provide before_begin() and cbefore_begin(), which yield the position of a virtual element before the first element (technically speaking, the anchor of the linked list), which can be used to let built-in algorithms ending with $_{a}fterexchangeeventhefirstelement$.

2.1.6.3 No size()?

The decision not to provide size() might be especially surprising because size() is one of the operations required for all STL containers. Here, you can see the consequences of the design goal to have "zero space or time overhead relative to a hand-written Cstyle singly linked list." The alternative would have been either to compute the size each time size() is called, which would have linear complexity, or to provide an additional field in the forward_list object for the size, which is updated with each and every operation that changes the number of elements. As the design paper for the forward list, "It's a cost that all users would have to pay for, whether they need this feature or not." So, if you need the size, either track it outside the forward_list or use a list instead.

If you have to compute the number of elements, you can use distance()

```
#include <forward_list>
#include <iterator>

std::forward_list<int> l;
std::cout << "Size: " << std::distance(l.begin(), l.end()) <<
    ↪  std::endl;
```

2.1.6.4 Similarities to list

- A forward list does not provide random access. For example, to access the fifth element, you
- must navigate the first four elements, following the chain of links. Thus, using a forward list to access an arbitrary element is slow.
- Inserting and removing elements is fast at each position, if you are there. You can always insert and delete an element in constant time, because no other elements have to be moved. Internally, only some pointer values are manipulated.
- Inserting and deleting elements does not invalidate iterators, references, and pointers to other elements.
- A forward list supports exception handling in such a way that almost every operation succeeds or is a no-op. Thus, you can't get into an intermediate state in which only half of the operation is complete.
- Forward lists provide many special member functions for moving and removing elements. These member functions are faster versions of general algorithms, because they only redirect pointers rather than copy and move the values. However, when element positions are involved, you have to pass the preceding position, and the member function has the suffix _after in its name.

2.1.6.5 Constructors

- `forward_list<Elem> c`
  Default constructor; creates an empty forward list without any elements.
- `forward_list<Elem> c(c2)`
  Copy constructor; creates a new forward list as a copy of `c2` (all elements are copied).
- `forward_list<Elem> c = c2`
  Copy assignment operator; creates a new forward list as a copy of `c2` (all elements are copied).
- `forward_list<Elem> c(rv)`
  Move constructor; creates a new forward list, taking the contents of the rvalue `rv` (since C++11).
- `forward_list<Elem> c = rv`
  Move assignment operator; creates a new forward list, taking the contents of the rvalue `rv` (since C++11).
- `forward_list<Elem> c(n)`
  Creates a forward list with `n` elements created by the default constructor.
- `forward_list<Elem> c(n, elem)`
  Creates a forward list initialized with `n` copies of element `elem`.
- `forward_list<Elem> c(beg, end)`
  Creates a forward list initialized with the elements of the range `[beg, end]`.
- `forward_list<Elem> c{inilist}`
  Creates a forward list initialized with the elements of initializer list `inilist` (since C++11).
- `forward_list<Elem> c = {inilist}`
  Creates a forward list initialized with the elements of initializer list `inilist` (since C++11).
- `c.forward_list()`
  Destroys all elements and frees the memory.

2.1.7 STL Sets and multisets

2.1.7.1 Implementation

Sets and multisets are implemented as height balanced binary search trees. (red-black trees)

Set and multiset containers sort their elements automatically according to a certain sorting criterion. The difference between the two types of containers is that multisets allow duplicates, whereas sets do not

The elements of a set or a multiset may have any type T that is comparable according to the sorting criterion. The optional second template argument defines the sorting criterion. If a special sorting criterion is not passed, the default criterion less is used. The function object less sorts the elements by comparing them with operator <

The optional third template parameter defines the memory model. The default memory model is the model allocator, which is provided by the C++ standard library.

2.1.7.2 Strict weak ordering

The sorting criterion must define strict weak ordering, which is defined by the following four properties:

1. It has to be **antisymmetric**.
   - This means that for operator `<`: If $x < y$ is true, then $y < x$ is false.
   - This means that for a predicate `op()`: If `op(x, y)` is true, then `op(y, x)` is false.
2. It has to be **transitive**.
   - This means that for operator `<`: If $x < y$ is true and $y < z$ is true, then $x < z$ is true.
   - This means that for a predicate `op()`: If `op(x, y)` is true and `op(y, z)` is true, then `op(x, z)` is true.
3. It has to be **irreflexive**.
   - This means that for operator `<`: $x < x$ is always false.
   - This means that for a predicate `op()`: `op(x, x)` is always false.
4. It has to have **transitivity of equivalence**, which means roughly: If a is equivalent to b and b is equivalent to c, then a is equivalent to c.
   - This means that for operator `<`: If $!(a < b)$ && $!(b < a)$ is true and $!(b < c)$ && $!(c < b)$ is true, then $!(a < c)$ && $!(c < a)$ is true.
   - This means that for a predicate `op()`: If `op(a, b)`, `op(b, a)`, `op(b, c)`, and `op(c, b)` all yield false, then `op(a, c)` and `op(c, a)` yield false.

Note: Note that this means that you have to distinguish between less and equal. A criterion such as operator `<=` does not fulfill this requirement.

Based on these properties, the sorting criterion is also used to check equivalence. That is, two elements are considered to be duplicates if neither is less than the other (or if both `op(x, y)` and `op(y, x)` are false).

For multisets, the order of equivalent elements is random but stable. Thus, insertions and erasures preserve the relative ordering of equivalent elements (guaranteed since C++11).

2.1.7.3 Abilities

Like all standardized associative container classes, sets and multisets are usually implemented as balanced binary trees

The major advantage of automatic sorting is that a binary tree performs well when elements with a certain value are searched. In fact, search functions have logarithmic complexity. For example, to search for an element in a set or a multiset of 1,000 elements, a tree search performed by a member function needs, on average, one-fiftieth of the comparisons of a linear search

2.1.7.4 Changing elements directly, no direct element access Automatic sorting also imposes an important constraint on sets and multisets: You may not change the value of an element directly

Therefore, to modify the value of an element, you must remove the element having the old value and insert a new element that has the new value. The interface reflects this behavior:

- Sets and multisets don't provide operations for direct element access.
- Indirect access via iterators has the constraint that, from the iterator's point of view, the element value is constant.

2.1.7.5 Constructors

- `set c`
  Default constructor; creates an empty set/multiset without any elements.
- `set c(op)`
  Creates an empty set/multiset that uses `op` as the sorting criterion.
- `set c(c2)`
  Copy constructor; creates a copy of another set/multiset of the same type (all elements are copied).
- `set c = c2`
  Copy assignment operator; creates a copy of another set/multiset of the same type (all elements are copied).
- `set c(rv)`
  Move constructor; creates a new set/multiset of the same type, taking the contents of the rvalue `rv` (since C++11).

- `set c = rv`
  Move assignment operator; creates a new set/multiset of the same type, taking the contents of the rvalue `rv` (since C++11).
- `set c(beg, end)`
  Creates a set/multiset initialized by the elements of the range `[beg, end]`.
- `set c(beg, end, op)`
  Creates a set/multiset with the sorting criterion `op` initialized by the elements of the range `[beg, end]`.
- `set c{inilist}`
  Creates a set/multiset initialized with the elements of initializer list `inilist` (since C++11).
- `set c = {inilist}`
  Creates a set/multiset initialized with the elements of initializer list `inilist` (since C++11).
- `c.šet()`
  Destroys all elements and frees the memory.

2.1.7.6 Types

- `set<Elem>`
  A set that by default sorts with `less<>` (operator `<`).
- `set<Elem, Op>`
  A set that by default sorts with `Op`.
- `multiset<Elem>`
  A multiset that by default sorts with `less<>` (operator `<`).
- `multiset<Elem, Op>`
  A multiset that by default sorts with `Op`.

2.1.8 STL Maps and multimaps

Maps and multimaps are containers that manage key/value pairs as elements. These containers sort their elements automatically, according to a certain sorting criterion that is used for the key. The difference between the two is that multimaps allow duplicates, whereas maps do not

2.1.8.1 Implementation

Maps and multimaps are implemented the same as sets and multisets, height balanced binary search trees (red-black trees).

2.1.8.2 Template parameters

The first template parameter is the type of the element's key, and the second template parameter is the type of the element's associated value. The elements of a map or a multimap may have any types Key and T that meet the following two requirements:

1. Both key and value must be copyable or movable.
2. The key must be comparable with the sorting criterion.

The optional third template parameter defines the sorting criterion. As for sets, this sorting criterion must define a "strict weak ordering" The elements are sorted according to their keys, so the value doesn't matter for the order of the elements. The sorting criterion is also used to check for equivalence; that is, two elements are equal if neither key is less than the other.

If a special sorting criterion is not passed, the default criterion less<> is used. The function object less<> sorts the elements by comparing them with operator <

2.1.8.3 Abilities

Sets, multisets, maps, and multimaps typically use the same internal data type. So, you could consider sets and multisets as special maps and multimaps, respectively, for which the value and the key of the elements are the same objects. Thus, maps and multimaps have all the abilities and operations of sets and multisets. Some minor differences exist, however. First, their elements are key/value pairs. In addition, maps can be used as associative arrays.

Maps and multimaps sort their elements automatically, according to the element's keys, and so have good performance when searching for elements that have a certain key. Searching for elements that have a certain value promotes bad performance. Automatic sorting imposes an important constraint on maps and multimaps: You may not change the key of an element directly, because doing so might compromise the correct order. To modify the key of an element, you must remove the element that has the old key and insert a new element that has the new key and the old value. As a consequence, from the iterator's point of view, the element's key is constant. However, a direct modification of the value of the element is still possible, provided that the type of the value is not constant.

2.1.8.4 Constructors and types

- `map c`
  Default constructor; creates an empty map/multimap without any elements.
- `map c(op)`
  Creates an empty map/multimap that uses `op` as the sorting criterion.
- `map c(c2)`
  Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied).
- `map c = c2`
  Copy assignment operator; creates a copy of another map/multimap of the same type (all elements are copied).
- `map c(rv)`
  Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue `rv` (since C++11).
- `map c = rv`
  Move assignment operator; creates a new map/multimap of the same type, taking the contents of the rvalue `rv` (since C++11).
- `map c(beg, end)`
  Creates a map/multimap initialized by the elements of the range `[beg, end]`.
- `map c(beg, end, op)`
  Creates a map/multimap with the sorting criterion `op` initialized by the elements of the range `[beg, end]`.
- `map c{inilist}`
  Creates a map/multimap initialized with the elements of initializer list `inilist` (since C++11).
- `map c = {inilist}`
  Creates a map/multimap initialized with the elements of initializer list `inilist` (since C++11).
- `c.~map()`
  Destroys all elements and frees the memory.

Here, map may be one of the following types:

- `map<Key, Val>`
  A map that by default sorts keys with `less<>` (operator `<`).
- `map<Key, Val, Op>`
  A map that by default sorts keys with `Op`.
- `multimap<Key, Val>`
  A multimap that by default sorts keys with `less<>` (operator `<`).
- `multimap<Key, Val, Op>`
  A multimap that by default sorts keys with `Op`.

2.1.8.5 Using maps as associative arrays Associative containers don't typically provide abilities for direct element access. Instead, you must use iterators. For maps, as well as for unordered maps, however, there is an exception to this rule. Nonconstant maps provide a subscript operator for direct element access. In addition, since C++11, a corresponding member function at() is provided for constant and nonconstant maps

at() yields the value of the element with the passed key and throws an exception object of type out_of_range if no such element is present

For operator [ ], the index also is the key that is used to identify the element. This means that for operator [ ], the index may have any type rather than only an integral type. Such an interface is the interface of a so-called associative array.

For operator [ ], the type of the index is not the only difference from ordinary arrays. In addition, you can't have a wrong index. If you use a key as the index for which no element yet exists, a new element gets inserted into the map automatically. The value of the new element is initialized by the default constructor of its type. Thus, to use this feature, you can't use a value type that has no default constructor. Note that the fundamental data types provide a default constructor that initializes their values to zero

2.1.9 STL Unordered containers

2.1.10 STL Containers: Implementations

- **std::vector**
  - Implemented as a dynamically resizable array with contiguous memory.
- **std::deque**
  - Implemented as a sequence of dynamically allocated arrays (blocks) for efficient insertion/removal at both ends.
- **std::list**
  - Implemented as a doubly linked list, where each node contains pointers to the previous and next nodes.
- **std::forward_list**
  - Implemented as a singly linked list, where each node contains a pointer to the next node.
- **std::set / std::multiset**
  - Implemented as a self-balancing binary search tree (typically Red-Black Tree).
- **std::unordered_set / std::unordered_multiset**
  - Implemented as a hash table with separate chaining or open addressing for collision resolution.
- **std::map / std::multimap**
  - Implemented as a self-balancing binary search tree (typically Red-Black Tree) for sorted key-value pairs.
- **std::unordered_map / std::unordered_multimap**
  - Implemented as a hash table with separate chaining or open addressing for key-value pairs.

2.1.11 STL Containers: Iterator Functions

- **Containers with all the iterator functions (begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend())**:
  1. Vector
  2. Deque
  3. List
  4. Set
  5. Muliset
  6. Map
  7. Mulimap
  8. Unordered set
  9. unordered multiset
  10. unordered map
  11. unordered multimap
- **Containers with limited iterator support:**
  1. **Forward_list**: Only supports forward iterators (begin(), end(), cbegin(), cend()).

2.1.12 STL containers: Main concepts, differences, uses

- **Vectors**:
  - Dynamic array, automatic resizing.
  - We have access to capacity and reserve methods.
  - Fast at end operations.
  - Contiguous memory, random access.
  - at() method to index with error checking.
- **Deque**:
  - Multiple blocks / Dynamic arrays to give access to both ends.
  - Fast at both ends.
  - Front and back operations.
  - Slower iterator access compared to vectors.
  - Iterators are smart pointers.
  - No capacity access
- **List:**
  - Doubly-linked list
  - Insertion and removing is fast
  - Access at any element thats not the first or last is slow.
  - NO random access
  - Member method to sort
  - Splice
  - Unique
  - Merge
- **Forward_list**
  - Singly linked list
  - No size method
  - No reverse iterators
  - No pointer to last element, no back(), push_back(), or pop_back() methods
  - For all member functions that modify forward lists in a way that elements are inserted or deleted at a specific position, special versions for forward lists are provided. The reason is that you have to pass the position of the element before the first element that gets manipulated, because there you have to assign a new successor element. Because you can't navigate backwards (at least not in constant time), for all these member functions you have to pass the position of the preceding element. Because of this difference, these member functions have a after suffix in their name. For example, instead of insert(), insert_after() is provided, which inserts new elements after the element passed as first argument; that is, it appends an element at that position.

For this reason, forward lists provide before_begin() and cbefore_begin(), which yield the position of a virtual element before the first element (technically speaking, the anchor of the linked list), which can be used to let built-in algorithms ending with after exchange even the first element

- **Sets and multisets**
  - Height balanced bst
  - No duplicates in set, can have duplicates in multiset
  - logarithmic searching
  - logarithmic insertion and deletion
  - Automatic sorting
  - Can't change elements directly
  - No direct element access
  - Constant iterators

2.1.13 STL Containers: Iterator invalidation

- **Vectors:**
  - **Insertion:** All iterators are invalidated if a reallocation occurs; otherwise, only iterators at or after the point of insertion are invalidated.
  - **Deletion:** Iterators at or after the point of deletion are invalidated.
- **Deque**:
  - **Insertion/Deletion:** At beginning or end, no invalidation unless reallocation occurs. Inserting or deleting in the middle invalidates all iterators.
- **List**:
  - **Insertion:** No invalidation.
  - **Deletion:** Only the iterator to the erased element is invalidated.
- **Forward list**:
  - **Insertion:** No invalidation.
  - **Deletion:** Only the iterator to the erased element is invalidated.
- **Set/multiset**:
  - **Insertion:** No invalidation.
  - **Deletion:** Only the iterator to the erased element is invalidated.
- **unordered set/unordered multiset**:
  - **Insertion:** No invalidation unless rehashing occurs.
  - **Deletion:** Only the iterator to the erased element is invalidated.
  - **Rehashing:** All iterators are invalidated.
- **Map/Multimap**:
  - **Insertion:** No invalidation.
  - **Deletion:** Only the iterator to the erased element is invalidated.
- **Unordered map/unordered multimap**:
  - **Insertion:** No invalidation unless rehashing occurs.
  - **Deletion:** Only the iterator to the erased element is invalidated.
  - **Rehashing:** All iterators are invalidated.

2.1.14 STL Containers: Reallocation

- **Vectors**: Reallocation occurs when inserting elements exceeds the current capacity.
- **Deque**: Reallocation occurs when inserting elements requires more blocks (typically at both ends, but can happen internally).
- **List, forward list**: No reallocation occurs, as they allocate nodes dynamically and do not store elements contiguously.
- **set, multiset, map, multimap**: No reallocation occurs, as they use balanced trees, and elements are not stored contiguously.
- **unordered set, unordered multiset, unordered map, unordered multimap**: Reallocation occurs when the load factor exceeds a threshold, triggering a rehash to a larger bucket array.

2.1.15 STL Containers: Element access

- **std::vector**
  - Direct access via index: `v[i]`, `v.at(i)`
  - Front element: `v.front()`
  - Back element: `v.back()`
- **std::deque**
  - Direct access via index: `d[i]`, `d.at(i)`
  - Front element: `d.front()`
  - Back element: `d.back()`
- **std::list**
  - No direct access via index.
  - Front element: `l.front()`
  - Back element: `l.back()`
- **std::forward_list**
  - No direct access via index.
  - Front element: `fl.front()`
- **std::set / std::multiset**
  - No direct access via index.
  - Access via iterator or functions like `find()`, `lower_bound()`, `upper_bound()`.
- **std::unordered_set / std::unordered_multiset**
  - No direct access via index.
  - Access via iterator or `find()`.
- **std::map / std::multimap**
  - Access by key: `m[key]` (for `std::map` only, not `std::multimap`).
  - Access via iterator or functions like `find()`, `lower_bound()`, `upper_bound()`.
- **std::unordered_map / std::unordered_multimap**
  - Access by key: `um[key]` (for `std::unordered_map` only, not `std::unordered_multimap`).
  - Access via iterator or `find()`.

2.1.16 STL Containers: Uses and advantages

- **std::vector**
  - Advantages: Fast random access, contiguous memory, efficient for dynamic arrays.
  - Uses: When frequent random access and dynamic resizing are needed.
- **std::deque**
  - Advantages: Fast insertion/removal at both ends, efficient dynamic array.
  - Uses: Double-ended queue operations, efficient at both front and back.
- **std::list**
  - Advantages: Constant time insertion/removal anywhere, no reallocation.
  - Uses: When frequent insertions/removals in the middle are needed.
- **std::forward_list**
  - Advantages: Singly linked list, smaller memory overhead, constant time insertion/removal.
  - Uses: Memory-constrained environments, where only forward traversal is needed.
- **std::set / std::multiset**
  - Advantages: Sorted elements, fast lookup (logarithmic time).
  - Uses: When you need a sorted collection with unique or non-unique elements.
- **std::unordered_set / std::unordered_multiset**
  - Advantages: Fast average-time lookup (constant time), no sorting.
  - Uses: When fast lookup is needed without element ordering.
- **std::map / std::multimap**
  - Advantages: Sorted key-value pairs, fast lookup (logarithmic time).
  - Uses: Key-value pairs where keys must remain sorted.
- **std::unordered_map / std::unordered_multimap**
  - Advantages: Fast average-time lookup (constant time), no sorting.
  - Uses: Key-value pairs where fast lookup is needed without ordering.

2.1.17 STL Iterators

An object that iterates/navigates over elements in the container. They are essentially an abstraction of pointer

- **Some notes about iterators**:
  1. each container provides its own iterator
  2. interfaces of iterators of different containers are largely the same
  3. internal behaviors depend on the data structure of the container
- **Operations**:
  1. operator * returns the element of the current positions
  2. operator -> access a member of the element
  3. operator ++ step forward
  4. operator – step backward
  5. operator == and != whether two iterators represent the same position
  6. operator = assign an iterator
- **Important iterators**:
  1. begin() gets you the beginning of a container
  2. end() gets you just past the end
- **Iterator types**
  - **iterator**
  - **reverse_iterator**
  - **const_iterator**
  - **const_reverse_iterator**
- **Iterator categories**
  1. **Input Iterator:**
     - **Purpose:** Read-only access to elements in a single-pass manner.
     - **Operations:** Can be incremented (++), compared for equality (==), and dereferenced (*) to access elements.
  2. **Output Iterator:**
     - **Purpose:** Write-only access to elements in a single-pass manner.
     - **Operations:** Can be incremented (++) and dereferenced (*) to assign values.
  3. **Forward Iterator:**
     - **Purpose:** Read and write access to elements; can traverse the container in a single direction.
     - **Operations:** Can be incremented (++), compared for equality (==), and dereferenced (*).
  4. **Bidirectional iterator**:
     - **Purpose:** Read and write access to elements; can traverse the container in both directions.
     - **Operations:** Supports both increment (++) and decrement (–) operations.

5. **Random Access Iterator:**
   - **Purpose**: Read and write access with the ability to jump to any element in constant time.
   - **Operations:** Supports all operations of bidirectional iterators plus direct arithmetic operations like addition (+), subtraction (-), and subscript ([]).

- **Containers and their iterators**:

  1. **Vector**: Random access iterator
  2. **Deque**: Random access iterator
  3. **List**: Bidirectional iterator
  4. **Forward_list**: Forward iterator
  5. **Set**: Bidirectional iterator
  6. **Multiset**: Bidirectional iterator
  7. **Map**: Bidirectional iterator
  8. **Multimap**: Bidirectional iterator
  9. **Unordered_map**: Forward iterator

- **Insert iterators**: Insert iterators in C++ are special types of iterators that allow you to insert elements into a container at specific positions rather than overwriting existing elements. There are three primary types of insert iterators provided by the C++ Standard Library:

  if a container has an insert method, you can and often should use it directly when inserting elements, especially if you want to insert a single element or a specific range of elements into the container.

  Containers that have an insert method are: vector, deque, list, forward list, set, multiset, unordered set, unordered multiset,map, multimap, unordered map, unordered multi map.

  Insert iterators (std::back_inserter, std::front_inserter, and std::inserter) should be used when working with algorithms or situations where automatic insertion logic simplifies your code.

  Some things in <algorithm> require these inserters

  1. **std::front_inserter**: Inserts elements at the front of a container. Calls the container's push_front method to add elements to the front. Used with containers that support push_front

```cpp
#include <list>
#include <algorithm>
#include <iterator>

int main() {
    std::list<int> lst = {1, 2, 3};
    std::list<int> to_add = {4, 5, 6};

    // Insert elements at the front of lst
    std::copy(to_add.begin(), to_add.end(),
    ↪ std::front_inserter(lst));

    // lst now contains: 6, 5, 4, 1, 2, 3
}
```

2. **std::back_inserter**: Inserts elements at the end of a container. Inserts elements at the end of a container. Used with containers that support push_back

```cpp
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
    std::vector<int> vec = {1, 2, 3};
    std::vector<int> to_add = {4, 5, 6};

    // Insert elements at the end of vec
    std::copy(to_add.begin(), to_add.end(),
    ↪ std::back_inserter(vec));

    // vec now contains: 1, 2, 3, 4, 5, 6
}
```

3. **std::inserter**: Inserts elements at a specific position in a container. Takes an iterator indicating the insertion position and calls the container's insert method. Used with containers that support insertion at arbitrary positions

```cpp
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
    std::vector<int> vec = {1, 2, 3};
    std::vector<int> to_add = {4, 5, 6};

    // Insert elements starting at the second position (before 2)
    std::copy(to_add.begin(), to_add.end(), std::inserter(vec, vec.begin() + 1));

    // vec now contains: 1, 4, 5, 6, 2, 3
}
```

Databases

3.1 Introduction to databases (db concepts)

3.1.1 Definitions and theorems

- **What is a database?**: A database is a collection of stored operational data used by the application systems of some particular enterprise, better yet a collection of related data.
- **What is an enterprise?**: a generic term for any reasonably large-scale commercial, scientific, technical, or other application. Such as
  - Manufacturing
  - Financial
  - Medical
  - University
  - Government
- **Operational data**: Data maintained about the operation of an enterprise, such as
  - Products
  - Accounts
  - Patients
  - Students
  - Plans

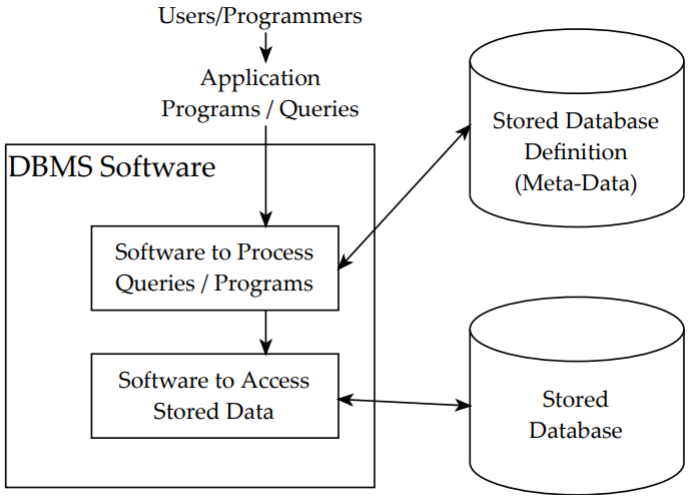
  Note: Notice that this DOES NOT include input/output data
- **Database Management System (DBMS)**: A Database Management System (DBMS) is a collection of programs that enables users to create and maintain a database. Ie a general-purpose software system that facilitates
  - Definition of databases
  - Construction of databases
  - Manipulation of data within a database
  - Sharing of data between users/applications
- **Defining a database**: For the data being stored in the database, defining the database specifies
  - The data types
  - The structures
  - The constraints
- **Constructing a Database**: Constructing a database is the process of storing the data itself on some storage device

Note: The storage device is controlled by the DBMS

- **Manipulating a Database**
  - retrieve specific information in a query
  - update the database to include changes
  - generate reports from the data

  Most likely already defined by whatever dbms you choose

- **Sharing a Database**: Sharing a database Allows multiple users and programs to access the database at the same time, any conflicts between applications are handled by the DBMS
- **Other Important Functions of a Database**: Other important functions provided by a DBMS include
  - Protection, system protection, security protection
  - Maintenence, allows updates to be performed easily
- **Simplified Database System Environment**:



- **Main characteristics of a database system are:**
  - Self-describing nature of a database system
  - Insulation between programs and data, and data abstraction
  - Support for multiple views of the data
  - Sharing of data and multi-user transaction processing
- **Other Capabilities of DBMS Systems**: Support for at least one data model through which the user can view the data, There is at least one abstract model of data that allows the user to see the "information" in the database, Relational, hierarchical, network, inverted list, or object-oriented

Support for at least one data model through which the user can view the data

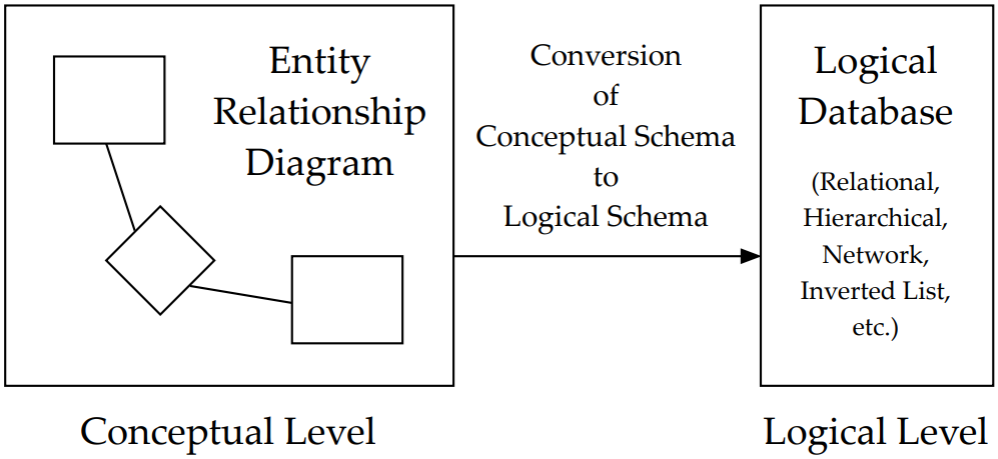
- efficient file access which allows us to "find the boss of Susie Jones"
- allows us to "navigate" within the data
- allows us to combine values in 2 or more databases to obtain "information"

Support for high-level languages that allow the user to define the structure of the data, access that data, and manipulate it

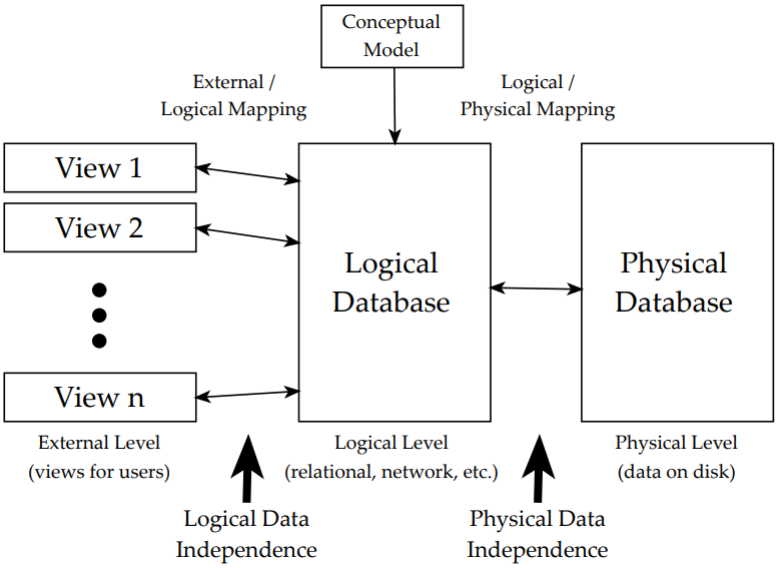
- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- query language access data
- operations such as add, delete, and replace

- **Transaction Management**: Transaction management is a feature that provides correct, concurrent access to the database, possibly by many users at the same time, ability to simultaneously manage large numbers of *transactions*
- **Access Control**: Access control is the ability to limit access to data by unauthorized users along with the capability to check the validity of the data. This is to protect against loss when database crashes and prevent unauthorized access to portions of the data
- **Resiliency**: Resiliency is the ability to recover from system failures without losing data, Ideally, should be able to recover from any type of failure, such as
  - sabotage
  - acts of God
  - hardware failure
  - software failure
  - etc.

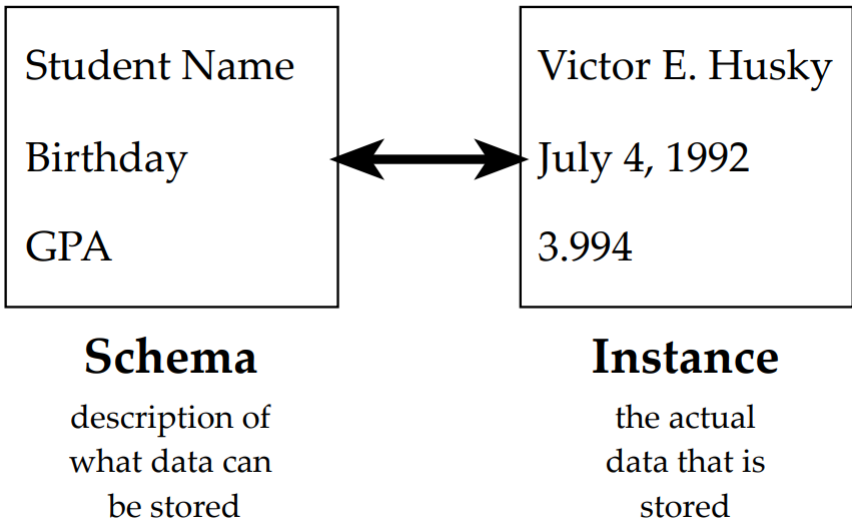
  Note: Obviously, some of these would require more than just software - offsite back-ups, etc
- **Use of Conceptual Modeling**:



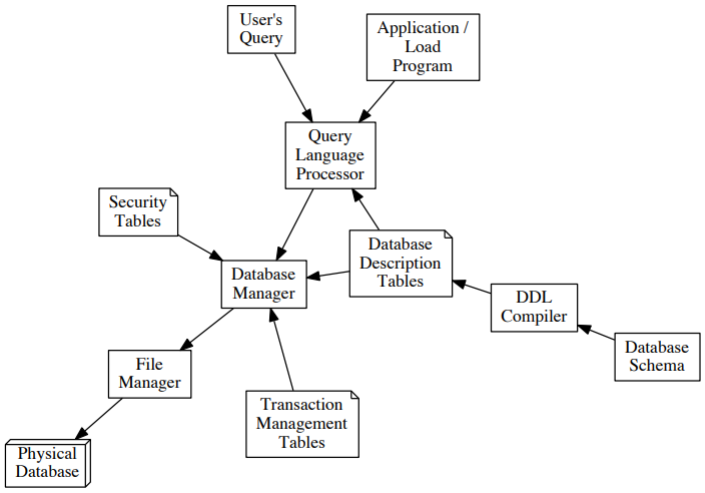
- **Leveled Architecture of a DBMS**:



- **External level**: a view or sub-schema, a portion of the logical database, may be in a higher level language
- **Logical Level**: abstraction of the real world as it pertains to the users of the database. DBMS provides a data definition language (DDL) to describe the logical schema in terms of a specific data model such as relational, hierarchical, network, inverted list, etc.
- **Physical Level**: The collection of files and indices, the collection of files and indices, this is the actual data
- **Instance**: An instance of the database is the actual contents of the data, it could be
  - the extension of the database
  - current state of the database
  - a snapshot of the data at a given point in time
- **Schema**: The schema of a database is the data about what the data represents. Such as,
  - plan of the database
  - logical plan
  - physical plan
  - the intention of the database
- **Schema vs Instance**:



- **Data Independence**: Data Independence is a property of an appropriately designed database system, it has to do with the mapping of logical level to physical level, and logical to external
  - **Physical data independence**: Physical schema can be changed without modifying logical schema
  - **Logical data independence**: logical schema can be changed without having to modify any of the external views
- **DCL (Control), DDL (Definition), DML (Manipulation)**: may be completely separate (example is IMS), may be intermixed (DB2), or may be a host language, for example an application program in which DML commands are embedded such as COBOL or PL/I
- **DBMS Components**:



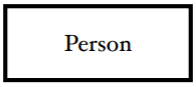
- **Overall DBMS Usage Scenario**: Database Administrator (DBA) define the conceptual, logical, and physical levels using DDL. DBMS software stores instances of these in schemas. User defines views (External Schema) in DDL. User accesses database using DML
- **Advantages of a Database**:
  - Controlled redundancy
  - Reduced inconsistency in the data
  - Shared access to data
  - Standards enforced
  - Security restrictions maintained
  - Integrity maintained more easily
  - Provides capability for backup and recovery
  - Permitting inferences and actions using rules
- **Disadvantages of a Database**:
  - Increased complexity needed to implement concurrency control
  - Increased complexity needed for centralized access control
  - Security needed to allow the sharing of data
  - Necessary redundancies can cause complexity when updating
- **Data vs Information**:
  - **Data**: Data refers to raw, unprocessed facts, figures, and details. It represents basic elements that have not been interpreted or given any meaning.
  - **Information**: Information is processed, organized, or structured data that is meaningful and useful. It is data that has been interpreted or analyzed to provide context, relevance, and purpose.

3.2 Conceptual Modeling and ER Diagrams

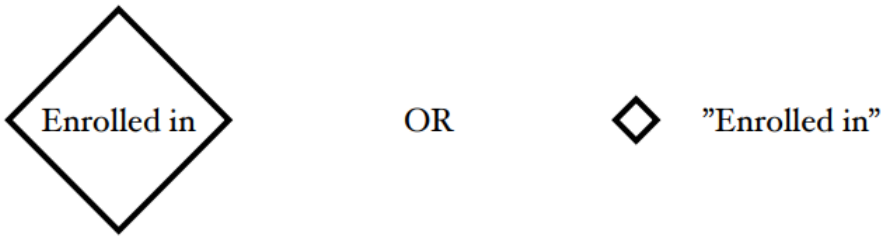
3.2.1 Definitions and theorems

- **Data Models**: A means of describing the structure of data, we typically have A set of operations that manipulate the data (for data models that are implemented)
- **Types of data models**:
  - Conceptual data model
  - Logical data models - relational, network, hierarchical, inverted list, or object-oriented
- **Conceptual Data Model**:
  - Shows the structure of the data including how things are related
  - Communication tool
  - Independent of commercial DBMSes
  - Relatively easy to learn and use
  - Helps show the semantics or meaning of the data
  - Graphical representation
  - Entity-Relationship Model is very common
- **Logical Data Models - Relational**: Data is stored in relations (tables). These tables have one value per cell. Based upon a mathematical model.
- **Logical Data Models - Network**: Data is stored in records (vertices) and associations between them (edges), Based upon a model called CODASYL
- **Logical Data Models - Hierarchical**: Data is stored in a tree structure with parent/child relationships
- **Logical Data Models - Inverted List**: Tabular representation of the data using indices to access the tables, Almost relational, but it allows for non-atomic data values [1], which are not allowed in relations
- **Logical Data Models - Object Oriented**: Data stored as objects which contain
  - Identifier
  - Name
  - Lifetime
  - Structure
- **Entity-Relationship Model**: Meant to be simple and easy to read. Should be able to convey the design both to database designers and unsophisticated users
- **Entities**: Principle objects about which information is kept - These are the *things* we store data about. If you look at the ER Diagram like a spoken language, the entities are nouns - Person, place, thing, event. When drawn on the ER diagram, entities are shown as rectangles with the name of the entity inside.

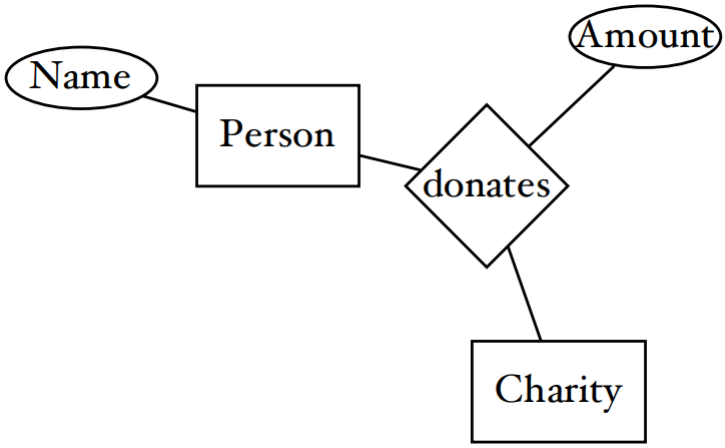
[1] "Non-atomic data values" refer to data structures or values that are composed of multiple components, as opposed to atomic data values, which are indivisible and represent a single value.



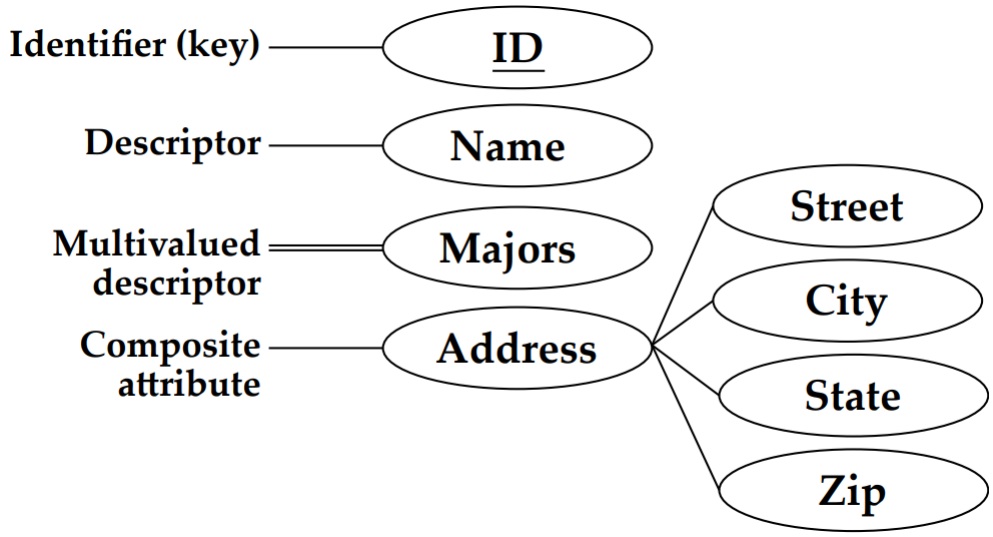
- **Relationships**: Relationships connect one or more entities together to show an association. A relationship *cannot* exist without at least one associated entity. Graphically represented as a diamond with the name of the relationship inside, or just beside it



- **Attributes**: Characteristics of entities **OR** of relationships, Represent some small piece of associated data, Represented by either a rounded rectangle or an oval.

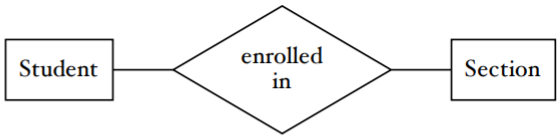


- **Attributes on Entities**: When an attribute is attached to an entity, it is expected to have a value for every instance of that entity, unless it is allowed to be null. For instance, in the diagram above, Name was an attribute of Person. Every person that we store data about will have a value for Name.
- **Attributes on Relationships**: When an attribute is attached to a relationship, it is only expected to have a value when the entities involved in the relationship come together in the appropriate way. In the diagram from before, the Amount attribute is attached to the donates relationship, which connects the Person and Charity entities. Amount will have one value for each time a Person donates to a Charity, denoting how much that person donated to the charity. It will not necessarily have a value for a given person, or a given charity. This can be referred to as the **intersection data**.
- **Types of attributes**:

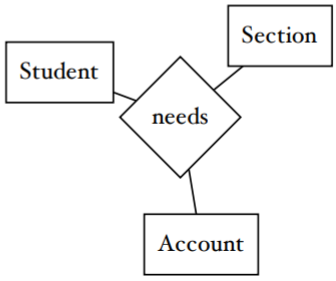


- **Degree of a Relationship**: The degree of a relationship is defined as how many entities it associates. If one entity is associated more than once (such as with a recursive relationship), then the degree counts each time it is referenced.

  - binary



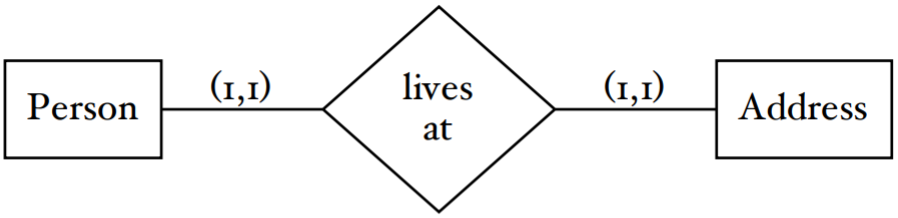
  - ternary



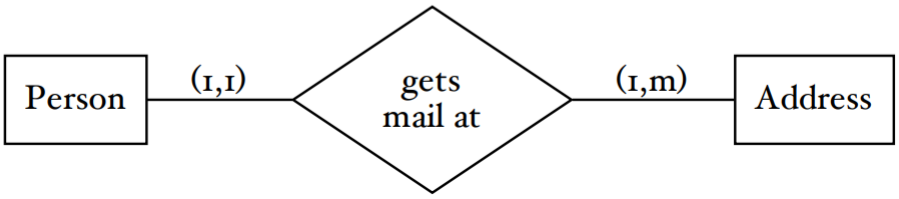
  Note: There is no limit to how many entities there can be in a relationship. After binary, and ternary, we start to call the relationships n-ary, where n is the degree

- **Connectivity of a Relationship**:
  - A constraint of the mapping of associated entities
  - Written as (minimum, maximum).
  - Minimum is usually zero or one.
  - Maximum is a number (commonly one) or can be a letter denoting many.
  - The actual number is called the cardinality.

- one-to-one

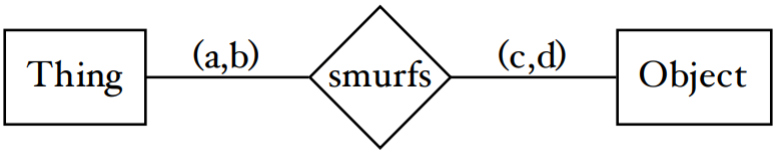


- one-to-many



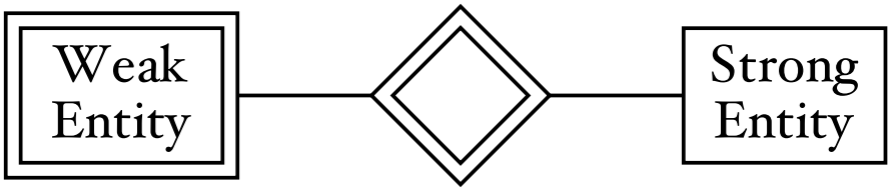
Together (from the image) both sides make up the connectivity, to refer to a single side, we use the term "cardinality", ie the cardinality of a person is (1,1). If we hold Address constant (We know a specific address and are therefore refering to that), how many persons may live at that address, in this case (1,1)

- **Attributes on Relationships (revisited)**: Must be on a many-to-many relationship. (1-many and 1-to-1 relationships should have the attribute on one of the entities involved. Someone needs to know all of the associated entities to access the attribute.
- **Reading Cardinalities**: For binary relationships:
  - For each Thing that smurfs, there are a minimum of c, and a maximum of d Objects.
  - For each Object that smurfs/is smurfed, there is a minimum of a and a maximum of b Things

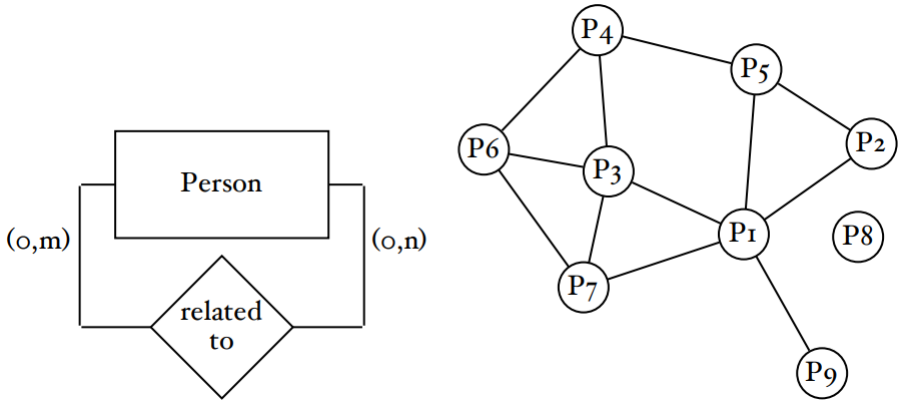


- **Weak Entities**: Sometimes you may run into an entity that depends upon another entity for its existence. The weak entity is a tool you can use to represent this.:w

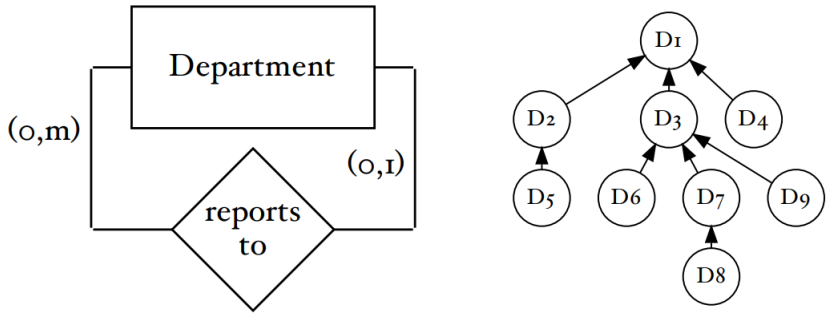
Weak entities are written like normal entities, except that they have a double rectangle outline. The relationship that connects the weak entity to the strong entity it depends upon will be written with a double diamond. This does not mean that the relationship is weak. It is just to indicate upon which entity the weak entity depends.



- **Recursive Relationships**: It is possible for an entity to have a relationship with itself. This is called a recursive relationship. It makes more sense if you think of entities as collections of objects of their appropriate type
- **Recursive Relationships - Many-To-Many**: A many-to-many recursive relationship means that the objects are arranged in a network structure, Notice that the minimum is 0 on both sides. This is important.



- **Recursive Relationships - One-To-Many**: A one-to-many recursive relationship means that the objects are arranged in a tree structure, Notice that the minimum is still 0 on both sides. This is important.

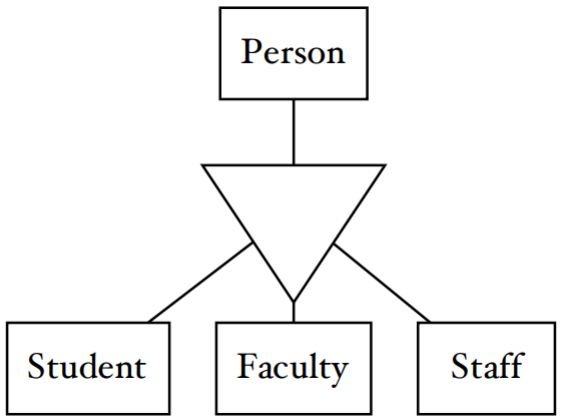


- **Entity or Attribute?**: Sometimes it isn't clear whether something should be an entity or an attribute of some other entity. Usually the decision will come down to how complicated it is to store the data, and how important it is. If it ends up being used in multiple places, it might be a clue that you should use an entity

- **Inheritance**: Two types of inheritance available
  - ”is a” inheritance. This shows that the subtype IS a member of the supertype.
  - ”is part of ” inheritance. This shows that the supertype contains, or is made up of members of the subtypes.

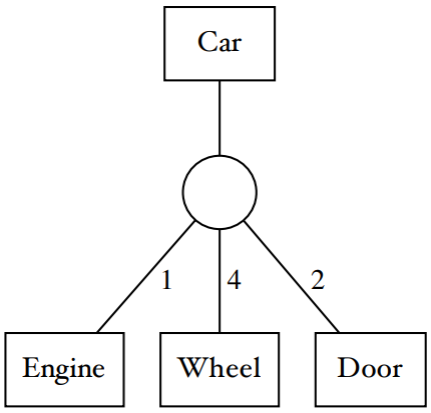
  All attributes of the supertype entity are inherited by the subtype entities. The identifier of the subtypes will be the same as the supertype

- **IS A Inheritance**: This type of inheritance happens when you have a supertype and one or more subtypes that are members of the supertype. Denoted by an upside-down triangle, with the supertype on top, and the subtypes coming out the bottom.



- **Defining IS-A inheritance**: There are two things you need to choose when using IS-A inheritance:
  - **Generalization (no) vs. specialization (yes)**: can the supertype occur without being a member of the specified subtypes?
  - **Overlapped (yes) vs. disjoint subtypes (no)**: is it possible for a single occurrence of the supertype to be a member of more than one subtype?

  They are mutually exclusive so you need to pick one of each, ie. GO, GD, SO, SD
- **IS-A inheritance - Generalization**: Supertype is the union of all of the subtypes, This means that an instance of the supertype CANNOT EXIST without belonging to at least one subtype.
- **IS-A inheritance - Specialization**: The subtype entities specialize the supertype, This means that an instance of the supertype CAN exist without being related to any of the subtypes
- **IS-A inheritance - Overlapping Subtypes**: It is possible for an instance of the supertype to be related to more than one of the subtypes
- **IS-A inheritance - Disjoint Subtypes**: the subtype entities are mutually exclusive, it is not possible for an instance of the supertype to be related to more than one subtype.
- **IS-PART-OF Inheritance**: ”Is part of ” inheritance indicates that the supertype is constructed from instances of the subtypes. It is shown on an ER diagram as a circle, with the supertype on the top, and subtypes on the bottom.

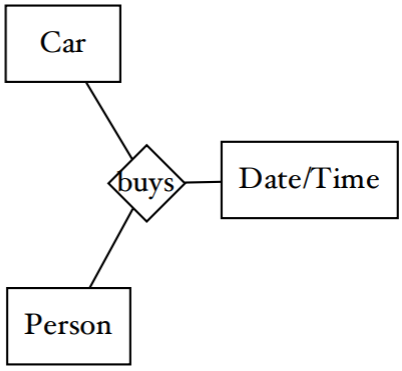


- **Warning about IS-PART-OF**: The IS PART OF inheritance operator does have its uses, but it is not very commonly used, If you see something involving a certain number of things being present, there are several possibilities
  - Sometimes a number is specified that isn't actually important for what we are modeling. This won't even be represented on an ER Diagram. This is the case when changing the number wouldn't have any effect on the necessary structure of a database.
  - If you need a certain number of items for a relationship to hold, you should explore using the connectivity of the relationship to express that.
  - Finally, this IS PART OF inheritance might be useful. It is almost never necessary, however.
- **Are you actually representing what you want to?**: Let's say you're running a business selling used cars. A simple ER diagram for the sales might look like the following:

The resulting database would have one entry for each time a specific person buys a specific car. If the same person buys the same car more than once (obviously selling it to someone else at some point), this model would no longer be appropriate.

The resulting database would have one entry for each time a specific person buys a specific car. If the same person buys the same car more than once (obviously selling it to someone else at some point), this model would no longer be appropriate.

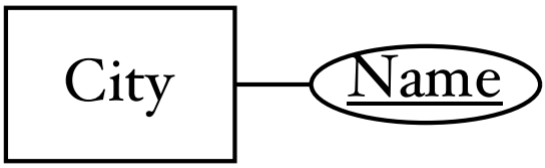
Adding a new entity to the relationship for the date/time of the purchase can fix this problem.



Notice that the connectivities can change when you add new entities to the relationship.

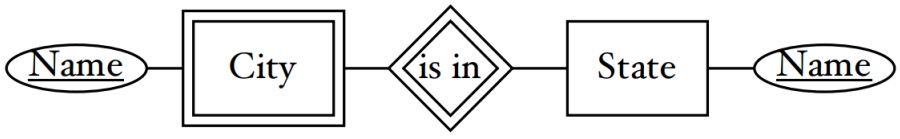
- **Weak Entities - Introduction**: So far, all of the entities we have used have been things that stand on their own. There are some situations where we are modeling an object for which we certainly need to store data, but the items exist only in the context of some other entity. Many of these examples can occur

  One example of a time that an entity depends on another would be the idea of a city. Within a state, we can generally be assured that cities will have unique names. If we were working only at that level, the City could be an entity as we saw above. A good identifier for it would be the name of the city, so we would see the following:



In some situations, this would be valid. The Name attribute can serve, in those circumstances, as an appropriate identifier.

To indicate this sort of dependency, we can make the dependent entity a "weak" entity. This is drawn with a double-edged rectangle, shown below.



Notice that the City entity is now drawn as a weak entity, with a double border. The relationship between the weak entity and the strong entity is also drawn with a double border. The relationship is not weak, per se, but it is used to indicate which strong entity the weak entity depends upon.

- **Discriminant (partial key)**: The discriminant, also known as the partial key, is an attribute (or a set of attributes) within the weak entity that can uniquely identify the weak entity, but only in combination with the primary key of the strong entity it is associated with. In other words, the discriminant helps to distinguish instances of the weak entity when they are tied to a particular instance of the strong entity.
- **Schema**: In databases, a schema is the structural definition of how data is organized in a database. It outlines the way data is stored

3.3 The Relational Model

- **Basic Structure**:
  - **Relations**: In the relational data model, our database is made up of one or more **relations** (tables). Each relation should have a unique name.
  - **Schema**: The schema of a relation is written as **Relation_Name**$(A_1, A_2, ..., A_n)$, Where $A_1, A_2, ..., A_n$ are placeholders for the attribute names
  - **Column headers (attributes)**: The attributes becomes the column headers of the relation.
  - **Instance data, tuples**: When there is instance data, it will come in the form of **tuples** (rows), which have a value for each attribute, as shown below

  Note: No field may contain than one value.

Relation_Name				
A_1	A_2	A_3	...	A_n
x_1	x_2	x_3	...	x_n
y_1	y_2	y_3	...	y_n
...	...	...		...

- **The domain of an attribute**: Each attribute becomes a column heading

  Each attribute (column) also has an associated **domain**. The domain of an attribute is the set of all valid values for it. The domain may be looked at as a data type, but may have additional constraints.

- **The domain of a set of attributes**: The domain of a set of attributes is the set of all possible combinations of values for the attributes in the set.
- **Tuples (Rows)**: A tuple is a special type of (mathematical) set containing values for each attribute within the relation. Tuples are shown as rows in the table, with the value for each attribute under the appropriate column
- **Atomic tuples**: The values are required to be atomic; there can be only one value per tuple per attribute
- **Relation vs relationship**: Though they have similar names, A relation (table) and a relationship (from an ER diagram) **ARE NOT** the same thing.
  - **Degree of relation**: The degree of a relation is the number of attributes present.
  - **Cardinality of a Relation**: The cardinality of a relation is the number of tuples present.
- **Keys**: Speaking generally, the purpose of a key is to uniquely identify a tuple in some relation.
  - **Super keys**: A super key within a relation is an attribute or set of attributes whose values can uniquely identify any tuple within that relation
  - **The trivial key**: Every relation has at least one - the set of all attributes in the relation
  - **Candidate Keys**: A candidate key is a minimal super key – the minimum set of attributes necessary to uniquely identify a tuple within the relation

  - **Primary Key**: The primary key for a relation is chosen by the database designer from among the relation's candidate keys. It becomes the "official" key that is used to reference tuples within the relation. There can be only one
  - **Prime, non-prime attributes**: Once a primary key is chosen, each of the attributes in the relation will be either **prime** or **non-prime** with respect to the relation. A prime attribute is one of the attributes that can be found in any of the candidate keys. A non-prime attribute is one of the attributes not found in any of the candidate keys

    Once a primary key is chosen for it, the schema of a relation is written with the primary key's attributes underlined
  - **Foreign Keys**: A foreign key is a tool used to link relations within a database. Since every relation has a primary key that uniquely identifies each tuple, the values of those key attributes can be used from another relation to reference individual tuples.

    The relation whose primary key is being used is the **home relation**

- **Order Independence**: In relations, the order things appear doesn't matter. There are ways to force them to sort later when we're working with SQL, but the relation itself has no order for either rows or attributes...
- **Order Independence - Attributes**: It doesn't matter what order the attributes appear in, if two relational schemas have the same name, the same attributes, and the same primary key, then they are equivalent.
- **Order Independence - Tuples**: Tuples are stored unordered. If you need to have them appear in some order later, you will be able to sort based on the values inside of them using SQL.
- **Constraints**: Constraints are limits imposed on the domains of various attributes. These can come from the system your database is modeling
- **Entity Integrity Constraint**: The entity integrity constraint applies to all relations. It states that no tuple may exist within a relation that has null value for any of attributes that make up the primary key. This is a consequence of the primary key being a candidate key, which is minimal and cannot do its job with any less data.
- **Referential Integrity Constraint**: It constrains the values of foreign keys in relations to values that actually exist as primary keys for tuples within the home relation. If the foreign key is otherwise allowed to be NULL, then that is also an acceptable value.

3.4 Relational Model Normalization

- **Designing Relational Databases**: There are a large number of possible ways to represent each problem with using relations. Some choices will perform better than others for various reasons. The option chosen should be the best one, but how do we know which one that is?

  We should study:

  - Problems that can come up
  - How to avoid them
  - Desirable properties
  - How to guarantee them

- **Basic Example**: If our database is a single relation with schema **SP**(SuppName, SuppAddr, Item, Price) with the instance data:

SuppName	**SuppAddr**	**Item**	**Price**
John	10 Main	Apple	$2.00
John	10 Main	Orange	$2.50
Jane	20 State	Grape	$1.25
Jane	20 State	Apple	$2.25
Frank	30 Elm	Mango	$6.00

  There are some common things that we might want to do that would cause issues

  - **Insertion Anomaly**: Let's say we want to add a new vendor, "Sally", and store her address, "40 Pine", but she is not selling anything yet. Can this be inserted into the relation SP?

    NO. The primary key is (SuppName, Item), but we only have SuppName. The entity integrity constraint is violated if we try to insert the data as a tuple in this relation. It cannot fit. We call this an insertion anomaly.

  - **Deletion Anomaly**: This time, let's say that Frank no longer sells Mango. We want to take that out of the database so nobody can order a mango that is not available. Can this tuple remain in the relation with the Mango information removed?

    NO. The primary key is (SuppName, Item), and the Item is going away. The entity integrity constraint is violated if we remove the data from the tuple in this relation. We can either keep the whole tuple, advertising fake mango, or delete the whole tuple and lose the information on Frank, which doesn't exist in any other tuples. We call this a deletion anomaly.

  - **Update Anomaly**: Next, let's say that John is moving to a different address. We would have to change it once for every item John is selling. This isn't a big deal with only two items, but as John's list of supplied items grows, so does the amount of database work that needs to be done every time he moves. If any of the SuppAddr values for John don't agree, then it may not be clear which is the right address for John. This is an update anomaly.

  - **Redundancy**: Redundancy is when values are repeated.

It can be

  - **Good:** If you have an off-site backup of your entire database, the redundancy is useful, and can be used to restore in case of a failure.
  - **Bad:** Redundancy on the same physical device is unnecessary. It wastes space and comes with the potential for update anomalies.

- **Note:** The good redundancy is something the DBA/IT department should handle. When we talk about redundancy in the design of our database, we will be talking about the bad kind.

- **Anomolies summarized**:

  Insertion anomalies:

  - When a piece of data cannot be inserted because it violates some constraint of the relation.
  - Usually this is the entity integrity constraint being violated, but not always. See the Sally example

  Deletion anomalies:

  - When deleting some piece of data, a deletion anomaly is when more data is lost than intended
  - Usually this is caused when the data removed is part of the primary key, which would cause a violation of the entity integrity constraint. See the Frank example

  Update anomalies:

  - When updating a single value requires changes to multiple tuples, this is an update anomaly. See the John example.
  - This is caused by unnecessary redundancies in the data.
  - These cause inefficiency, and potential inconsistencies.

- **Decomposition**: There is no rule that says that a relational database must be made up of a single relation. The way we will solve these anomalies is to add new relations to our database and change the old ones. This is called decomposition.

  Using the example from above, we can remove the anomalies by decomposing the database into two relations.

  SP(SuppName, Item, Price)

SuppName	**Item**	**Price**
John	Apple	$2.00
John	Orange	$2.50
Jane	Grape	$1.25
Jane	Apple	$2.25

  S(SuppName, SuppAddr)

SuppName	SuppAddr
John	10 Main
Jane	20 State
Frank	30 Elm
Sally	40 Pine

- **When to decompose**: One way of designing a database could be to list all of the possible anomalies and then decompose to fix each of them. The problem with this is that any anomalies you don't see coming will not be fixed.

  We will look at a systematic method of identifying the potential for anomalies. This method is called normalization

- **Normalization**: Normalization involves making sure that each of your relations follows certain rules. Depending on which rules are followed, each of the relations in your database will be in one or more normal forms. These rules are based on functional dependencies

- **Functional Dependencies**: A functional dependency is a statement about which attributes can be inferred from other attributes. If we take X and Y as sets of attributes, we can write:

  $$X \to Y.$$

  Which means, if, whenever unique values for **all** of the attributes in X are known, unique values for **each** of the attributes of Y are guaranteed to be possible to look up or to infer using those values.

  This is read either as:

  - X functionally determines Y
  - Y is functionally dependent upon X

- **Functional Dependencies: Real-life Examples**:

  - **ZID** $\to$ **StudentFirstName, StudentLastName, Birthday**: If I identify a student using their ZID, that student has one first name, last name, and birthday
  - **StudentFirstName** $\not\to$**ZID**: The first name is not enough to determine a single ZID, as there are multiple students with the same first name
  - **ZID, CourseID, Semester** $\to$**Grade**: If I know which student, which course, and which semester, I can find a single grade

- **Functional Dependencies: Keep In Mind**: FDs are constraints present within the operational data your database models. They don't necessarily describe how things work in the real world, but they do have to accurately describe any data you will store in your database

  FDs **must** hold for all possible data values. Attempts to add data that does not obey the FDs will result in anomalies.

FDs can be enforced during insertion if the database is set up properly

- **Armstrong's Axioms**: Armstrong's Axioms are a set of rules for operations that are permissible when manipulating functional dependencies
  - **Reflexivity**: If $Y \subseteq X$, then $X \to Y$
  - **Augmentation**: If $X \to Y$, the $XZ \to YZ$ for any Z
  - **Transitivity**: If $X \to Y$ and $Y \to Z$, then $X \to Z$
  - **Decomposition**: If $X \to YZ$, then $X \to Y$ and $X \to Z$
  - **Composition**: If $X \to Y$ and $A \to B$, then $XA \to YB$
  - **Union (Notation)**: If $X \to Y$ and $Y \to Z$, then $X \to YZ$
  - **Pseudo-transitivity**: If $X \to Y$ and $YZ \to W$, then $XZ \to W$
  - **Self-determination**: $I \to I$ for any I
- **Functional Dependencies: Keys Revisited**: Now that we know about functional dependencies (FDs), we can assert:

The attributes of a superkey must functionally determine all of the attributes of the relation.

Candidate keys and primary keys are superkeys, so this is true of them as well, and they also satisfy additional requirements.

Example: As an example, say we have the relation **R**(a,b,c,d,e,f). We can say

$$\begin{aligned} & a \to a, b, c, d, e, f \\ \implies & a \to b, c, d, e, f. \end{aligned}$$

- **First Normal Form (1NF)**: You should recall from the introduction to relations that all of the values in a tuple with a relation must be atomic. This means that there is a maximum of one value per attribute per tuple

The requirement for a relation to be in First Normal Form (1NF) is this same requirement that all of the values must be atomic

What this usually looks like is a table with mutltiple values in a single cell. A non-1NF relation would not even technically count as a relation.

Given the table:

X	Y	Z
x1	y1	z1 z2 z3
x2	y2	z4
x3	y2	z5

It looks like X would have been the primary key, but it's not doing its job of uniquely determining Z, which is showing as a repeating group so X can't be a key

What usually causes this is not having the correct primary key

The table above has the following function dependencies:

$$X \rightarrow Y$$
$$X, Z \rightarrow Z.$$

To move this pseudo-relation into an actual relation that doesn't violate 1NF, we need to choose a real primary key that meets the requirements. We do that using the FDs. In this case, (X , Z) works.

Changing the primary key yields: - $R(X, Y , Z))$

X	Y	Z
x1	y1	z1
x1	y1	z2
x1	y1	z3
x2	y2	z4
x3	y2	z5

- **Pseudo-relation**: The notation for a "pseudo-relation" like the one above would be to use inner parenthesis on the repeating group, ie. $\mathbf{R}(X, Y , (Z))$
- **Second Normal Form (2NF)**: Second Normal Form (2NF) has to do with the concept of full dependence.

  Given two sets of attributes, X and Y , we can say that Y is fully dependent on X, if (and only if)

  $$X \rightarrow Y.$$

  And no subset of X determines Y

  A relation is in 2NF if:

  – It already meets the requirements of 1NF, and
  – All non-prime attributes of the relation are fully dependent upon the entire primary key

  What breaks 2NF is when attributes are dependent upon only part of the primary key. To fix 2NF violations once we're in 1NF, decomposition is the solution.

  Example: Going back to our earlier example: **EmpProj**(EmpID, Project, Supv, Dept, Case)

EmpID	Project	Supv	Dept	Case
e1	p1	s1	d1	c1
e2	p2	s2	d2	c2
e1	p3	s1	d1	c3
e3	p3	s1	d1	c3

Functional Dependencies:

EmpID, Project → Supv, Dept, Case
EmpID → Supv, Dept
Supv → Dept

A quick glance confirms all of the values are atomic, so 1NF is confirmed.

There is a 2NF violation caused by (EmpID → Supv, Dept) because the primary key is (EmpID, Project), but only EmpID is on the LHS.

Observing the instance data, you should easily see that the attributes of the RHS cause update anomalies in this table. We also can't insert a new employee with no project (insertion anomaly), and removing e2 from p2 would remove e2 from the database entirely (deletion anomaly). These are symptoms of the 2NF violation.

Decomposition Pattern: There is a pattern to follow for the decomposition. Start with the original relation, and the FD that causes the violation.

EmpProj(EmpID, Project, Supv, Dept, Case)
EmpID → Supv, Dept.

The attributes on the RHS of the FD are removed from the original relation and placed into a newly created relation that has the FD's LHS as its primary key. A foreign key links the attribute from the LHS in the original table (the LHS is not removed) to the corresponding tuple in the new table, where it is the primary key.

EmpProj(EmpID, Project, Case)
Employee(EmpID, Supv, Dept).

Instance of 2NF Version:

EmpProj(EmpID, Project, Case)

EmpID	Project	Case
e1	p1	c1
e2	p2	c2
e1	p3	c3
e3	p3	c3

Employee (EmpID, Supv, Dept

EmpID	Supv	Dept
e1	s1	d1
e2	s2	d2
e3	s1	d1

- **Third Normal Form (3NF)**: To be in Third Normal Form (3NF), a relation must

  1. already qualify to be in 2NF
  2. none of the non-prime attributes may be transitively dependent upon the primary key

  By definition, all non-prime attribute are functionally dependent upon the primary key. What makes a transitive dependency is that there is also some non-prime attribute (which also depends on the key) that also functionally determines the attribute.

  To quickly identify the transitive dependencies from the list of FDs, look on the LHS for attributes that are non-prime in the context of the current relation.

  Example:

  EmpProj(EmpID, Project, Case)
  Employee (EmpID, Supv, Dept
  EmpID, Project $\rightarrow$ Supv, Dept, Case
  EmpID $\rightarrow$ Supv, Dept
  Supv $\rightarrow$ Dept.

  In this case, the FD that causes our relations to violate 3NF is (Supv $\rightarrow$ Dept), and the violation happens in the Employee relation. If you refer back to the instance data of that in the 2NF solution, you can see that the violation can cause anomalies, so we want to fix it.

  Just like 2NF, we fix 3NF by decomposing using the FD that causes the violation to occur. **AT NO POINT DO WE CHANGE THE FDs**

  Decomposition Pattern: We follow the same pattern for decomposition in 3NF as we did in 2NF. Start with the relation that has the violation, and the FD that causes the violation to occur.

  Employee (EmpID, Supv, Dept)
  Supv $\rightarrow$ Dept.

  The attributes on the RHS of the FD are removed from the violating relation and placed into a newly created relation that has the FD's LHS as its primary key. A foreign key links the attribute from the LHS in the original table (the LHS is not removed) to the corresponding tuple in the new table, where it is the primary key.

  Employee(EmpID, Supv)
  SupvDept(Supv, Dept).

  The RHS (Dept) that was a violation when it was in Employee because the LHS (Supv) was non-prime is no longer there to cause the problem. It is in the new relation where the LHS (Supv) is the primary key, and therefore we don't have a transitive dependency. These two relations no longer have the 3NF violation.

- **Summary of the normalization forms**:

First Normal Form (1NF):

- No repeating groups. All values are atomic.
- A primary key must have been chosen, and this primary key must be a proper superkey – it needs to be able to functionally determine every attribute in the relation.

1NF violations are fixed by choosing an appropriate primary key

Second Normal Form (2NF) - To be in Second Normal Form, a relation must conform to 1NF and:

- All of the non-prime attributes must be fully dependent upon the entire primary key.
- No non-prime attribute may be functionally determined by any subset of the primary key.
- No partial key dependencies

2NF violations are fixed by decomposition.

Third Normal Form (3NF) - To be in Third Normal Form, a relation must conform to 2NF and:

- There may be no transitive dependencies.
- No non-prime attribute may functionally determine another non-prime attribute.

3NF violations are fixed by decomposition.

3.5 ERD to Relations (Conceptual to logical)

- **The basic outline (steps)**

  1. Handle all of the entities
  2. Handle all of the relationships

- **Entity handling**: We will start with entities, because they can stand on their own, unlike relationships or attributes. In general, each entity will get its own relation. The attributes of the entity will become attributes in the schema of the relation created. There are some special cases to take into account, which will be handled from most independent to least, so:

  a. Strong (non-weak) entities that are not subtypes
  b. Strong.(non-weak) entities that are subtypes
  c. Weak entities

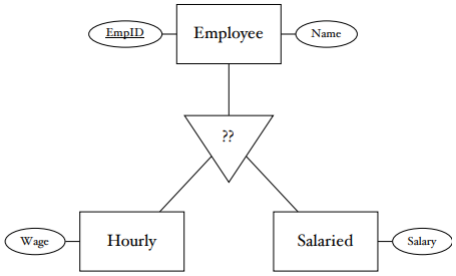
- **Entities like date**: there is no reason to make a relation for a "Date" entity or similar. The single value for the date is enough to determine it, and any other data associated with it is generally happening through a relationship anyway. Think about what data would go into such a table and how little use there would be for storing it separately.

- **Handling strong, non subtype entities**: Make a new relation, whose name will be the same as the name of the entity The primary key of the relation will be all of the identifier attributes, taken together All attributes of the entity become attributes of the relation Every instance of the entity gets the relevant values put into a new tuple in the relation

  Example: Suppose we had an entity A with attributes $\underline{ID}$, and other:

  Then, we would make a relation $A(\underline{ID}, other)$

- **Handling strong, subtype entities**: Suppose



  Employee is a supertype (not subtype) so it gets handled in the previous step

  Employee(<u>EmpId</u>, name)

  Hourly and **Salaried** are each strong, but they are subtypes (each is a type of Employee), so they are handled here

  This type of inheritance means that the subtypes are types of the supertype, so they are identified by **Employee's** EmpID

There are two methods of handling these.

1. **Big table**: The first method involves putting the attributes of the subtypes into the relation made for the supertype. So, the original relation:

   Employee(EmpID, Name)

   Would become something like:

   Employee(EmpID, Name,Wage, Salary)

   but it would need to be modified to indicate which subtypes a given employee belongs to. Let's examine that on the next page.

   The big table method needs a way to know which of the subtypes the current instance of the supertype belongs to, which is handled differently depending on the IS-A's configuration.

   For **disjoint subtypes**, where an instance of the supertype can only be one of the subtypes at a timei, we can add an attribute, EmpType that has a value indicating which type this employee is.:

   Employee(EmpID, Name, EmpType,Wage, Salary)

   For generalization, EmpType would not allow NULL. For specialization, it would be allowed.

   For **overlapping subtypes**, it is possible to be more than one at a time, so we need an individual true/false answer for each type:

   Employee(EmpID, Name, IsHourly,Wage, IsSalaried, Salary)

   In this case, nothing about the schema would indicate generalization vs. specialization

2. **New relation**: Method 2 involves creating a new relation for the subtype entity The name of new relation would be the same as the name of the entity.

   The primary key of the new relation would be the same as the primary key for the supertype's relation.

   The primary key is also a foreign key to the existing table.

   An instance of the supertype entity will only have a tuple in the subtype relation if it is a member of that subtype, so we will not need any extra attributes like we did in method 1.

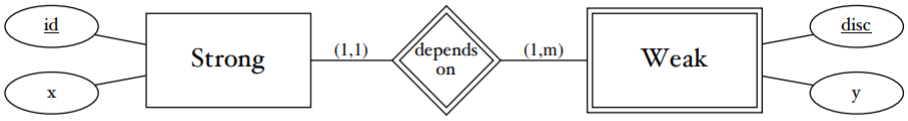
   The foreign key can be used to look up any of the attributes that are being inherited from the supertype

   Thus, we would have

   Employee(EmpID, Name
   Hourly(EmpID†,Wage)
   Salaried(EmpID†, Salary).

Note: The (†) (dagger symbol) will be used in these slides to indicate that the attribute is part of a foreign key (and, in this example, the whole thing).

- **Handling weak entities**: Suppose



Example with weak entity

The strong entity would already have a relation.

Strong(id, x)

The weak entity gets its own relation. The primary key will be the concatenation of the weak entity's discriminator with the strong entity's identifier. The other attributes of the entity are brought in as non-prime attributes.

Weak(id†, disc, y)

The id portion is a foreign key to the Strong relation

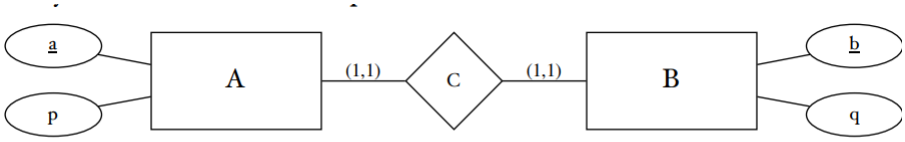
- **Entities: Functional Dependencies**: The only functional dependencies introduced by the entities of an ER diagram are the ones introduced when the identifiers become primary keys. Remember that a primary key has to functionally determine all of the other attributes in a relation
- **Handle relationships**: The relationships will be handled in order from lowest degree to highest degree, and within that, from simplest cardinality (one-to-one) to more complicated cardinalities (many-to-many, etc.).

  The purpose of a relationship is to form connections between entities. We know that we are using relations to represent our entities, so we will need to use a tool that can link those relations to each other.

  The tool best suited to linking tuples from relations together is the foreign key.

  Every relationship we model in the relational model will have one or more foreign key involved. Where we put these foreign keys will depend on the cardinality, and the decisions are motivated by the normal forms we discussed.

  1. **Binary one-to-one Relationships**: In a binary relationship, we will already have made a relation for each of the entities involved.



Here, C is a binary, one-to-one relationship between A and B.

A(a,p) *and* **B**(b, q)

Since each instance of B will have one of A, and each instance of A will have one of B through C, we can represent this one-to-one relationship by putting a new foreign key into the entity for either side. Choose either:

$$\mathbf{A}(\underline{\text{a}},\text{p},\text{b}\dagger) \qquad \textit{or} \qquad \mathbf{B}(\underline{\text{b}},\ \text{q},\ \text{a}\dagger)$$

The relationship implies the functional dependencies:

$$a \to b$$
$$b \to a.$$

2. **Binary one-to-many Relationships**: In a binary relationship, we will already have made a relation for each of the entities involved.

$$\mathbf{A}(\underline{\text{a}},\text{p}) \qquad \textit{and} \qquad \mathbf{B}(\underline{\text{b}},\ \text{q})$$

For this one-to-many relationship, there can be many instances of B for each of A, so we can't have the foreign key in the A table (wouldn't be atomic, so 1NF would be violated). We still do have the option of putting a foreign key in the B table pointing to the corresponding A, so our only option is:

$$\mathbf{B}(\underline{\text{b}},\text{q},\text{a}\dagger)$$

The only FD is

$$b \to a.$$

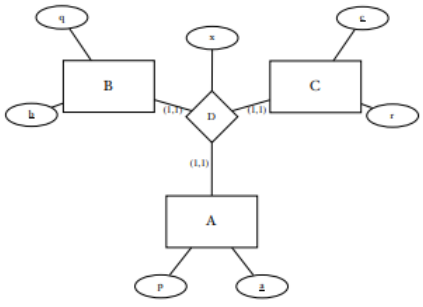
3. **Binary many-to-many Relationships**: In a binary relationship, we will already have made a relation for each of the entities involved.

$$\mathbf{A}(\underline{\text{a}},\text{p}) \qquad \textit{and} \qquad \mathbf{B}(\underline{\text{b}},\ \text{q})$$

There are no new functional dependencies introduced by the relationship, and putting a foreign key into either relation would not be atomic (1NF violation). The many-to-many relationship requires a new relation. Its foreign key will be the concatenation of the primary keys of each of the entity relations, which will be used as foreign keys to the corresponding tables. Any intersection data is put into this new relation as a non-prime attribute.

$$\mathbf{C}(\underline{\text{a}}\dagger,\ \underline{\text{b}}\dagger,\ \text{x})$$

4. **Relationships Greater than Binary: one-to-one-to-one**:



So we have

$$\mathbf{A}(\underline{a},\ p) \text{ and } \mathbf{B}(\underline{b},\ q) \text{ and } \mathbf{C}(\underline{c},r$$

Each of the "one legs" represents a functional dependency, and each of them gives us a potential relation to choose from for our relation.

Note: If we have say only two ones, like a one to one to many relationship, we

Functional Dependency	**Potential Relation for D**
$a, b \to c$	$\mathbf{D}\left(a^{\dagger}, b^{\dagger}, c^{\dagger}, x\right)$
$b, c \to a$	$\mathbf{D}\left(a^{\dagger}, b, c^{\dagger}, x\right)$
$a, c \to b$	$\mathbf{D}\left(a^{\dagger}, b, c^{\dagger}, x\right)$

would just have less functional dependencies and therefore less options to choose from (see table above)

5. **Greater than Binary without any "ones"**: No functional dependencies are implied by this relationship. To stay in 3NF, the relation we must use is:

   $$\mathbf{D}(\underline{a}\dagger,\ \underline{b}\dagger,\ \underline{c}\dagger,\ x)$$

6. **Date entities (and similar)**: For relationships that have a "Date" entity (or the equivalent), recall that we did not make a relation for that entity. The only change necessary for your relationship involving that entity is that the date value is used instead of a foreign key, and that attribute will not be a foreign key, because the home relation would not exist

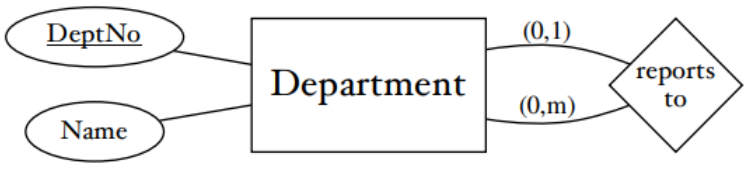
   As an example, if the C entity in the ternary relationship with no "ones" ER diagram were a Date entity, we would not create the C relation for it, and the relation to represent the relationship would be modified. Notice that the attribute is still part of the primary key, but no longer a foreign key.

   From: $\mathbf{D}(\underline{a}\dagger,\ \underline{b}\dagger,\ \underline{c}\dagger,\ x)$
   To: $\mathbf{D}(\underline{a}\dagger,\ \underline{b}\dagger,\ \underline{c},\ x)$

   .

7. **Recursive Relationships: one-to-many**: Recursive relationships will be handled as if they were normal relationships of the same degree and cardinality. The practical difference is that the entity that is linked multiple times will still only have one relation, so multiple foreign keys might go to the same table.

   Suppose:



   There should obviously only be one relation for the entity Department, because it is only a single entity.

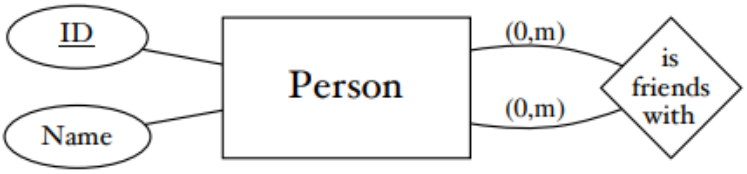
   Department(DeptNo, Name)

With a non-recursive one-to-many binary relationship, we would have put a foreign key to the relation for the one side into the relation on the one side. In this version, we only have one table, so the decision is easy. We will need to come up with another name for the foreign key, as we cannot have two attributes with the same name inside the same relation. Thus, we grow Department into the following:

Department(DeptNo, Name, ReportsToDept†)

Where the home relation for the new attribute, ReportsToDept, is that same relation, Department. The tuple of the department that the current department reports to will be have a DeptNo that equals the ReportsToDept in the current tuple. Alternatively, ReportsToDept can be NULL if the department does not report to another.

8. **Recursive Relationships, many-to-many**: Suppose



Like non-recursive many-to-many relationships, we will need to create a new relation. Unlike the non-recursive version, we only have one home relation for our two foreign keys. As in the one-to-many version, we will need to choose a new name for at least one copy of the foreign key, since they can't share the same name. The relation for our Person entity would be **Person**(ID, Name)

The new relation created to represent the relationship would be in the following form:

Friends(activeFriend†, passiveFriend†)

ActiveFriend and PassiveFriend are foreign keys to the tuple in Person with data for the person that is taking part in the relationship. This can be done in a directed or undirected way, and you probably want to put a comment somewhere about which way you intend to use it.

directed: (Person1, Person2) would not imply (Person2, Person1)

undirected: (Person1, Person2) does imply (Person2, Person1)

ID	Name
1	Regina George
2	Karen Smith
3	Cady Heron
4	Gretchen Wieners
5	Janis Ian

ActiveFriend	PassiveFriend
2	1
3	5
3	2
3	4
4	2
4	3
5	3