Mathematical Background 1

CSE 373
Data Structures

Basic Discrete Math Concepts

Sets
Cardinality
Relations
Cartesian Products
Functions
Properties of Functions

Sets

A *set* is a collection of distinct objects.
(An object is some identifiable person, place, thing, or idea).

The objects are usually represented by symbols.

The set consisting of Jupiter and Saturn:
{Jupiter, Saturn}

Sets (Continued)

The set consisting of the first 5 primes:
{ 2, 3, 5, 7, 11}

The set consisting of all strings made up of only a and b.
{ "", "a", "b", "aa", "ab", "ba", "bb", . . . }

(or, without the use of quotes...)
{ λ, a, b, aa, ab, ba, bb, . . . }

Sets (continued)

The objects that make up a set are called its *elements* or *members*.

If an object e is an elements of a set S, then we write $e \in S$

The empty set { } contains zero elements.

A set may contain other sets as members:

{ {a}, {b}, {a, b} } contains three (top-level) elements.

{ { } } contains one element.

Cardinality

A set may be finite or infinite.

The *cardinality* of a finite set is the number of (top-level) elements it contains.

Card({ a, b, c }) = 3

We sometimes use vertical bars:

| { a, b, c } | = 3

Binary Relations

Suppose S is a set. Let a ∈ S and b ∈ S.
Then (a, b) is an *ordered pair* of elements of S.
The set of all ordered pairs over S is:
{ (x, y) | x ∈ S, y ∈ S }
= the set of all ordered pairs (x, y) such that x is in S and y is in S.

Any set of ordered pairs over S is called a *binary relation* on S.

Binary Relations (cont)

Examples:

Let S = { a, b, c }

B_1 = { (a, b), (c, b), (c, c) } is a binary relation on S.

B_2 = { (a, a), (b, b), (c, c) } is a binary relation on S.
It happens to be reflexive.

B_3 = { } is a binary relation on S.

It happens to be empty.

Binary Relations (reflexivity)

A binary relation on S is *reflexive* provided that for every element in S, the pair of that element with itself is a pair in S.

S = { a, b, c }

R_4 = { (a, a), (a, b), (b, b), (b, c), (c, c) }
is reflexive.

R_5 = { (a, a), (a, b), (b, b) } is not reflexive, because c ∈ S but (c, c) ∉ R_5.

Binary Relations (symmetry)

A binary relation R on S is symmetric provided that any pair that occurs in R also occurs “reversed” in R.

S = { a, b, c }

R_6 = { (a, b), (b, a), (c, c) } is symmetric.

{ } is symmetric.

R_7 = { (a, b), (b, b), (c, c) } is not symmetric, because (a, b) ∈ R_7 but (b, a) ∉ R_7.

Binary Relations (transitivity)

A binary relation B on S is *transitive* provided that whenever there is a two-element “chain” in B there is also the corresponding “shortcut” in B.

B is transitive iff

(∀x ∈ S, ∀ y ∈ S, ∀ z ∈ S)

(x, y) ∈ B and (y, z) ∈ B → (x, z) ∈ B)

R_8 = { (a, b), (a, c), (b, c), (c, c) } is transitive.

R_9 = { (a, b), (b, a)} is not transitive, because (a, b) and (b, a) form a chain, but (a, a), the shortcut, is not present.

Cartesian Products

Let S_1 and S_2 be sets.
Then the *cartesian product* S_1 X S_2 is the set of all ordered pairs in which the first element is a member of S_1 and the second element is a member of S_2.

Example:
Let A = { a, b, c }, Let B = { 1, 2 } then

A X B = { (a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2) }

Cartesian Products (n-way)

The n-way *cartesian product* S_1 X S_2 X ... X S_n is the set of all ordered n-tuples in which the i^{th} element is an element of S_i.

S_1 X S_2 X ... X S_n =
{ (s_1, s_2, ..., s_n) | $s_1 \in S_1$, $s_2 \in S_2$, ..., $s_2 \in S_2$ }

Functions

Let S_1 and S_2 be sets.

Let f be a subset of S_1 X S_2.

(f is a binary relation on S_1 and S_2)

If for each x in S_1 there is precisely one y in S_2 such that (x, y) $\in$ f, then f is a *function from* S_1 *to* S_2. We also say f is a *function on* S1.

S_1 is called the *domain* of f and S_2 is called the *range* of f.

Functions (Examples)

Let S_1 = { a, b, c} and S_2 = { 1, 2}.
Let f_1 = { (a, 1), (b, 1), (c, 2) }
f_1 is a function on S_1.

Let f_2 = { (a, 1), (b, 1), (b, 2), (c, 1) }
f_2 is not a function.

Let f_3 = { (a, 1), (b, 2)}
f_3 is *not* a function on S_1.
But it is a *partial function* on S_1.
It's actually a function on { a, b }.

Properties of Functions

Let f be a function from S_1 to S_2.
If every element of S_2 appears as the second element of some ordered pair in f, then f is said to be "onto". (It's also said to be a *surjection*.)

With S_1 = { a, b, c} and S_2 = { 1, 2}.
and f_1 = { (a, 1), (b, 1), (c, 2) },
f_1 is onto.

Let f_4 = { (a, 1), (b, 1), (c, 1) } with the same domain and range. f_4 is *not* onto.

Properties of Functions (cont)

Let f be a function from S_1 to S_2.
If no two elements of S_1 are paired with the same element of S_2 then f is said to be "one-to-one". (It's also said to be a *injection*.)

With S_1 = { a, b, c} and S_2 = { 1, 2}.
and f_1 = { (a, 1), (b, 1), (c, 2) },
f_1 is not one-to-one, since a and b are both paired with 1.

Let f_5 = { (a, 1), (b, 2)} with domain { a, b}. f_5 is one-to-one.

Properties of Functions

Let S_1 = { a, b, c} and S_2 = { 1, 2}.
Let f_1 = { (a, 1), (b, 1), (c, 2) }
f_1 is a function on S_1.

Let f_2 = { (a, 1), (b, 1), (b, 2), (c, 1) }
f_2 is not a function.

Let f_3 = { (a, 1), (b, 2)}
f_3 is *not* a function on S_1.
But it is a *partial function* on S_1.
It's actually a function on { a, b }.

Abstract Data Types

- **Motivation**
- **Abstract Data Types**
- **Example**
- **Using math. functions to describe an ADT's operations.**

Motivation for ADTs

To organize some data, we need to say what general form it has, and what we expect to do with it.

Object-oriented programming provides one way of organizing data: using classes, with their data members and methods.

It is often helpful to specify our data without having to choose the particular data structure yet.

Abstract Data Type (Def.)

An *abstract data type* consists of two parts:
  a description of the general form of some data.
  a set of methods.

The data is usually a set of elements, but may also include relationships among the elements.

{ 2, 3, 5, 7, 11}
[2 < 3, 3 < 5, 5 < 7, 7 < 11, etc.]

ADT Methods

Each method in an ADT names and specifies an operation. The operation can be described by a function. Normally, an instance of the ADT data is one of the arguments to the function.

Examples of methods:
INSERT(x)
MEMBER(x)
SMALLEST( )
CREATE( ) --- A "constructor" does not use an instance of the ADT, but creates one.

Example ADT: Stack of Integers

Data: (Ordered) list of integers.

Methods:
  CREATE, PUSH, POP, TOP, DESTROY

Using Math. Functions to Describe ADT Methods

Why?
  Math. can be used to give a concise and unambiguous description of a method.

What?
  1. gives a clear indication of input & output.
  2. clarifies how the data changes and what is returned by the operation.

Math. Description of Methods: Domain and Range of the Method's function

Example: the POP operation on a stack can be described by a mathematical function:

$$f_{POP}: \textit{stacks} \rightarrow \textit{elements} \times \textit{stacks}$$

stacks = the set of all possible stacks (according to this ADT).

elements = the set of all acceptable elements in our stacks (e.g., integers).

Because the operation produces an element and it causes a change to the stack, the range of f_{POP} is a cartesian product.

The Function's Effect

Now we describe how the function changes the stack and produces a value.

$$f_{POP} : [e_0,e_1,\ldots,e_{n-1}] \mapsto (e_{n-1}, [e_0,e_1,\ldots,e_{n-2}])$$

The symbol " $\mapsto$ " means "maps to". On its left side is a description of a generic *domain* element. On its right side is a description of the corresponding *range* element.

This formula indicates that the POP operation takes an n-element stack, returns the last element, and removes it from the stack.