måndag 26 januari 2015

Datatyper och Algoritmer

Introduction

Goal:

Improve programming skills

Understand complexity in time space (comparing)

Understand important types of algoritms and data types

Demands:

Read the book

Exercises

Assignments/exercises:

Analyzing algorithms

Test

Debugging

Tables

Group:

- Complexity analysis
- Memory and files
- Algorithm construction
- Advanced data types
- Preparation for the exam

Exam:

15 minutes before!

The actual course

Algorithms

Set of instruction for calculating or doing something

Examples
Sorting
Path finding
Item finding
Controlling

History
Something, something, something

Time and space complexity
Pseudocode
Design

Data types

A way of classifying pieces of information
Useful for computers

Examples
Primitive
- Integers, reals(floats, doubles), boolean

Composite
- Lump of primitive
- Arrays ,struct, unions

Abstract
- Stacks, queues, heaps, trees

Program design

Writing code

Testing code

Datatypes

Integers

int nVar; (should be the bit of the computer (64bit cpu -> 64bit int, but is not guaranteed))

short int nVar; (16bit, guaranteed)

long int lVar; (32bit, guaranteed)

Tip: declare only one variable per line

unsigned int nVar; (only positive)

int nArray[nSize];

Floats/doubles

float fVar;

double dVar; (32/64bit)

long double dVar; (64bit)

Char

char cVar; (8bit)

signed char cVar;

char strArray[nSize]; (must have a \0 in the end)

Void

void* pVar;

void *pVar; (asterisks position doesn't matter)

Program Design

The ideal way:

- Customer specification (should be user guided)
- Program / top level specification
- Top level design (design)
- Low level design (functions)
- Implementation

Tip: plan ahead/design before programming

Customer specification
From the users point of view
What to do but not how

Program / top level specification
Modelling the system
What data
What function/methods

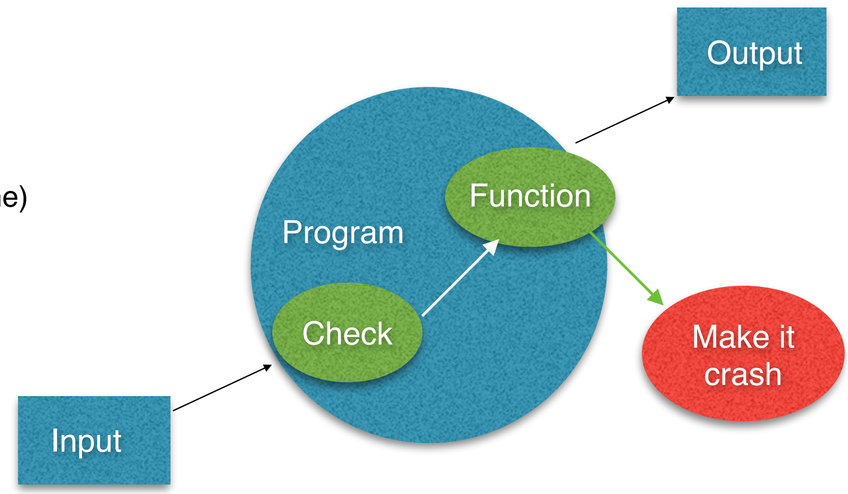
Top level design
Structured analysis / Structured design (problemed, out of fashion)
Object oriented (Encapsulation, reuse)
Agents Design (Task oriented, look at tasks)

Low level design
Function design
- KISS (keep it simple stupid)
- Does one thing!

Algorithm design
- Reuse other ones, and analys for the best one

Pseudocode

Program boundary
If something gone wrong, make it crash
Don’t hide crashes
assert() is good (only works if debug is one)



Writing Code
Hungarien notation:
- n - integer, nVar
- l - long, lVar
- f - float, dVar
- b - boolean bVar
- m_ - member m_nVar
- o - object oVar
- Extra: fp, i, j

Block structure
- {}
- Indentation

Testing Code

Does it meet the requirements?
- Time constraints
- Space constraints

Does it work (bugs?)
- Show others your code

Usability

Block / function test
- test a part of the code

Assambly / integration test
- Assemble the tested code

Customer / validation testing

Different Ways:

Static:
- Test a code before you run it
- Asymptotic analysis
- Logical proof
- checking (show others)

Dynamic
- Running the code
- Debugger:
  - Break points
  - Check variables
  - Change variables
  - Jump in and out of code
  - Step though code
  - Look at the memory

Analys of algoritmer

- Exekverignstid
- Minnesåtgång
- Implementationkomplexitet
- Förstålighet
- Korrekthet

Exekveringstid/minnesåtgång

Varför analysera:
är algoritmen praktiskt körbar
Vill ha den snabbaste - att implementera och köra

Beräkningsbar/hanterbar

Olika kategorier:
Beräkningsbara - Ej beräkningsbara(går ej göra en algoritm av)

Beräkningsbara:
Hantera polynom $1+n^2+3*n$ (kan lösas på en rimlig tid)
Ej hantera superpolynom ($n!$, n^n, …

Stora ordo

Sätt att förenkla funktioner av tidsåtgång

Gäller att hitta en funktion som är garantera att vara större en den riktiga funktionen efter en viss punkt.

Def:
Giver funktionen f(n) och g(n) säger vi att f(n) är 0(g(n)) om och endast om f(n <= c*g(n) för >= n0 och c < 0 n0 > 1.

f(n) är O(g(n))

Välja rätt algoritm är mycket viktigare än att programmering ska vara optimerad

Ohanterbara:
Shemaläggning
Handelresande
Moore’s lag förändrar den situationen? nej

Hur hanteras ohanterbarhet:
Heuristik:
- Förenkling (Lösa nästan rätt problem)
- Approximation (Lösa problemet nästan rätt)

NP-kompletta problem
En speciell klass av ohanterabara problem
Har problem x en lösning med egenskaperna y
Ekvivalenta:
- Transformeras (om ett problem kan lösas kan alla lösas)
- Högst exponentiella
- Saknar bevis för ohanterbarhet

Mäta tidsåtgång

Hur:
- Experimentell analys
  - Implementera algoritmen
  - Kör programmet med varierande datamängd
    - Storlek
    - Sammansättning
  - Använda metoder för tidtagning så som
    - time (sekunder)
    - clock (processortid)
    - gettimeofday (bättre precision än time, men ej standard)
  - Plotta uppmätt data

Om det är svårt att sätta en funktion på plotten:
Bästa, värsta, medel

Begränsningar:
- Måste implementera och testa algoritmen
- Svårt att veta om programmet har stanna eller fast i beräkningar
- Experimenten kan endast utföras på en begränsad mängd data
- Hård och mjukvara måsta varas samma för all implementationer

Kontrollera sin slutsats
- Plotta uppmätna tid/uppskattad (f(n)/g(n)
  - Borde gå mot konstant eller 0 för stora värden om korrekt
    Konstant betyder bra, 0 betyder för stor uppskattning
    Lätta att missa ln(n)

Asymptotisk analys (maskinoberoende)

Utgår från pseudokod
Räkna operationer
- Ställ upp ett uttryck för antalet operationer beroende av

Förenkla tidsuttrycket
Ta fram en funktion som begränsar tidsuttrycket ovanifrån...

Primitiva operationer som räknas som en operation:
- Lågnivå beräkningar som är i stort sett oberoende av programspråk och kan definieras i termer av pseudokod:
  - Anropa en metod/funktion
  - Returnera från en metod/funktion
  - Utföra en aritmetisk operation (+, -, ...) ✳ Jämföra två tal, etc
  - Referera till en/ett variabel/objekt
  - Indexera i en array

```
Algorithm addAllEven(n)
    sum <- 0              1
    i <- 2                1
    while i<=n do         (n/2+1)*1 +(n/2)*[]   ([] är innehållet i loopen)
        sum <- sum+i      1+1 (skulle kunna skrivas += vilket är en operation)
        i <- i+2          1+1
    return sum;           1
```

T(n)=1+1+(n/2+1)*1 +(n/2)*(1+1+1+1)+1=4n+n/2+

En for loop räknas: 1 operation för initiera, n(1+1) antal ++, innehållet ([] + 1) * n

Memory and Files

Memory

Memory need drivers
HAL - Hardware abstract level
stdin - standard in (keyboard…)
stdout - standard out (screen…)
stderr - stadard error (screen…)
file - device

Von Neumann architecture (standard)
Linier
Program code and data is in the same block, one long line
Everything has an adress
Normally uses hex adresses (0x<hex number>) (ex: 0xf8 - 0xff serial I/O)

Direct Memory access
Paging / caching

Variable storage class:

- Automatic
  - Declared at start of a block
  - Memory allocated as needed
  - Stack
- Register (keyword) (saying this memory needs to go fast)
  - Loke automatic but stored CPU registers (if possible)
  - Faster code
- External
  - Global variables allocated for the life time of the program
  - Not a good idea!
- Static
  - Limited scope
  - Allocated for life time of the program
  - Heap

Data Alignment
- Word size access

Structure size
- Padding

Aligment
- n-bytes

Speed vs memory usage

måndag 26 januari 2015

Dynamic memory

- From stdlib.h
- malloc
- realloc (changing size)
- calloc
- free
- return void or void*

How:
- void* malloc(size_t size)
- size_t - unsigned integer (min 16 bits)
- size - number of size
- void* - the adress of the first byte of the allocated memory
  - if null then memory allocation failed
  - **Always** check for null!!! (assert)

- void* realloc(void *ptr, size_t size)
- Resizes an already allocated block
- Returns point ro block
- Use the returned pointed as the block may have moved in memory
  - Always check for null

- void* calloc(size_t num, size_t, size)
- Like malloc but allocates for an array
  - size - some of array element
  - num - number of elements
  - Always check for null pointer

- void free(void* ptr)
- Returns no longer used memory back to the heap
- ptr - pointer to a block of memory previously allocated buy an dynamic memory function

Caching algorithmes

cpu -(fast)- cache -(slow)- memory

- Maximise fast memory access
- Minimise slow memory access
- How:
- Pages
- (speed, size)

Best way to cache program:
- Least recently used (keep)
- Most recently used (keep)

Memory problems

Memory Leaks:
- Failing to free memory
- Keep track of allocations

Over stepping the bounds:
- Going beyond the limits of an allocated block
- assert the variable (index)

Running out of memory
- Check for null pointers

Other memory Functions

- malloc.h (intel specific, (old))
- void* alloca(size_t size)
  - void* _malloca(size_t size)
  - Allocates from the stack (don't use free!)
- void* far_fmalloc(size_t size) (intel)
- Big endian / little endian
  - Big - most significant byte first
  - Little - least significant byte first)

File IO

C standard
- Buffered file system
- Formatted

Unix standard
- Unbuffered
- Unformatted

Streams and files
- Constant interface regardless of device - stream
  - Abstraction
- Device being used
  - File

Types of streams:
- **Text**
  - Sequence of characters
  - Lines of text (new line character)
- **Binary**
  - Sequence of bytes
  - One-to-one correspondence (may have null bytes on the end)

Control / managment functions

- FILE* fp = fopen(const char* filename, const char* mode)
  - Filename chan include path specification
  - Mode
    - r, w, a, r+, w+, a+, rb (read binary) …
- int fclose(FILE *fp)
- int fflush(FILE* fp)   (flush to make sure the data is written, if fp == null, then all file are flushed)
- int remove(char* filename)
- void rewind(FILE* fp)   (put back the data pointer to the start)
- int fseek(FILE* fp, long number_of_bytes, int origin)   (jump to different parts)
- int feof(FILE* fp)   (important fo binary, birary date, false end of file, return true after the file ended)
- ferror()
- fScanf

Input functions
Output functions
in stdio.h

- file* fp
  - structure defining file

All about in and out

Stack: Filling up the stack with data. Last in first out
Queue: First in, first out

Stack operations

(see presentation for specific implementations of the operations)

Set of data
S[1 … S.top]

Operations:
- Is empty
- Push
- Pop

Stack Implementation

With
- Array (problem with size: fixed and unused)
- Linked list

Queue operations

(see presentation for specific implementations of the operations)

Q[Q.head … Q.tail]

- Enqueque(Q.tail) (like push)
- Dequeque(Q.head) (like pop)

> The Q.tail loops together with Q.head and therefor as long as head < tail the data is never overwritten

Queue implementations

With
- Array
- Linked list (you don't have to loop)

Complexity

Always constant O(1)

Testing

Aims

Correctness (verification)
- Requirements
- Time
- Space

Reliability (validation)

Maintainability (regression)
- Important with documentation

How to meet those terms:
- Test each module/function
- Integrate and test larger units
- Test whole program

How to do that:
- Formal reviews (present your work to someone else)
- Standards
- Measurements (time, size, comments, how much percent is tested)

Testing

What to test for?
- Memory leaks, overriding arrays, overflows

What can be tested for?

What does the test actually tell us?
- Does a crash means it doesn't work?

When to test
- Often

Static Testing

- Analysis the code (looking at the code, let someone else to look through it)
- Logical analysis
- Check standards
- Check loops / reachability
- Check variables
- Check documentation
- Metrics (code size, complexity)

Dynamic Testing

- Running the code
- Debugging

- — Software specification
- -> Test cases
- -> Test scripts
- -> Desired results
- -> Code execution
- -> Compare results

Test cases:
- Derived from specification

Valid input:
- Specification

Invalid input:
- > max
- < min

Test plan
Memory leaks?

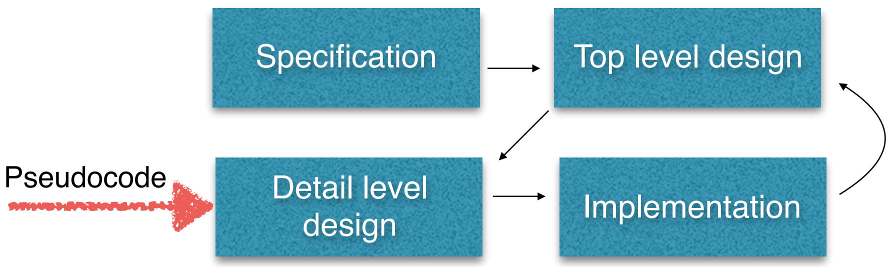
Coverage (can't test everything):
- Statement coverage (lines of code)
- Decision coverage
- Path coverage

Boxes:
- Black box (you don't know what code your testing, just input and output)
- White box (look through the code your testing)

Pseudocode

No standard

Top level (often OOP)
Detail level (functions, algorithms)



- Program design language:
  - Design algorithms
- Structured English
- Code like syntax
- Design -> comments

What do you do in program? (what pseudocode includes)
- Declarations / assignments
- Loops
  - For
  - While
- If statements
  - Else
  - Switch
- Function calls
- Logic operations
- Blocks

- Local variables (assumes local)
- Variables as objects
  - Attributes
  - *attr*[x]
- References
- Loops:
  - Same as in C, C++, or Java. Loop variable still defined after loop
    - for i <- 0 to max do
      - …
        - …
  - while i < 0 to max do
- If statements:
  - if a <= b then
    - …

- Function calls
  - Parameters passed as pointers
  - Change attributes

- Logic operations
  - and
  - or
  - not
- Short circuiting (if first in an if statements it doesn't bother calculating the rest)

- Indentation defines blocks:

- …
  - …
    - …
    - …
  - …

Algorithm

What is algorithm?:
- Problem solving instructions
- Structured method
- Detailed, step by step plan
- Calculable

- Finiteness
  - The algorithm must stop
- Assertiveness
  - Each step must be unique
- Input
  - Must have zero or more input
- Output
  - Must have one or more output
- Efficiency / Feasibility
  - Each stop f the algorithm must be done in a finite time

Lists

Abstract data structure
Data arranged in liner order

Types if lists:
- Arrays
- Ordered / unordered
- Linked lists
  - Single
  - Double
  - Circular
- Skip lists

Linked list:
- Head
- Data

Double linked list
- Tail

Operations on lists:
- Insert
- Delete
- Search
- Min / Max
- Successor
- Predecessor

Sentinel
- Null dummy objects in the end

Implementation issues
- Pointers
- Memory leaks

Skip lists
- Local line
- Express lines
- To save time
  - if theres to many in express line or to few then it's inefficient
  - Most used
    - Dynamic skip list
  - Uniformly
    - Evenly spaces nodes (best case)
- Even faster skip list: add more express lanes

Sorting

Bubble sort

Worst case: $O(n^2)$ $T_n = cn^2+cn+c$
Best case: $\Omega(n)$

Memory:
Small - fixed
$O(1)$

Stability

stable if multiple of same keys is sorted in the same ways evey run
unstable if the keys may change places
ex: 1:b 1:a 5:c 10:e -> 1:a 1:b 5:c 10:e

Bubble sort is stable - two equal elements will never get swapped

Quicksort

Divide and conquer

Algorithm:
Divide the array in to two halves
- about a pivot

conquer by recursive calls to quicksort
Combine - no need as they are sorted in place

Proper algorithm:
```
Quicksort(a, p, r)
if p < r then
        q <- Partion(a, p, r)
        Quicksort(a, p, q-1)
        Quicksort(a, q+1, r)
```

Initial call: Quicksort(a, 1, n)
p = pivot

Quicksort time complexity

(See presentation for specific math and calculations)

Worst case:
- pivot is around max or min (not the middle)
- Sorted or reverse sorted

All elements are distinct

$O(n^2)$

Best case:
• pivot in middle
$T(n) = O(n\log_2 n)$

Middle case:
ex: 1:9

$O(n\log n)$

Quicksort improvement

Choice of pivot
- Randomise

Improve the partition part

Quicksort stability

Not stable
- It is possible to partiton the sort in such a way that to elements with the same key get swapped
- add a second key to preserve order

=> stable

Debugging code

What can go wrong?

Syntax error
Run time error (array over runs, etc …)
Logic error

Common code errors
- == & =
- Break in switch statements
- Forgetting &
- Arrays overruns
- Array sizes
- integer devision (double d = 1/3)
- Where are you pointing
  - Initialize all your pointers to NULL
  - Have you freed the memory
- Strings terminated with '\0'
- fgetc returns an int

Bug hunting

How to find bugs?:
- Compiler output
- Code reviews
- Assert
- Debugger
- Scripts
- Robust code!

Debugger:
- Breakpoints
- Stepping (in, over, through)
- Variables
- Call stack
- Memory

Codeblocks:
Menu -> debug -> toggle breakpoints

Table, hash tables, relation and dictionaries

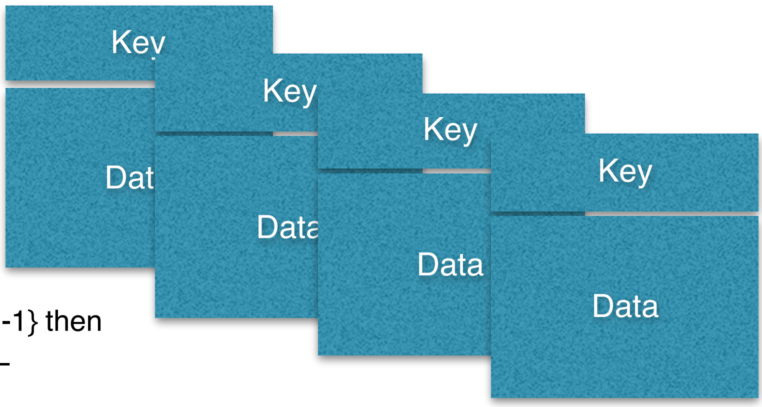
abstract datatypes - can be implemented different ways

Tables

- Library database
  - Books with unique / ISBN
  - Authors can author more than one book
- How to organize them?
- Organize by title
  - Arrange alphabetically
- Arrange in numerical order
- Do I have a given book?
  - Search using letters or ISBN

Tables formally

- Collection of related data
- Implementation
  - Array
  - Linked list
  - Hashing
- Operations
  - Search
  - Insert
  - Delete
  - Empty
- Record (item, post)
  - Key
  - Data
- To access the table
  - Keys taken from set $U \subseteq \{0, 1, \ldots, m-1\}$
  - Set up an array of used keys $T = \{0, 1, m-1\}$ then
  - $T[k] = \{x$ if $x \in k$ and $key[x] = k$ or NULL
  - To find: O(1)
- Key :
  - By using an array as key holder
  - Problems:
    - Large keys (array)
    - Not all position used (unused spaces in the array)



Hash tables

- Library Database
- Organize according to author?
  - (even book title or parts of an ISBN)
- Divide up in to chunks

- $U \subseteq \{0, 1, \ldots, m-1\}$
- S (subset) $\subseteq$ U
- Map h(s) → U
- When hash keys points to multiple data (collision):
  - Use link list. ex 0 points to 23 and 25, but 23 points to 25. So it goes 0 → 23 → 25
  - Another hash table
- Worst case:
  - Everything maps to the same location
  - O(n)

Hash functions

- Probability of hashing to a given slot just a likely regardless of previous hashing
- Uniformity in keys should not effect probability
- n = number of keys
- m = number of slots
- Load factor $\alpha = n/m$
- Unsuccess … check presentation

- h(k) = k mod m
- Problems:
  - m with a small divisor
- Pick m as prime number (but not close to 10 or 2)

- h(k) = (A*k mod 2^w) rsh(right shift bit) (w-r)
- w = word size
- A is an odd integer $2^{w-1} < A < 2^w$
- r = integer

Open addressing

Another name: closed hashing

- Resolving collision through repeated probes (another hash funktion)
- Depends on hash function + probe
- Search:
  - Same probe sequence
- Problems:
  - Table can become full
  - Deletion can be difficult

- Linier probing:
- h(k, i) = (h'(k) + i) mod m
- h'(k) = ordinary hash funktion
- Can lead to clustering (blocks and wasted space in-between)

- h(k, i) = (h_1(k) + i*h_2(k)) mod m
- m needs to be the power of 2
- h_2(k) needs to produce odd numbers

Relations

- Table of related data
  - Attributes
- Relations
  - Connecting attributes from table to table

Dictionaries

- Associative list or map
  - Key, value
- Operations:
  - insert
  - delete
  - modify
  - search

Problem solving

Breaking and entry

Brute force:
- Simple
  - Start at the beginning
  - Keep going till …
  - You find a solution
- Ex: searching a list, traveling sale man problem, make words out of other words
- It can work, sometimes only solution
- Problem:
  - Inefficient
  - Can become time consuming

Optimal subproblems:
- Optimal solution has optional subproblem solution
- Optimal sub-structure
- Greedy algorithms
- Else
  - Overlapping substructures
  - Dynamic programming

Greedy Algorithms:
- Local optimum results in global optimum (not guaranteed)
- Ex. problem:
  - Shortest path from A to B:
  - Always take the shortest sub-rout
- Not all local optimal solutions can lead to global optimal solutions
- If globally optimal can be fast
- Problems:
  - No back tracking

Dynamic Programming (not as in computer programming):
- Overlapping sub problems
  - Once calculated, save and reuse
- Dynamic
  - Updates and changes things but not as in dynamic languages
- Programming
  - Filling tables, not computer programming

Divide and Conquer:
- Recursion
  - Break the problem reclusively into small problems
  - Preferably evenly
- Solve the smaller problems
- Combine together to produce the overall solution
- Master method the find O(), look up in table

- Master method:
  - $T(N) = a*T(n/b) + f(n^c)$
  - Three cases:
  - If $f(n) = \Theta(n^c)$, where $c < \log_b(a)$ then $T(n) = O(n^{\log_b(a)})$
  - If f(n) = O(n^c*log^k(n)), where $c = \log_b(a)$, then T(n) = O(n^(logb(a)*log^(k+1)(n))
  - If f(n) = Ω(n^c) … (se presentation)

Problem Solving

Top down:
- Greedy

Bottom up
- Dynamic

Trees

Why trees?

- Structural relationships
- Hierarchies
- Efficient insertion and search
- Flexibility (moving tree around)

Examples:
- Navigation (robot)
- Files systems
- Family trees (animals, relationships)
- Assembles (things that are assembled)
- Hierarchies
- Decision making

Terminology

- Node (contains data, connected)
- Root (top node)
- Branches
- Leaves (end nodes)
- Label (identifier)
- Parent (above node)
- Child (below node)
- Sibling (beside node)
- Levels (1, 2, 3, …)
- Position (where nodes are)
- Height (how many levels, h(o) = leaves, h(1) = parent)
- Depth (reverse of hight, d(0) = root)

- Recursive
- Finite number of nodes
- Homogeneous data type (same type)
- Sub tree (part of tree)
- Unique path (each node have, hmmm)
- No multiple paths

- Different types of trees:
  - **Ordered**
    - Sibling linearly ordered
    - Siblings in a list
  - **Unordered**
    - Disorganized
    - No significance between siblings
  - **Directed trees**
    - Paths in only one direction
    - Down directed trees lack parentes
    - Up directed trees lack children
  - **Binary trees**

    - Max two children
  - **Sorted**
    - Relationship / values

- **Binary trees**
  - Left child
  - Right child
  - Number of nodes in a binary trees
    - At least n = 2*h - 1
  - Number of leaves in a full binary trees
    - l = (n+1)/2
  - Full binary trees
    - Each node has 0 or 2 children
  - Complete
    - Full at each level with exception on leaves
  - Balanced
    - Left and right sub trees have same huber of nodes

  - Height:
    - $n/2^k = 1$, n = number of nodes att bottom level
    - $k = \log_2 n$
    - Max number of nodes per level
      - 2^k (k-1 = levels) (1, 2, 4, 8, 16, …)

Operations

- Navigate
- Balancing
- Enumerate
- Search
- Insert
- Delete
- Pruning (delete subtree)
- Grafting (insert subtree)

- **Navigate**
  - Travers
  - **Breadth first:**
    - Create a queue
    - Save all out going nodes to the queue
    - Work though the queue and check each node
    - End when queue is empty
  - **Depth first:**
    - Walking the tree
    - Walk
    - Pre-order
      - Parent traversed before child
      - Go down left child, back up, go to right
    - Post-order
      - Child traversed before parent
    - In-order
      - Left tree, then node, then right tree

- Balancing a tree
  - Adding sorted data to tree
  - Tree becomes like a list
  - Rotate left
  - Red-black trees
- Delete
  1. Deleting a leaf
  2. Deleting a node with one child
  3. Deleting a node with two children
  - Case 2
    - Remove node and replace with child
  - Case 3
    - Select either in order predecessor or successor, r, of n
    - Copy r to n
    - Recursively call delete on r until case 1 or 2
- Insert in red-black tree
  - Se presentation for red-black specifics

Implementation

- Linked list
- Label
- Pointer to parent
- Array of pointers to children
- Struct

- As an array
  - Label
  - Parent

Priority Queue

Special case:
- Queues
- Stacks

Operations:
- Insert (or push, enqueue)
  - With priority
- Remove (or pop, dequeue)
  - With priority
- isEmpty
- inspectFirst

Implementation:
- Array
- Linked list
- Heap (tree)

Priority Queue Examples

- A* (path finding)
- Discrete Event Simulator (not-real simulation (chunk of time simulation))

Heap

- Tree data structure
  - Ofter binary tree
- Ordered relationship between child and parent
  - Not between siblings
  - Heap property
  - Max / Min heap

Operations:
- Create
- Insert
- Delete (max or min)
- isEmpty
- Extract max or min

Heap Implementation

- Usually as an array
- (Often) Binary tree
- Almost complete

Priority queue as a heap
- Root at index 1
- Left at index 2
- Right at index 3
- For node n, children at 2n and 2n+1

Operations:
- Insert
- Add at end
  - New leaf on the tree
- Fix the priorities
  - Compare with parent
  - If smaller, swap
  - Continue until priorities are fixed
- RemoveMax
  - Return at index 1
  - Copy end of array to index 1
  - Compare and swap to restore priorities

Building a Heap

Build a heap
- In a binary tree, where do the leaves start?
  - floor(n/2) + 1 to n
  - n = 5 laves start at 3
- Every leaf is a heap with one node!

Heapify
- Go up and down to compare each element in a tree and swap them to create a heap

How many nodes at each level?
- ceil($n/2^{h+1}$)
- Height h = 0, n = 7, number of nodes = 4

HeapSort
?

Heap operations

(See presentation for details)

- Extract max
- Insert
  - Insert at the end and copy upwards till heap proper is satisfied
- HeapSort

Variations of Heap

- Fibonacci heaps - set of trees ((often) min), no predefined order, number of children are low
- Binomial heaps - collection of tree on order less than parent

Sets

- Collection of items
- No specified ordered
- Unique values
- Implementation of mathematical concept of finite set
- Static or dynamic

- Items in a set are members of the set
  - $x \in X$
- Subsets
  - $a \subseteq A$
- Union of sets
  - $A \cup B$
- Intersections
  - $A \cap B$

Operations
- Create
- Inset
- Remove
- Is member of $\in$
- Is empty
- Select (lookup)
- Size of
- Enumerate
- Compare

Implementation

- Simple
  - Linked lists
  - Array
- Efficient
  - Trees
  - Hash tables

- Insert
  - Check for duplicates
  - Union
- If list has duplicates
  - Check when doing
    - Equal
    - Remove
    - Intersection
    - Difference
- Bit vector (array)
  - 0 or 1
  - Set the bit if the item is in the queue

- Faster operations
  - Bit operations hardware (bitwise AND or OR)
  - Masking
    - 0010110101 AND 0010000000
- Minimize memory
  - n/m

- Bit field
  - Struct {int var :3;}

- Limitations
  - Cant' use bit field variables in and attar
  - Can't take the memory adress of a bit field variable
  - Can't overlap integer boundaries

- Priority Queues
  - Linux kernel
- Caching algorithms / memory pages

Complexity

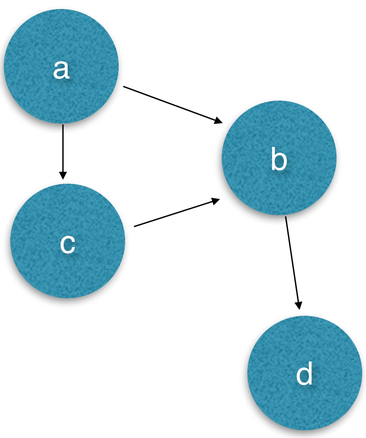
Depends on the implementation
- Improve set operations such as union or intersection
- Improve insert, search, remove

O(n) or O(logn)
Some set operations can take O(m*n)

Another bit on the bit vectors:
- Number of bit in an array
  - Hamming Weight
    - Population count
    - Bit wise operations
    - Counting

Graphs

- Set of nodes or vertices + pairs of nodes
  - G = (V, A)
- Directed or undirected
  - Undirected a to b is the same as b to a
  - Node - Undirected
  - Vertices - directed
- Edge
  - Arcs (directed)
  - Connection between nodes
  - Weighted or unweighted



- V = {a, b, c, d}
- A = {(a, b), (a, c), b, d), (c, d)}
- Adjacency
  - 2 edges are adjacent if they share a common vertex
  - (a, c) and (a, b)
  - 2 vertices are adjacent if they share a common … (se presentation for additional notes)

Implementation:

```
struct Node {
	int nodeID;
	void* out;
	int out; (antal som går ut)
};
```

OR

```
struct Edge {
	int edgeID;
	int start;
	int end;
};
```

Types:

Trivial graph
- One vertex (node)

Edgeless graph
- Vertices and no edges

Null graph
- Empty graph

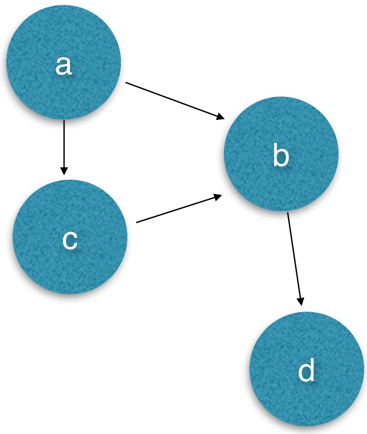
Terminology:

- Paths
  - Sequence of nodes
  - {a, b, c}
- Simple path
  - No repetition of nodes
- Cyclic path
  - Round and round
- Walk
  - Open walk = path
  - Closed walk = cyclic path

- Connected graph
  - All nodes have a path to all other nodes
- Sub graphs
  - Vertices are a sub set of G
  - Adjacency relationship are a subset of G's and restricted to the subgraph
- Complete graph
  - All nodes connected to all other nodes
  - Undirected graph A = n(n-1)/2
  - If a < n-1 then the graph is not connected

- Weighted graph
  - Maps
  - Places as node
  - Roads as arcs
  - Distance as weights on the edges
- Weights can represent ”cost”
- Some algorithm require restrictions on weights
  - Rational numbers or integers
  - All positive integers
- Weight of path
  - Sum of the weights for a given path

Constructing Graphs



- Adjacency lists
  - An array of arrays of adjacent vertices
- Adjacency matrix (a bunch of squares with sides abcd and abcd
  - Node x node matrix (n*n) (vertical and horizontal)

- Undirected graph
  - Symmetrical

Graph example

- Robot navigation

Grafalgoritmer

(Se presentation för algoritmer)

Traversiering

- **Djupet-först-traversering**
  - Rekursivt besöka alla noder - gå hela vägen genom noder fören besöka flera grannar
  - Cykler ger risk för oändlig traversering - kan undvikas genom att komma ihåg noder som besökts
- **Bredden-först-traversering**
  - Besöka alla grannar fören börja söka i nästa nod
  - Risk för oändlig loop och inget minnet av besökta
  - Implementeras best med en kö

- **Kortast-väg-algoritm**
  - Med hjälp av bredden-först och en distans funktion kan kortast väg hittas
  - Utgår efter ta alltid kortast väg till nästa nod

- Båda traverseringar har O(n) eftersom alla noder besökts en gång, (O(n+m) där m är antal grannar)

Finna vägar till en nod

- **Floyds shortest path (O(n^3))**
  - Bygger på att man representerar grafen med en matris
  - Loopar 3 gånger i varandra för att hitta längden av den kortaste vägen mellan alla noder
  - Kan läggas in mer loopar för att komma ihåg den väg som tagits
  - Sparar alla längder mellan alla noder i en matris

- **Dijkstras shortest path**
  - Kortast väg från en nod till alla andra noder
  - Alla noder har tre attribut: visited, distance, parent
  - Använder en prioritetskö (inte an vanlig kö)

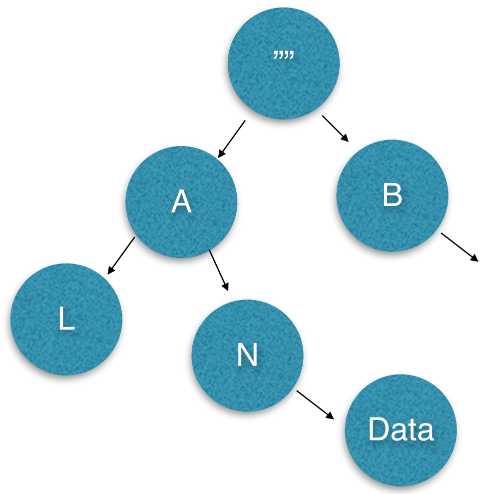
- Skillnad:
  - Floyds: jämför alla noder med all
  - Djikstra: jämför en nod med all - bättre för glesa träd

Konstrura ett (minsta) uppspännande träd

- bredden/djupet först gav båda uppspännande träd
- Graf med vikter: uppspännande träd ska ha så liten summa som möjligt

- **Kruskals Algoritm**
  - Greedy
  - Man färglägger olika delgrafer
- **Prims Algoritm**
  - By ett allt större träd
  - Varje steg välj med minimal vikt (greedy algoritm)

Trie



- Special type of tree
  - Retrieval
- Dynamic sets
- Keys is a string
  - Root node is an empty string
- Example:
  - Autocomplete/correct
  - File structures
  - DNA sequencing
  - Data compression
    - LZ78
    - Huffman encoding

Implementation
- As a table
- 2x2 array
- One columns per letter in the alphabet, n
- One row per node, m
- $\log_2 m$ to represent the data

- A a linked list
- Each node contains
  - A letter
  - A link to child
- de la Brandais tree

Operations
- Insert child
- Delete child
- Child
  - Lookup

Example LZ78:
- Lossless data compression
- Dictionary based
- Random access

Example Huffman coding:
- Frequency encoding
- To encode:
  - Right = 1 and left = 0 (in tree)
- Applications:
  - Zip file
  - jpeg
  - PNG

Binary search tree

- Each node has max two child nodes
- Relationship between child nodes
  - Nodes key is larger than all the nodes in the left sub tree
  - Nodes key is smaller than all the nodes in the right sub tree

- Binary search tree and a binary tree
  - In a binary search tree all nodes much have a label

  - Delete can break the tre
    - Fix it downwards
  - Insert must insert in sorted order

Operations:
- Search
- Insert
- Delete

Search
- Fast if tree is ordered
- If tree is complete: O(log *n*)

Insert
- Keep the tree complete
- Check left sub tree. If it is full, insert the values in the higher tree and move old down the tree

Delete
- Case 1
  - No sub tree
- Case 2
  - One sub tree
  - Move sub tree to the deleted node's position
- Case 3
  - Two sub trees
  - Delete the node
  - Promote the lowest value in the right sub tree (lowest value(right) takes the deleted values place)

Balancing a binary tree
- Rotations
- Self-balancing
  - Performed at key times (programmer decides what key times is)

Binary tree extensions
- Quad tree
  - As binary tree but based in 4 instead of 2
  - Breaks up a 2d region into 4 space
    - Collision detection

Revision

Data types

(from complex → simple)
- Graphs
- Trees (heap, trie, binary search trees)
- Queues and stacks (Priority queues)
- Linked lists
- Arrays, structures, unions (collections, builtin in C)
- Int, float, double, char, bool

(Abstract datatype = can be implemented different ways)

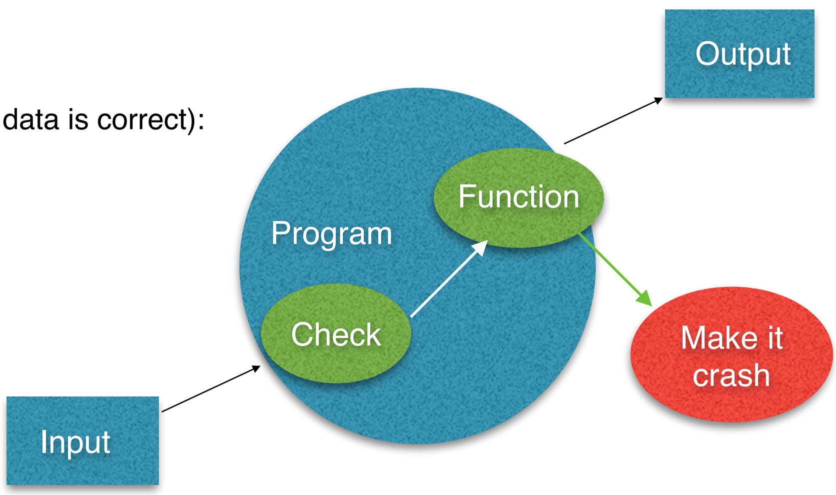
Algorithms

3 types:
- Constant time
- Iterative
- Recursive

Example:
- BubbleSort
  - $n^2/2 + n/2$. $n \to \infty$: $O(n^2)$
- Quicksort
  - O(log *n*)
- Mergesort
- Heapsort
- Greedy Algorithms
- Dynamic programming
  - Overlapping subproblems (save result for later use)
- Brute force

Programming / implementation

- Structures
- Arrays
- Pointers
- Program boundary (check you data is correct):
  - Memory allocation
    - malloc
  - i/o functions (fopen etc)
  - Other project functions



Datatyper Generella Teorier

(att själv designa en datatyp)

Abstract Datatyp

- Koncept för att diskutera och jämföra datatyper
- Hög abstraktnivå
  - Främst intresserad av struktur och organisation, inte implementation
- Operationerna ger datatypen karaktär
- och specifikation visar datatypens uttrycksfullhet

Operationskategorier
- **Konstruktörer** - skapa ny datatyper:
  - Grundkonstruktorer (empty, make, create, …) - inga argument
    - Boken: empty = inga värden eller struktur, make = inga värden men struktur, create = värden och struktur
    - Formalitiserat: Pythons __init__
  - Vidareutvecklade konstruktörer (insert, push, …) - flera argument
  - Kombinerande konstruktörer (Set-union) - flera argument
- **Inspektioner** - undersöka värden
  - Avläsning (inspekt, top, lookup)
  - Test av extremfall (tree-has-left-child
  - Mätning av objekt (Isempty, has-value)
- **Modifikationer** - ändrar datatyper
  - Insättning, bortagning, tilldelning (set-value, pop, insert)
- Navigatorer - ta fram datatypens struktur
  - Landmärken, lokala förflyttningar, traverseringar (first, end, next, tree-left-child)
- **Komparatorer** - jämför datatyper
  - Equal, subset

Generella Teorier

Uttrycksfullhet
- Abstrakt datatype = objekt + konstruktion av gränsytan
- Frågar vid skapandet av datatyp:
  - Vilken är värdemängden?
  - Vilka interna resp. externa egenskaper har objekten?
  - Vad ska man göra med objekten?
  - Specificera en gränsyta informellt och formellt
  - Överväga olika implementationsmöjligheter
- Vad kan göras med objekten?
  - Uttrycksfullhet

Hur mycket ska man kunna göra i datatypen?
- Skapa alla objekt i datatypen och skilja dem åt?
- Utnyttja all information i datatypen till att skriva algoritmer?
- Att algoritmerna man skriver också ska vara effektiva?
- Ställer olika krav på den gränsyta man väljer att hat till datatypen

Uttrycksfullhet

- Datatypespecifikationen har två roller:
  - Slå fast hur datatypen är beskaffad, vilka egenskaper den har
  - Fungerar som en regelsamling för användning av datatypen
- Specifikationens uttrycksfullhet kan mätas med tre begrepp
  - Objektfullständighet
  - Algoritmfullständighet
  - Rik gränsytan

Objektsfullständighet

- Är det svagaste kriteriet
- De ska vara möjligt att konstruera och skilja mellan alla objekt som anses höra till datatypen

Algoritmfullständighet

- Starkaste än (och implicerar) objektfullständig
- Man ska kunna implementera alla algoritmer i denna datatyp
  - D.v.s. allt som man kan göra med datatypen ska och gå att implementer utifrån specifikationens operatorer
- Alltså: Algoritmfullständighet = objektfullständig + likhetstest

Rik gränsyta

- Starkaste kriteriet, implecerar de andra två
- Även om man har algoritmfullständighet så kan vissa algoritmer bli hopplöst ineffektiva
- Krav: man ska med hjälp av gränsytan kunna implementer speciella analysfunktioner som kan:
  - Kunna plocka isär ett objekt och bygga ihop det igen
  - Får varken innehålla iteration eller rekursion i sin definition

Att utforma en gränsyta:

- Man utgår från de operationer som gör datatypen speciell
- Sedan applicera de teoretiska begreppen
  - Objektsfullständighet
  - Ska vara primitiva (kan inte delas upp i mindre operationer)
  - Algoritmfullständighet
  - Är oberoende, kan inte ta bort en enda operation och ändå ha kvar en algoritmfullständig gränsyta
- Detta ger en rätt stram yta med få operationer

Fördelar med en stram gränsyta

- Utbytbarhet
- Portabilitet
- Integritet
  - Mindre risk för att operation läggs till som strider mot grundidén med datatypen

Hänsyn till när skapa nya

Många programmeringsspråk ger mycket litet eller inget stöd alls. Då krävs:

- Namngivning
- Operationsval
- God dokumentation
- Disciplin (inte gå in och peta i interna strukturer)

Inför Tenta (exempel)

Exempel psuedo-kod: Jämför 2 arrayer om de är lika, oberoende av ordning och antalet av ett värde

```
Algorithm equal (arr1, arr2)
    if subEqual(arr1, arr2) and subEqual(arr2, arr1) then
        return true


Algorithm subEqual (arr1, arr2)
    for i <- low(arr1) to high(arr1) do
        val <- inspect-value(arr1, i)
        eq <- false
        for j <- low(arr2) to high(arr2) do
            if val = inspect-value(arr2, j) then
                eq <- true
        if not eq then
            return eq
    return true
```

Exempel insertsort med ord:

Vi går igenom alla de osorterade värdena en efter en och stoppar in dom på rätt plats bland de vi sorterat hittills

Exempel förklara vad är en prioritetskö:

Fungerar som en kö men värderna istället för ordning efter hur de kom in, behandlas de med någon relation som bestämmer ordning efter prioritets

Traversera bredd först behövs en kö, djup först behövs en stack

Vad menas med en linjär hantering av kollision vid hashtabell?:

Om kollision uppstår, gå linjärt till nästa oanvänd plats (samma med lookup)

Vilka problem uppstår?:

Kan bilda klumpar av värden som blir segt att söka igenom torsdag 12 mars 2015

Ordförståelse:

Abstract datatyp - en datatyp som är bestämd men inte hur den ska implementeras

Relativ komplexitet - komplexiteten för ens algoritm tar endast hänsyn till algoritm och inga yttre (primitiva) funktioner och datatyper som används

Glest matris - 2 dimisionellt fält med **många** värden som är 0 eller något annat fixed värde som inte skiljer värdarna ifrån varandra

Objektfullständighet - skapa varje element och de kan utskiljas, i en datatyp

Gränsyta - vilka operationer som en datatyp har och vilka parametrar de tar

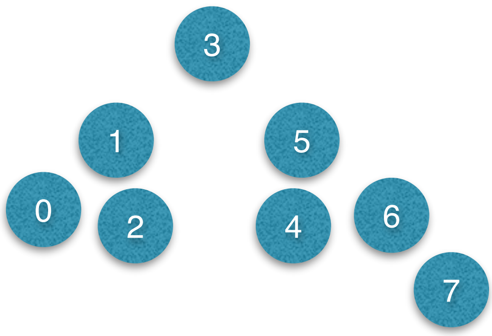
Hanterbara problem - problem som har en algoritm med högst polonomisk komplexitet

Komplett graf - graf där varje nod är granne med alla andra noder i grafen

Divide and Conquer - delar up problemet till mindre problem rekursivt och slår ihop de senare för att fylla huvudproblemet

Binärt sökträd: 3, 5, 6, 7, 1, 4, 2, 0

Hitta 2: 3 noder



Traverseringsorder

Djup först inorder (andra besöket): 01234567

Djup först postorder (rad för rad): 70246153

Minst antal total väg problem

Prim eller Kruskals(färglägg) algoritmer