Welcome!

COMP2521 19T0
Data Structures + Algorithms

COMP2521 19T0
Week 1, Tuesday: Hello, world!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

course introduction
more C syntax
linked lists, redux
tools of the trade

What?

Course Aims

thinking like a *computer scientist*
not just a programmer

know and understand
fundamental techniques,
data structures, algorithms

reason about
applicability + *effectiveness*

What?

Course Syllabus

Over the next few weeks...

- ADTS: stacks, queues, lists, trees, hash tables
- algorithm analysis: complexity, performance, usability
- sorting and searching techniques
- graphs, graph algorithms

Who’s Teaching?

Dr John Shepherd (jas@)
is the lecturer-in-charge

Jashank Jeremy (jashankj@)
is the lecturer

Sim Mautner
Hayden Smith
Clifford Sesel
Deepanjan Chakrabarty
Olga Popovic
Elizabeth Willer
Gal Aharon
Kristian Nolev

are your tutors and lab assistants

Who’s Learning?

recent students from...
COMP1511 (andrewt, andrewb, jas, ashesh)
COMP1917 (richardb, blair, salilk?, angf, simm)
COMP1921 (mit, ashesh, anymeyer?)

some C experience,
familiarity with pointers, ADTS,
style, and testing

(also a sense of humour)

Who's Learning?

...and what are they supposed to know?

At the start of this course, you should be able to

- produce a correct C program from a specification
- understand the state-based model of computation (variables, assignment, addresses, parameters, scope)
- use fundamental C data types and structures (`char`, `int`, `float`, arrays, pointers, `struct`)
- use fundamental control structures (sequence, selection (`if`), iteration (`while`))
- use and build abstraction with function declarations
- use linked lists

Who’s Learning?

...and what are they supposed to learn?

By the end of this course, you should be able to

- analyse performance characteristics of algorithms
- measure performance behaviour of programs
- choose + develop effective data structures (DS)
- choose + develop algorithms (A) on these DS
- reason about the effectiveness of DS+A
- package a set of DS+A as an ADT
- develop + maintain C systems <10 kLoC.

How?

by lecturing at you!
in interactive tutorials!
in hands-on laboratories!
in assignments and exams!

How?

...by lecturing at you!

- present a brief overview of theory
- demonstrate problem-solving methods
- give practical demonstrations
- lectures are based on text-book.
- slides available as PDF
  (*usually* up before the lecture... :-)
- feel free to ask questions...
  but No Idle Chatting, please.

Tue 14–17, Thu 10–13
Ainsworth G03

How?

...in interactive tutorials!

- clarify any problems with lecture material
- work through problems related to lecture topics
- give practice with design skills
  ... think before coding
- exercises available (usually) the week before
  please read and attempt *before* your class

Webster252 ...[MTW]10, [MW]14, T16
GoldsteinG01 ...F10
GoldsteinG02 ...[HF]14

How?

…in hands-on laboratories!

- build skills that will help you to
  …complete the assignment work
  …pass the final exam
- give you experience applying tools + techniques
- small implementation/analysis tasks
- some tasks will be done in pairs
- don't fall behind! start them before your class if needed
- usually up in advance, due by Sunday midnight

J17–306 sitar
[MTWF]11–13; [MWHF]15–17; T17–19

How?

...in assignments!

- give you experience applying tools/techniques to larger problems than the lab exercises
- assignment 1 is an individual assignment
- assignment 2 is a group assignment
- will *always* take longer than you expect
- organise your time
  ...don't leave it to the last minute!
  ...steep late penalties apply!

How?

...in exams!

- practical exams in weeks 5, 8; each worth 5%
- 3h theory + practical extravaganza; worth 55%

How?

...in exams!

- Supplementary exams are only available to students who
  ...do not attend the exam **AND**
  ...have a serious documented reason for not attending
- If you attend an exam
  ...you are making a statement that you are 'fit and healthy enough'
  ...it is your only chance to pass (i.e., no second chances)

Assessment

5% + 5% prac exams
10% lab marks
10% assignment 1
15% assignment 2
55% final exam

Course Evaluation and Development

assessed with myExperience

also, we’d love to hear from you...
provide feedback throughout the session!

Acknowledgements

Always give credit if you use someone else's work!
COMP2521 material drawn from...

- slides by Angela Finlayson (COMP2521 18x1)
- slides by John Shepherd (COMP1927 16s2)
- slides by Gabriele Keller (COMP1927 12s2)
- lectures by Richard Buckland (COMP1927 09s2)
- slides by Manuel Chakravarty (COMP1927 08s1)
- notes by Aleks Ignjatovic (COMP2011 '05)
- slides and books by Robert Sedgewick

Academic Conduct and Integrity

On Academic Integrity

You'll be fired into space
or, at least, out of this course
if you're found to be using others' work as your own.

The lawyers would like me to remind you that
UNSW and CSE consider plagiarism as
an act of academic misconduct with severe penalties
up to and including exclusion from further study.

Academic Conduct and Integrity

On Academic Conduct

...don't be a dick.

The lawyers would like me to remind you that
UNSW and CSE consider bullying, harassment, ..
both on- and off-campus (including online!)
an act of student misconduct with severe penalties
up to and including exclusion from further study.

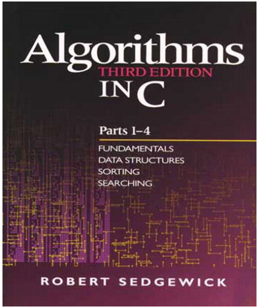
Course Website

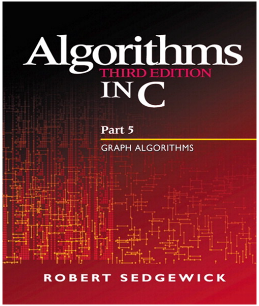
webcms3.cse.unsw.edu.au/COMP2521/19T0

cse.unsw.edu.au/~cs2521/19T0

Resources

The Textbook





Algorithms in C, parts 1–4 and 5, by Robert Sedgewick

BEWARE!

there are *many* editions/versions of this book,
with various different programming languages
including C, C++, Java, and Pascal

Resources

Additional Face-to-Face Help

- weekly consultations...
  for extra help with labs and lecture material
  more time slots scheduled near assignments/exams
  email cs2521@ for additional consultations, if needed
- help sessions...to be advised
- WebCMS3 course forums

Advice

- Do lab exercises and assignments yourself
  (or with your pair partner when appropriate)
- Programming is a skill that improves with practice
  The more you practice, the easier labs/assignments/exams will be.
- Don't restrict practice to lab times
  ...or two days before assignments are due.
- Make use of tutorials by
  ...attempting questions before the class
  ...participating!
- Go to consults if you need help or fall behind
- We want you to do the best you can!

More C Syntax

LOOKING FOR dcc?

dcc held your hand in *many* ways.
the training wheels are now off! no *dcc* for you!
if you're desperate, try *3c*

- compiling for normal use
  `$ 2521 3c -o prog prog.c`
- compiling multiple files
  `$ 2521 3c -o prog prog.c f2.c f3.c`
- compiling with leak checking
  `$ 2521 3c +leak -o prog prog.c f2.c f3.c`

Style in COMP1511/1917/1921

COMP1511, COMP1917, COMP1921
used a restricted subset of C

mandated layout, mandated brackets,
only `if` + `while`,
no side-effects, no conditional expressions,
functions with only one return...

... but this style is used in
no texts + no real code.

Style in COMP2521

the good
more freedom, more power!
more choice in how you express programs
can write more concise code

the bad
easy to produce code that's
cryptic, incomprehensible, unmaintainable

the style guide
available on the course website

Style in COMP2521

layout: consistent indentation
brackets: omit braces around single statements

control: all C control structures
(except `goto` ... that's how you get ants)

assignment statements in expressions
(but prefer to avoid side-effects ... that's how you get ants!)

conditional expressions ('ternaries') permitted
(use with caution! that's how you get ants!!)

functions may have multiple returns
(concise $\not\to$ clear! ants!!!)

‘for’ loops

with while

```
init;
while (cond) {
    /* … do something */;
    incr;
}
```

with for

```
for (init; cond; incr)
    /* … do something */;
```

‘for’ loops

with while

```
int sum = 0;
int i = 0;
while (i < 10) {
    sum = sum + i;
    i++;
}
```

with for

```
int sum = 0;
for (int i = 0; i < 10; i++)
    sum += i;
```

‘for’ loops
pros and cons

all interesting parts of the loop in one spot!
... but easy to write disgusting code

prefer `for` when *counting* or with *sequences*
... otherwise, use a `while` loop

'switch' statements

```c
if (colour == 'r') {
    puts ("red");
} else if (colour == 'b') {
    puts ("blue");
} else if (colour == 'g') {
    puts ("green");
} else {
    puts ("invalid?");
}
```

```c
switch (colour) {
case 'r':
    puts ("red"); break;
case 'g':
    puts ("green"); break;
case 'b':
    puts ("blue"); break;
default:
    puts ("invalid?");
}
```

the break is critical...
if it isn't present, execution will fall through

‘switch’ statements

```
char *month_name (int);
```

Exercise: Switched On

Write a function `month_name`
that accepts a month (1 = Jan …12 = Dec)
and returns a string containing the month name
… assume the string will be read only
… use a switch to decide on the month

Exercise: Hip, Hip, Array

Suggest an alternative approach using an array.

jumping around: 'return', 'break', 'continue'

avoid deeply nested statements!

return in a function
gives back a result to the caller
terminates the function, possibly 'early'

break in `while`, `for`, `switch`
allows *early termination* of a block
jumps to the first statement after the block

continue in `while`, `for`
terminates one iteration… but continues the loop
jumps to *after* the last block statement

Conditional Expressions (‘Ternaries’)

`if` statements can’t return a value.

```
if (y > 0) {
    x = z + 1;
} else {
    x = z - 1;
}
```

... but what if they *could*?

```
x = (y > 0) ? z + 1 : z - 1;
```

Conditional Expressions (‘Ternaries’)

Rewrite these using ternaries, or explain why we can’t do that.

Exercise: Rewriting (I)

```
if (x > 0)
    y = x - 1;
else
    y = x + 1;
```

Exercise: Rewriting (II)

```
if (x > 0)
    y = x - 1;
else
    z = x + 1;
```

Assignment in Expressions

- assignment is really an expression
  ... returns a result: the value being assigned
  ... returned value is generally ignored
- assignment often used in loop conditions
  ... combines test with collecting the next value
  ... makes expressing such loops more concise

Assignment in Expressions

```
int nchars = 0;
int ch = getchar ();
while (ch != EOF) {
    nchars++;
    ch = getchar ();
}
```

…or …

```
int ch, nchars = 0;
while ((ch = getchar ()) != EOF)
    nchars++;
```

Assignment in Expressions

Exercise: Mystery Biscuits

```
void what_does_it_do (void)
{
    int ch;
    while ((ch = getchar ()) != EOF) {
        if (ch == '\n') break;
        if (ch == 'q') return;
        if (! isalpha (ch)) continue;
        putchar (ch);
    }
    puts ("Thanks!");
}
```

Function Pointers

- In C, you may point to anything in memory.
- The compiled program is in memory.
- The compiled program is made up of functions.
- Therefore…you can point at functions.

- Function pointers
  … are references to memory addresses of functions
  … are pointer values and can be assigned/passed
  … are effectively opaque
  … (unless you're interested in machine code)
  … ((if you are, you'll enjoy COMP1521))

Function Pointers

return_t (**var*)(*arg_t*, …)

$\mathrm{int} \to \mathrm{int}$: `int (*fp)(int);`

$(\mathrm{int}, \mathrm{int}) \to \mathrm{void}$: `void (*fp2)(int, int);`

Function Pointers

```
int square (int x)    { return x * x; }
int times_two (int x) { return x * 2; }

int (*fp)(int);

// Take a pointer to the square function, and use it.
fp = &square;
int n = (*fp) (10);

// Taking a pointer works without the `&'.
fp = times_two;
n = (*fp) (2);

// Normal function notation also works.
n = fp (2);
```

Higher-Order Functions

functions that take or return functions

e.g., traverse an array, applying a function to all values.

```
void print_array (size_t len, char *array[])
{
    puts ("[");
    for (size_t i = 0; i < len; i++)
        printf ("%s\n", array[i]);
    puts ("]");
}
```

Higher-Order Functions

functions that take or return functions

e.g., traverse an array, applying a function to all values.

```
void traverse (size_t len, char *xs[], void (*f)(char *))
{
    for (size_t i = 0; i < len; i++)
        (*f) (xs[i]);
}

void print_array (size_t len, char *array[])
{
    puts ("[");
    traverse (len, array, &puts);
    puts ("]");
}
```

Using Higher-Order Functions

```
void traverse (link l, void (*f) (link));

traverse (my_list, print_node);
traverse (my_list, print_grade);
```

```
void print_node (link l)
{
    if (l == NULL)
        puts ("NULL");
    else
        printf ("%d -> ", l->data);
}
```

```
void print_grade (link l)
{
    if (l == NULL)
        puts ("(nil)");
    else if (l->data >= 85)
        printf ("HD ");
    else
        printf ("FL ");
}
```

Linked Lists

Recap: Linked Lists

- a *sequential* collection of ‘nodes’ holding value + pointer(s)
  ...no ‘random access’ to individual nodes
- easy to add, rearrange, remove nodes
- list node references other list nodes
  ...singly-linked list: `next` only
  ...doubly-linked list: `prev` and `next`
- last node’s `next` may point to
  ...`NULL` — no ‘next’ node
  ...a ‘sentinel’ node without a value
  ...the first node (a *circular* linked list)

Recap: Linked Lists in C

```
typedef int Item;

typedef struct node *link;
typedef struct node {
    Item item;
    link next;
} node;

// allocating memory:
link x = malloc (sizeof *x);
link y = malloc (sizeof (node));

// what's wrong with this?
link z = malloc (sizeof (link));
```

Recap: Linked Lists in C

```
// traversing a linked list:
link curr = …;
while (curr != NULL) {
    /* do something */;
    curr = curr->next;
}

// traversing a linked list, for loop edition
for (link curr = …; curr != NULL; curr = curr->next)
    /* do something */;
```

Functions on Linked Lists

Exercise: 'insert_front'

```
link insert_front (link list, link new);
```

Write a function to insert a node at the beginning of the list.

Would this prototype work?

```
void insert_front (link list, link new);
```

Exercise: 'insert_end'

```
link insert_end (link list, link new);
```

Write a function to insert a node at the end of the list.

Functions on Linked Lists

Exercise: ‘reverse’

Write a function which reverses the order of the items in a linked list.

```
link reverse (link list) {
    link curr = list;
    link rev = NULL;
    while (curr != NULL) {
        tmp = curr->next;
        curr->next = rev;
        rev = curr;
        curr = tmp;
    }
    return rev;
}
```

Deletion in Linked Lists

Demonstration: ‘delete_item’

```
// Remove a given node from the list
// and return the start of the list
link delete_item (link ls, link n);
```

Deletion in Linked Lists

- deletion is awkward:
  …we must keep track of the previous node
- can we delete a node if we only have the pointer to the node itself?
- we may need to traverse the whole list to find the predecessor
  …and that's if we even have a reference to the head

IDEA every node stores a link to both the previous *and* next nodes

The Doubly-Linked List

- Move forward and backward in such a list
- Delete node in a constant number of steps

```
typedef struct dnode *dlink;
typedef struct dnode {
    Item item;
    dlink prev, next;
} dnode;
```

The Doubly-Linked List

- Deleting nodes:
  easier, more efficient
- Other operations:
  ...pointer to previous node is necessary in many operations
  ...doesn't have to be maintained separately for doubly linked lists
  ...$2\times$ pointer manipulations necessary for most list operations
  ...memory overheads in storing an additional pointer

The Tools of the Trade

Documentation

learn how to access documentation ‘online’:
man(1), *info(1)* – available in exam environment!

you should even learn to *write* documentation:
mdoc, texinfo, doxygen, sphinx
all make it easy to document code and projects
(though are beyond the scope of the course)

man(1)

The Unix Programmer's Manual

the traditional 'Unix manual':
terse documentation in several sections
terrible tutorial, but great reference

commands *(1)*,
syscalls *(2)*,
library functions *(3)*,
file formats *(5)*,
the system *(7)*,
administrative tools *(8)*,
and more…

`man ls` gets *ls(1)*
`man printf` gets *printf(1)*
`man 3 printf` gets *printf(3)*

SOME USEFUL MAN-PAGES

intro in all sections,
stdio.h(0p), *stdlib.h(0p)*, *math.h(0p)*
printf(3), *ascii(7)*

info

GNU's Online Documentation System

GNU decided *man(1)* wasn't good enough
(a bundle of loose documents $\neq$ a good manual...)
so built the Texinfo system

SOME USEFUL INFO MANUALS

libc, *gdb*, *gcc*,
binutils, *coreutils*,
emacs, ...

the *info(1)* command
will fall back to *man(1)*-pages

other renderings of info pages:
dead trees, PDFs, web sites ...

Code, Compile, Crash, Confusion

Debugging in the Software Development Cycle

what's happening in your program as it runs?
why did that segfault happen?
what values are changing in my program?

"I'll just add some *printf(3)*s..."
clunky, not reliable, only gives what you ask for

a family of tools can help you find out:
debuggers

source debuggers: gdb/ddd/gud, lldb, mdb
specialist tools: valgrind, sanitizers

gdb

The Breaking-Point

`set args` *args*
: set command arguments

`run` *args*
: run the program under test

`break` *expr*
: set a breakpoint

`watch` *expr*
: set a watch expression

`continue`
: run the program under test

`print` *expr*
: print out an expression

`info locals`
: print out all local variables

`next`
: run to the next line of code

`step`
: step into a line of code

`quit`
: exit gdb

NOTE

you'll need to compile with `-g`
or GDB is very unfriendly indeed

Sanitizers

Programmers Must Wash Hands Before Returning To Codebase

{Address, Leak, Memory, Thread, DataFlow, UndefinedBehaviour}Sanitizer

a family of compiler plugins, developed by Google
which instrument executing code with sanity checks
use-after-free, array overruns, value overflows, uninitialised values, and more

you've been using ASan+UBSan already: *dcc* uses them!
usable on your own *nix systems (Linuxes, BSDs, 'macOS') too!
unfortunately… a bit of work to get going on CSE (hence *dcc* and *3c*)

```
clang -fsanitize=address,undefined -fno-omit-frame-pointer
   -g -m32 -target i386-pc-linux-gnu --rtlib=compiler-rt -lgcc -lgcc_s
      -o prog main.c f2.c
```

```
2521 3c -o prog main.c f2.c
```

valgrind

Have You Forgotten Something?

- finding memory leaks
  ... not `free`'ing memory that you `malloc`'d
- finding memory errors
  ... illegally trying access memory

```
$ valgrind ./prog
...
==29601== HEAP SUMMARY:
==29601==     in use at exit: 64 bytes in 1 blocks
==29601==   total heap usage: 1 allocs, 0 frees, 64 bytes allocated
==29601==
==29601== LEAK SUMMARY:
==29601==    definitely lost: 64 bytes in 1 blocks
```

Valgrind doesn't play well with ASan. Compile without '3c' if you really need it.

make

Making Everything Better

long, intricate compilation lines?
forgot to recompile parts of your code?

make lets you specify
rules, *dependencies*, *variables*
to define what a program needs to be compiled
doing only the necessary amount of work

implicit rules for compiling C (and more)
(`.c` $\rightarrow$ `.o`, `.o` $\rightarrow$ exec)

make

A Simple Example

```
CC       = gcc
CFLAGS   = -Wall -Werror -std=c99 -g
LDFLAGS  = -g -lm

# `prog' depends on `prog.o', `ADT.o'
prog: prog.o ADT.o
# `prog.o' depends on `prog.c', `ADT.h'
prog.o: prog.c ADT.h
# `ADT.o' depends on `ADT.c', `ADT.h'
ADT.o: ADT.c ADT.h
        ${CC} ${CFLAGS} -std=gnu11 -c $< -o $@
```

COMP2521 19T0
Week 1, Thursday: Abstraction, Your Honour

Jashank Jeremy

jashank.jeremy@unsw.edu.au

abstract data types, redux
fundamental data structures
testing

IMPORTANT

UNSW will have rolling short network outages
from **6am Sat 1 Dec** to **6pm Sun 2 Dec**.
save your work often if you're using VLAB!
CSE workstations may be affected.

Abstract Data Types

Abstraction

Meaning and Mechanism

"...the purpose of abstracting is not to be vague,
but to create a new semantic level in which
one can be absolutely precise."

— from *The Humble Programmer*
by E. W. Dijkstra (EWD 340), 1972

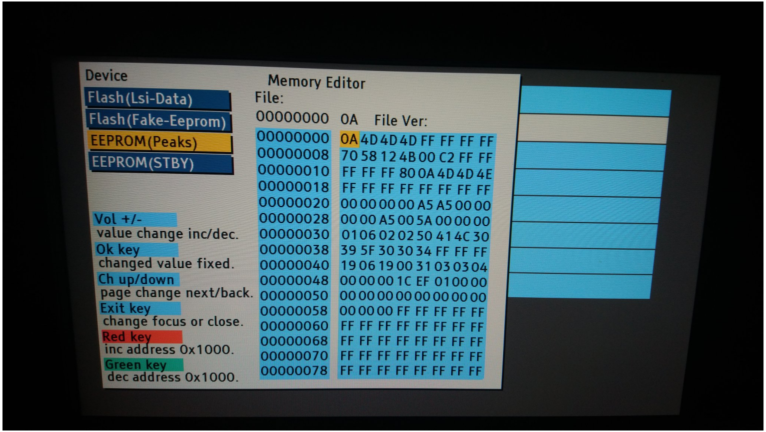
distinguish meaning and mechanism
...don't lose the forest for the trees

Abstraction

Abstraction in systems

To understand a system,
it should be enough to
understand what its components do
without knowing how...

e.g., we operate a television through its interface:
a remote control, and an on-screen display
... we do not need to open it up
and manipulate its innards



Good news: my parents TV has a hex editor
Bad news: I’m buying them a new tv

@0x47DF 2018-10-06 2142z

`twitter.com/0x47DF/status/1048342591668965377`

Data Types

Values and Operations

a set of values —

PRIMITIVE

`int`
(`char`, `short`, `long`, `long long`),
`float`
(`double`, longer!),
`void *`

COMPOSITE

`struct` *T*,
`enum` *T*,
`union` *T*

operations on those values —

`->`, `.`, `!`, `~`, `++`, `--` $+_2$, $-_2$, $*_1$, $\&_1$, $*_1$, `/`, `%`, $+_1$, $-_1$,
`<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, $\&_2$, `^`, `|`, `&&`, `||`, `?:`,
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`

Abstraction

Abstraction in the Machine

When designing a new library,
it is important to decide...

what are the abstract properties
of the data types we want to provide?

which operations do we need to
create, query, manipulate, destroy
objects of these types?

FOR EXAMPLE...
we do not need to know how
`FILE *` is implemented to use it

Abstraction

Creating Abstractions

We want to distinguish:

- DT (non-abstract) data type (e.g. C strings)
- ADO abstract data object
- ADT abstract data type (e.g., C strings)
- GADT generic abstract data type

ACHTUNG!
ADTs are not algebraic data types!
see COMP3141 / COMP3161 for more

ACHTUNG!
Sedgewick's first few examples
are ADOs, not ADTs!

Abstract Data Types

facilitate decomposition, encapsulation of complex programs

make implementation changes invisible to clients

improve readability and structuring of software

Abstract Data Types

Interface and Implementation

ADT interfaces provide

- an opaque view of a data structure
- function signatures for all operations
- semantics of operations (via documentation, proof, etc.)
- a contract between ADT and clients

ADT implementations provide

- concrete definition of the data structures
- function implementations for all operations

ADTs in C

Interfaces

- an opaque view of a data structure
  ... via `typedef struct t *T`
  ... we do not define a concrete `struct t`
- function signatures for all operations
  ... via C function prototypes
- semantics of operations (via documentation, proof, etc.)
  ... via comments (e.g., Doxygen)
  ... via testing frameworks (e.g., ATF-C)

ADTs in C

Implementations

- concrete definition of the data structures
  ... the actual `struct t` (and anything it needs)
- function implementations for all operations
  ... interface and internal functions

Stacks and Queues

Stacks and queues are

- … ubiquitous in computing!
- … part of many important algorithms
- … good illustrations of ADT benefits

Stacks

A stack is a collection of items, such that the last item to enter is the first item to leave:

Last In, First Out (LIFO)

(Think stacks of books, plates, etc.)

- Web browser history
- text editor undo/redo
- balanced bracket checking
- HTML tag matching
- RPN calculators
  (...and programming languages!)
- function calls

Stacks

Essential Operations

$\textsc{push} :: \mathcal{S} \to \texttt{Item} \to \texttt{void}$
add a new item to the top of stack $\mathcal{S}$

$\textsc{pop} :: \mathcal{S} \to \texttt{Item}$
remove the topmost item from stack $\mathcal{S}$

Stacks

Additional Operations

$\textsc{size} :: \mathcal{S} \rightarrow \texttt{size\_t}$
return the number of items in stack $\mathcal{S}$

$\textsc{peek} :: \mathcal{S} \rightarrow \texttt{Item}$
get the topmost item on stack $\mathcal{S}$, without removing it

a constructor and a destructor
to create a new empty stack, and
to release all resources of a stack

Stack ADTs

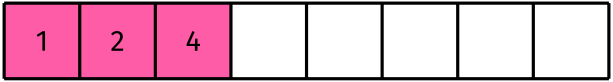
<stack.h>

```
typedef struct stack *Stack;

/** Create a new, empty Stack. */
Stack stack_new (void s);

/** Destroy a Stack, releasing its resources. */
void stack_drop (Stack s);

/** Add an item to the top of a Stack. */
void stack_push (Stack s, Item it);

/** Remove an item from the top of a Stack. */
Item stack_pop (Stack s);

/** Get the number of items in a Stack. */
size_t stack_size (Stack s);
```

Stack ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, ...
- Maintain a counter of the number of pushed items

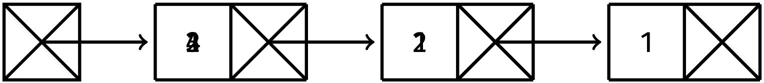


NEW PUSH (1) PUSH (2) PUSH (3) POP $\Rightarrow 3$ PUSH (4)

Stack ADTs

An Implementation using Linked Lists

- Add node to the front of the list on push
- Take node from the front of the list on pop



NEW PUSH (1) PUSH (2) PUSH (3) POP $\Rightarrow 3$ PUSH (4)

Stack ADTs

A Stack Client: Balancing Brackets

Sample input: `( [ { } ] )`

char	stack	check
		-
(	(	-
[	([	-
{	([{	-
}	([	{ = }
]	(	[=]
)		(=)
EOF		is empty

Stack ADTs

Sidenote: 'Prefix', 'Infix', 'Postfix'

$$2 + 3$$

infix

$$+\,2\,3$$

prefix

$$2\,3\,+$$

postfix

Many programming languages use infix operations.
Some (like Lisp) use prefix operations.
Some (like Forth, PostScript, *dc(1)*) use postfix operations.

Given an expression in postfix notation, return its value.

```
$ ./derpcalc "5 9 1 + 4 6 * * 2 + *"
1210
$ ./derpcalc "1 5 9 - 4 + *"
0
```

Stack ADTs

A Stack Client: a Postfix Expression Evaluator

- We use a stack!
- When we encounter a number:
  1. push it!
- When we encounter an operator:
  1. pop the two topmost numbers
  2. apply the operator to those numbers
  3. push the result back onto the stack
- At the end of input:
  1. print the last item on the stack

Queues

A queue is a collection of items, such that the first item to enter is the first item to leave:

First In, First Out (FIFO)

(Think queues of people, etc.)

- waiting lists
- call centres
- access to shared resources (e.g., printers)
- processes in a computer

Queues

Operations

$\textsc{enqueue} :: \mathcal{Q} \rightarrow \texttt{Item} \rightarrow \texttt{void}$
add a new item to the end of queue $\mathcal{Q}$

$\textsc{dequeue} :: \mathcal{Q} \rightarrow \texttt{Item}$
remove the item at the front of queue $\mathcal{Q}$

$\textsc{size} :: \mathcal{Q} \rightarrow \texttt{size\_t}$
return the number of items in queue $\mathcal{Q}$

$\textsc{peek} :: \mathcal{Q} \rightarrow \texttt{Item}$
get the frontmost item of queue $\mathcal{Q}$, without removing it

a constructor and a destructor
to create a new empty queue, and
to release all resources of a queue

Queue ADTs

<queue.h>

```
typedef struct queue *Queue;

/** Create a new, empty Queue. */
Queue queue_new (void q);

/** Destroy a Queue, releasing its resources. */
void queue_drop (Queue q);

/** Add an item to the front of a Queue. */
void queue_en (Queue q, Item it);

/** Remove an item from the end of a Queue. */
Item queue_de (Queue q);

/** Get the number of items in a Queue. */
size_t queue_size (Queue q);
```

Queue ADTs

An Implementation using Linked Lists

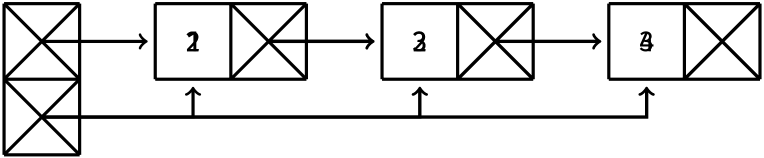
We need to add and remove items from opposite ends now!
We woulde either add or remove from the tail.
Can we do this efficiently? What do we need?

- If we only have a pointer to the head, **no**!
  We'd need to traverse the list to the tail every time.
- If we have a pointer to both head *and* tail,
  we don't have to traverse, and *adding* is efficient.
  (But not removing … why?)

Queue ADTs

An Implementation using Linked Lists

Add nodes to the end; take nodes from the front.

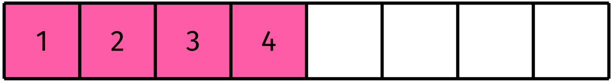


$\textsc{new}$ $\quad \textsc{en}\mathrm{Q}\,(1)$ $\quad \textsc{en}\mathrm{Q}\,(2)$ $\quad \textsc{en}\mathrm{Q}\,(3)$ $\quad \textsc{de}\mathrm{Q} \Rightarrow 1$ $\quad \textsc{en}\mathrm{Q}\,(4)$

Queue ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  ... some predefined fixed size
  ... dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items



NEW ENQ (1) ENQ (2) ENQ (3) DEQ $\Rightarrow 1$ ENQ (4)

Analysis and Testing

Analysis of Software

In COMP1911/1917/1511/1921,
the focus was on building software
(with unit testing for ‘quality control’)

In COMP2521, we focus more on analysis.
... which implies we have something to analyse.

Analysis of Software

Empirical vs Theoretical

Lots of the analysis we will do is **empirical**, executing and measuring, or **theoretical**, proving and deriving.

(We'll only be using proof-by-hand-waving...
COMP2111, COMP3141, COMP3153, COMP4141, COMP4161
go into formal methods in *much* more depth!)

Analysis of Software

What makes software 'good'?

correctness	returns expected result for all valid inputs
robustness	behaves 'sensibly' for non-valid inputs
efficiency	returns results reasonably quickly (even for large inputs)
clarity	clear code, easy to maintain/modify
consistency	interface is clear and consistent (API or GUI)

In this course, we're interested in correctness and efficiency.

A Moment of Robustness

Postel’s robustness principle:

Be conservative in what you do;
be liberal in what you accept from others

“defensive” programming

Analysing Effectiveness

We have two ways to determine effectiveness:

- empirical: **testing**, via program execution
  - devise a comprehensive set of test cases
  - compare actual results to expected results
- theoretical: **proof** of program correctness
  - define pre-conditions and post-conditions
  - establish that code maps from pre- to post-
  - (very loosely, Hoare logic)

Empirical vs Theoretical

For example: finding the maximum value in an unsorted array:

```
max = a[0];
for (i = 1; i < N; i++)
    if (a[i] > max) max = a[i];
```

What test cases should we use?

- max value is first, last, middle, ...
- values are positive, negative, mixed, same, ...

What are our pre- and post-conditions?

- pre: $\forall \texttt{j} \in [0 \cdots N-1], \mathrm{defined}\left(\texttt{a[j]}\right)$
- post: $\forall \texttt{j} \in [0 \cdots N-1], \texttt{max} \geq \texttt{a[j]}$

Empirical vs Analytical

Testing increases our confidence in correctness ...
better chosen test cases $\Rightarrow$ higher confidence
more thorough test cases $\Rightarrow$ higher confidence
...but cannot, in general, guarantee it!

Verification guarantees correctness:
any valid input will give a correct result,
but there's gaps; *e.g.*, how are invalid inputs treated?
(unless invalid input classes are included in pre-/post-conditions)

Dijkstra on Testing

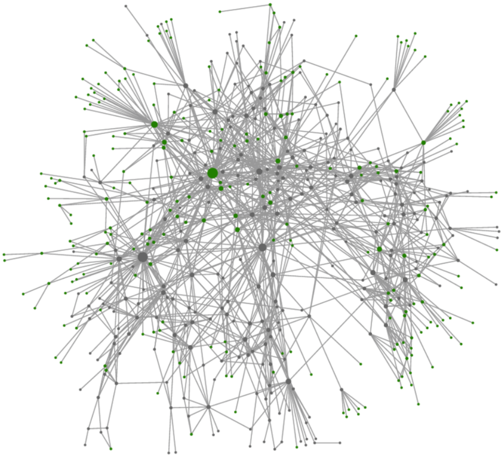
"Program testing can be used
to show the presence of bugs,
but never to show their absence!"

— from *Notes on Structured Programming*
by E. W. Dijkstra (EWD 249), April 1970

Software Verification

'seL4: Formal Verification of an OS Kernel', 2009
G. Klein, K. Elphinstone, G. Heiser *et al*;
UNSW/NICTA (now Data61 at CSIRO)

~9 kLoC C … ~55 kLoP, ~11 py



Testing Approaches

The “Big Bang” approach

The “Big Bang” approach:

- you write the entire program!
- then you design and maybe even run some test cases!

This is terrible!

Testing Approaches

Test-Driven Development

Test-Driven Development (TDD), or “test-first”:

- write the tests for a function,
- then, write the function,
- then, test the function!
- integrate that with other tested functions.
- rinse and repeat until you have constructed and tested an entire program

Testing Approaches

Test-Driven Development

Regression testing:

- Keep a comprehensive test suite!
- Always run all your tests; don't throw tests away!
- Re-run all your tests after changing your system!

Testing Approaches

Create, Mutate, Inspect [1]

Every test should follow a simple pattern:

create
set up a well-known environment

mutate
make *one* well-known change

inspect
check the results

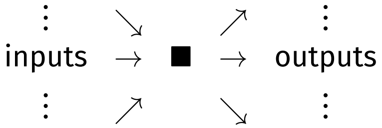
[1] I'm sure there's a better name for this.

White-Box and Black-Box Testing

Black-Box Testing

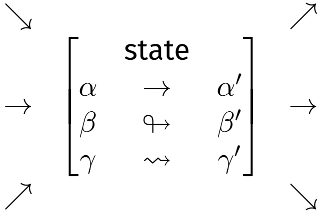
Black-box testing
tests code from the outside...

- checks specified behaviour
- expected input to expected output
- uses *only* the interface!
  ... implementation-agnostic



White-box testing
tests code from the inside...

- checks code structure
  and structure consistency
- checks internal functions
- tests rely on a particular
  implementation



Testing with *assert(3)*

Useful while developing, testing, debugging…
but *not* in production code!

assert(3) aborts the program;
emits error message useful to a programmer,
but not to the user of the application.
(e.g., those *gedit* errors)

Use *exception handlers* in production code
to terminate gracefully with a sensible error message

COMP2521 19T0
Week 2, Tuesday: Algorithms!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

algorithm analysis
complexity
recursion

Complexity

Problems, Algorithms, Programs, Processes

problem	something that needs to be solved
algorithm	well-defined instructions to solve the problem
program	implementation of the algorithm in a particular programming language
process	an instance of a program being executed

Analysis of Software

What makes software ‘good’?

correctness	returns expected result for all valid inputs
robustness	behaves ‘sensibly’ for non-valid inputs
efficiency	returns results reasonably quickly (even for large inputs)
clarity	clear code, easy to maintain/modify
consistency	interface is clear and consistent (API or GUI)

lecture 2: correctness.
today: efficiency.

Algorithm Efficiency

- algorithm runtime tends to be a function of input size
- often difficult to determine the average run time
- we tend to focus on asymptotic worst-case execution time
  ... easier to analyse!
  ... crucial to many applications: finance, robotics, games, ...

Algorithm Efficiency

By far, the *most important* determinant of a program's efficiency.

Small, often constant-factor speedups from

- operating systems,
- compilers,
- hardware,
- implementation details

More important: an efficient algorithm.

Algorithm Efficiency

Determining Theoretically and Empirically

Design

- complexity theory!

Implementation and Testing

- measure its properties!
  ...run-time using *time(1)*
  ...profiling tools like *gprof(1)*
  ...performance counters like *pmc(3)*, *hwpmc(4)*

Algorithm Efficiency, Empirically

1. Write a program that implements an algorithm.
   ... which may not always be possible!
2. Run the program with inputs of varying size and composition.
   ... which may not always be possible!
   ... choosing good inputs is *extremely* important
3. Measure the actual runtime.
   ... which may not always be possible (or easy)!
   ... similar runtime environments required
4. Plot the results.
   (Optionally, be confused about the results.)

Algorithm Efficiency, Theoretically

- Don't necessarily use an implementation!
  ... Use pseudocode or something close to it.
- Characterise efficiency as a function of inputs.
- Take into account *all possible* inputs
- Generally produces a value that is environment-agnostic
  ... allowing us to evaluate comparative efficiency of algorithms

Timing Execution

Absolute times will differ
between machines, between languages
...so we're not interested in absolute time.

We are interested in the *relative* change
as the problem size increases

Timing Execution

time(1)

We can use the *time(1)* command to measure execution time (and several other interesting properties).

There are two common implementations:
one built-into the shell,
and one at `/usr/bin/time`
both are OK for our purposes.

Timing Execution

time(1) output

```
$ time ./prog

./prog 0.01s user 0.02s system 97% cpu 0.028 total
0k shared 0k local 11k max 0+3280 faults
13+0 in 0+0+0 out 4 vcsw 4 ivcsw
```

Most of this information isn't interesting to us.
The *user* time is!

Redirect input into your program:

```
$ time ./prog < input > /dev/null
$ ./mkinput | time ./prog > /dev/null
```

Empirical Analysis

Analysis of a Linear Search

Time a linear search with different-sized inputs —

```
$ ./gen 100  A | time ./linear > /dev/null
$ ./gen 1000 A | time ./linear > /dev/null
```

(repeat a number of times and average)
What is the relation between *input size* and *user time*?

If I know my algorithm is quadratic,
and I know that for a dataset of 1000 items,
it takes 1.2 seconds to run …

- how long for 2000? 4.8 seconds
- how long for 10,000? 120 seconds (2 mins)
- how long for 100,000? 12000 seconds (3.3 hours)
- how long for 1,000,000? 1200000 seconds (13.9 days)

Searching in a Sorted Array

Given an array a of n elements,
where for any pair of indices i, j,
$i \leq j < n$ implies $a[i] \leq a[j]$
search for an element e in the array.

```
int a[N];         // array with N items
bool found = 0;
bool finished = false;
int i = 0;
while ((i < N) && (! found)) {&& (! finished)) {
    found = (a[i] == e);
    finished = (e < a[i]);
    i++;
}
```

Searching in a Sorted Array

Complexity Analysis

How many comparisons do we need
for an array of size N?

Best case: $t(N) \sim O(1)$

Worst case: $t(N) \sim O(N)$

Average case: $t(N) \sim O(N/2)\ O(N)$

Still a *linear* algorithm!
Can we do better?

Searching in a Sorted Array

Exploiting a Binary Search

Let's start in the middle.

- If e==$a[N/2]$, we found e; we're done!
- Otherwise, we split the array:
  ... if $e < a[N/2]$, we search the left half ($a[0]$ to $a[(N/2) - 1)]$)
  ... if $e > a[N/2]$, we search the right half ($a[(N/2) + 1)]$ to $a[N - 1]$)

Why Binary Search?

How many comparisons do we need
for an array of size N?

Best case: $t(n) \sim O(1)$

Worst case:

$$t(N) = 1 + t(\frac{N}{2})$$
$$t(N) = \log_2 N + 1$$
$$t(N) \sim O\left(\log N\right)$$

Algorithm Efficiency, Theoretically

Cost Modelling of Primitive Operations

In C, a line of code can do *lots* of things!

We're interested in 'primitive operations', though:
operations that can execute in one step,
which we can think of as hardware instructions.

(In COMP1521, we use the MIPS instruction set;
we get a feel for the primitive nature of instructions.)

Our cost-modelling will roughly follow the same lines,
but strictly we don't need to consider how long a primop takes.
We'll see why in a moment.

Expressing Complexity Classes

We express complexity using a range of
complexity models and *complexity classes*.

Most commonly, time complexity,
for which we use Big-O notation,
representing asymptotic worst-case time complexity.
I'll sometimes call this WCET.

Sometimes, space complexity too.
(Not so much in this course, but useful!)

Worst-Case Execption Time

```
ssize_t lsearch (int a[], size_t n, int key)
{
    for (size_t i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

$$1 + (n + 1) + n(1+$$
$$2 + ($$
$$0)$$
$$) + 1$$

- When does the worst case occur? … key $\notin$ a
- How many data comparisons were made? … n
- What is the worst-case cost? … $3 + 4n$ … $O(n)$

Growth rate is not affected (much, usually)
by constant factors or lower-order terms …
so we discard them.

$3 + 4n$ becomes $O\left(n\right)$ — a *linear* function
$3 + 4n + 3n^2$ becomes $O\left(n^2\right)$ — a *quadratic* function

These are an *intrinsic property* of the algorithm.

Complexity Theory

Discarding Less Obvious Terms

If a is time taken by the fastest primitive operation,
and b is time taken by the slowest primitive operation,
and $t(n)$ is the WCET of our algorithm …

$$a \cdot (3 + 4n) \leq t(n) \leq b \cdot (3 + 4n)$$

Where does the log-base go?

$$O(\log_2 n) \equiv O(\log_3 n) \equiv \cdots$$

(since $\log_b(a) \times \log_a(n) = \log_b(n)$)

Complexity Classes

All The Mathematics!

$f(n)$ is $O(g(n))$
if $f(n)$ is asymptotically less than or equal to $g(n)$
$f(n)$ is $\Omega(g(n))$
if $f(n)$ is asymptotically greater than or equal to $g(n)$
$f(n)$ is $\Theta(g(n))$
if $f(n)$ is asymptotically equal to $g(n)$

Given $f(n)$ and $g(n)$, we say $f(n)$ is $O(g(n))$
if we have positive constants c and n_0 such that

$$\forall n \geq n_0, f(n) \leq c \cdot g(n)$$

Complexity Theory

Some Common Big-O Functions

constant	$O(1)$ ...constant-time execution, independent of the input size.
logarithmic	$O(\log n)$ ...some divide-and-conquer algorithms with trivial split/recombine operations
linear	$O(n)$ ...every element of the input has to be processed (in a straightforward way)
n-log-n	$O(n \log n)$ ...divide-and-conquer algorithms, where split/recombine is proportional to input
quadratic	$O(n^2)$ ...compute every input with every other input ...problematic for large inputs!
cubic	$O(n^3)$ ... misery
factorial	$O(n!)$ ... real misery
exponential	$O(2^n)$ ... running forever is fine, right?

Complexity Theory

Complexity Classes

tractable — have a polynomial-time ('P') algorithm
... polynomial worst-case performance (e.g., $O(n^2)$)
... (useful and usable in practical applications)

intractable — no tractable algorithm exists (usually 'NP'[1])
... worse than polynomial performance (e.g., $O(2^n)$)
... (feasible only for small n)

non-computable — no algorithm exists (or can exist)

[1] nondeterministic polynomial time, on a theoretical Turing Machine

Thinking about Complexity

What would be the time complexity of
inserting an element at the beginning of

... a linked list?
... an array?

What about the end?

What if it’s ordered?

Recursion

Recursive Functions

Sometimes, problems can be expressed in terms of a simpler instance of the same problem.

- $1! = 1$
- $2! = 2 \times 1$
- $3! = 3 \times 2 \times 1$
- $\vdots$
- $(n-1)! = (n-1) \times \cdots \times 3 \times 2 \times 1$
- $(n)! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$

$$n! = (n-1)! \times n$$

Recursive Functions

Solving problems recursively in a program involves developing a program that calls itself.

base case
: (or *stopping case*)
no recursive call is needed

recursive case
: calls the function on a smaller version of the problem

Iteration vs Recursion

```
int factorial (int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

```
int factorial (int n) {
    if (n == 1) return 1;
    else return n * factorial (n - 1);
}
```

Too Many Damn Rabbits

Recursive code can be horribly inefficient!
2^n calls is $O(k^n)$ time — exponential!

```
switch (n) {
case 0:  return 0;
case 1:  return 1;
default: return fib (n - 1) + fib (n - 2);
}
```

COMP2521 19T0
Week 2, Thursday: Trees!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

recursion
trees

Recursion

Recursive Linked Lists

A linked list can be described recursively!

```
struct node {
    Item item;
    node *next;
};
```

"... this value, and the rest of the values"

Recursive Linked Lists

Functions Amenable to Recursion (I)

```
size_t list_length (node *curr)
{
    if (curr == NULL) return 0;                 // base case
    return 1 + list_length (curr->next);        // recursive case
}
```

```
int int_list_sum (intnode *curr)
{
    if (curr == NULL) return 0;                 // base case
    return curr->item +
        int_list_sum (curr->next);              // recursive case
}
```

Recursive Linked Lists

Functions Amenable to Recursion (II)

```
void int_list_print (node *curr)
{
    if (curr == NULL) return;
    printf ("%d\n", curr->item);
    int_list_print (curr->next);
}

void int_list_print_reverse (node *curr)
{
    if (curr == NULL) return;
    int_list_print_reverse (curr->next);
    printf ("%d\n", curr->item);
}
```

Divide-and-Conquer, Recursively

REMINDER divide and conquer algorithms tend to:

- divide the input into parts,
- solve the problem on the parts recursively, then
- combine the results into an overall solution.

(This is a common 'big-data' approach: map-reduce.)

(("There's no such thing as 'big data'."))

Divide-and-Conquer, Recursively

Maximum of an Unsorted Array (I)

Iteratively:

```
int array_max (int a[], size_t n)
{
    int max = a[0];
    for (size_t i = 0; i < n; i++)
        if (a[i] > max) max = a[i];
    return max;
}
```

complexity: $O(n)$

Divide-and-Conquer, Recursively

Maximum of an Unsorted Array (II)

Recursively:

```
int array_max (int a[], size_t l, size_t r)
{
    if (l == r) return a[l];
    int m = (l + r) / 2;
    int m1 = array_max (a, l, m);
    int m2 = array_max (a, m + 1, r);
    return (m1 < m2) ? m2 : m1;
}
```

complexity: ...

Divide-and-Conquer, Recursively

Maximum of an Unsorted Array (IIa)

How many calls of `array_max` are necessary?

for length 1, $c(1) = 1$
for length $n > 1$, $c(n) = c(\frac{n}{2}) + c(\frac{n}{2}) + 1$
... overall $c(n) = 2n - 1$ calls

in each recursive call, we do $O(1)$ steps.

$$\implies O(n)$$

Binary Search, Revisited

Recursive Binary Search (I)

Iteratively:

```
ssize_t binary_search (int a[], size_t n, int key)
{
    size_t lo = 0, hi = n - 1;
    while (hi >= lo) {
        size_t mid = (lo + hi) / 2;
        if (a[mid] == key) return mid;
        if (a[mid]  > key) hi = mid - 1;
        if (a[mid] <  key) lo = mid + 1;
    }
    return -1;
}
```

complexity: $O(\log n)$

Binary Search, Revisited

Recursive Binary Search (II)

Recursively:

```
ssize_t binary_search (int a[], size_t n, int key)
{
    return binary_search_do (a, 0, n - 1, key);
}

ssize_t binary_search_do (int a[], size_t lo, size_t hi, int key)
{
    if (lo > hi) return -1;
    size_t mid = (lo + hi) / 2;
    if (a[mid] == key) return mid;
    if (a[mid]  > key) return binary_search_do (a, lo, mid - 1, key);
    if (a[mid] <  key) return binary_search_do (a, mid + 1, hi, key);
    assert (!"unreachable");
}
```

complexity: $O(\log n)$

Trees

Searching

Search is a critical operation, e.g.

- looking up a name in a phone book
- selecting records in databases
- searching for pages on the web

Characteristics of the search problem:

- typically, very large amount of data (very many items)
- query specified by keys (search terms)
- effective keys identify a small proportion of data

Searching

The Search Problem (I)

We'll abstract the problem to:
a large collection of *items*,
each containing a *key* and other data
(We can think of these as
'key/data' or 'key/value' pairs.)

```
typedef ⟨···⟩ Key;
typedef struct {
    Key key;
    ⟨··· data ···⟩
}Item;
```

Searching

The Search Problem (II)

The search problem:

input a key value

output item(s) containing that key

Common variations:

- keys are unique; key value matches 0 or 1 items
- multiple keys in search, items containing any key
- multiple keys in search/item, items containing all keys

We assume: keys are unique, each item has one key.
Cheap, easy gains from searching sorted data.

Searching

The Search Problem (III)

Maintaining sorted sequences is hard...
inserting into a sorted sequence is a two-step problem.

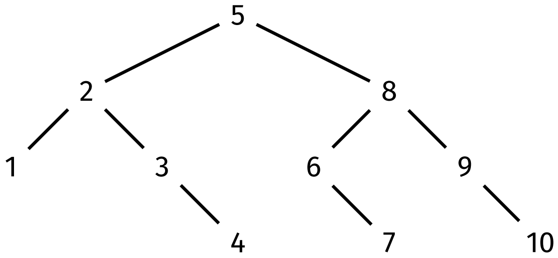
array search $O(\log n)$, insert $O(n)$
... we have to move all the items along

linked list search $O(n)$, insert $O(1)$
... search is *always* linear

Can we do better?

Search Trees

1	2	3	4	5	6	7	8	9	10



Trees are branched data structures,
consisting of nodes and edges, with no cycles.

Each node contains a value.
Each node has edges to $\leq k$ other nodes.
For now, $k = 2$ — binary trees

Trees can be viewed as a set of nested structures:
each node has k (possibly empty) subtrees.

A node is a parent if it has outgoing edges.
A node is a child if it has incoming edges.
The root node has no parents.
A leaf node has no children.

A node's depth or level is
the number of edges from the root to that node.
The root node has depth 0;
all other nodes have one more than their parent's depth

Trees

Terminology of Trees (III)

For a given number of nodes, a tree is said to be **balanced** if it has minimal height, and **degenerate** if it has maximal height.

A k**-ary tree**'s internal nodes have k children.
A tree is **ordered** if data/keys in nodes are constrained.

Trees

Uses of Trees

- representing hierarchical data structures
  (e.g., expressions in a programming language)
- efficient search
  (e.g., in sets, symbol tables)

Binary Trees

For much of the course,
we'll look at binary trees (where $k = 2$).

Binary trees are either empty,
or are a node with two subtrees,
where each node has a value,
and the subtrees are binary trees.

$$\begin{array}{lcl} \text{BTree} & := & \text{Empty} \\ & | & \text{Node} \quad x \quad \text{BTree}\, l \quad \text{BTree}\, r \end{array}$$

Binary Trees

Properties

A binary tree with n nodes has a height of
at most $n - 1$, if degenerate; or
at least $\lfloor \log_2 n \rfloor$, if balanced.

Cost for insertion:
balanced $O(\log_2 n)$, degenerate $O(n)$
(we always traverse the height of the tree)

Cost for search/deletion:
balanced $O(\log_2 n)$, degenerate $O(n)$
(worst case, key $\notin \tau$; traverse the height)

Binary Search Trees

A binary tree!

For all nodes in the tree:
the values in the left subtree are less than the node value
the values in the right subtree are greater than the node value

Structure tends to be determined by order of insertion:
$[4, 2, 1, 3, 6, 5, 7]$ vs $[6, 5, 2, 1, 3, 4, 7]$

Exercise: Happy Little Trees

Starting with an initially-empty binary search tree ... show the tree resulting from inserting values in the order given, and give its resulting height —

1. $[4, 2, 6, 5, 1, 7, 3]$
2. $[5, 3, 6, 2, 4, 7, 1]$
3. $[1, 2, 3, 4, 5, 6, 7]$

```
struct btree_node {
    Item item;
    btree_node *left;
    btree_node *right;
};
```

As before: the empty tree is `NULL`.

Binary Search Trees

Implementation in C: Search

```
// return the node if found, or NULL otherwise
btree_node *btree_search (btree_node *tree, Item key)
{
    if (tree == NULL) return NULL;
    int cmp = item_cmp (key, tree->item);
    if (cmp == 0) return tree;
    if (cmp <  0) return btree_search (tree->left, key);
    if (cmp  > 0) return btree_search (tree->right, key);
}
```

EXERCISE Try writing an iterative version.

Binary Search Trees

Implementation in C: Insertion

We're (recursively) inserting value v into tree τ.

Cases:

- τ empty
  $\Rightarrow$ make a new node with v as the root of the new tree
- the root of τ contains v
  $\Rightarrow$ tree unchanged (assuming no duplicates)
- $v < \tau$`->item`
  $\Rightarrow$ do insertion into τ`->left`
- $v > \tau$`->item`
  $\Rightarrow$ do insertion into τ`->right`

EXERCISE Try writing an iterative version.

Binary Search Trees

Implementation in C: Dimensions

- `btree_size` $:: \mathrm{BTree} \rightarrow$ `size`
  return the number of nodes in a tree
- `btree_height` $:: \mathrm{BTree} \rightarrow$ `size`
  return the height of a tree

Binary Tree Traversals

'serialisation' of a structure:
flattening it in a well-defined way,
such that the original structure can be recovered

Depth-first:

- pre-order traversal (NLR)
  ... visit node, then left subtree, then right subtree
- in-order traversal (LNR)
  ... visit left subtree, then node, then right subtree
- post-order traversal (LRN)
  ... visit left subtree, then right subtree, then node

Breadth-first:

- level-order traversal
  ... visit node, then all its children

Binary Search Trees

Deletion

Insertion is easy!
find location, create node, link parent
Deletion is much harder!
find node, unlink and delete, ...?

One option: don't delete nodes :-)
instead, just mark them as deleted, and ignore them

Otherwise, we must *promote* a child (carefully).
A child with no subtrees: drop.
A child with one subtree: promote that subtree.
A child with two subtrees: ...
replace node with leftmost of right subtree

COMP2521 19T0
Week 3, Tuesday: Graphic Content (I)!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

priority queues
graph fundamentals

Assignment 1: Textbuffer

Pitfalls and Pointers (I)

- Strings in C are pointers to arrays of `char`acters;
  following the last character is a `NUL` terminator: `'\0'`
  there won't be multiple `NUL` characters in a string
- To store `"hello\n"`: **7 bytes** —
  ... {`'h'`, `'e'`, `'l'`, `'l'`, `'o'`, `'\n'`, `'\0'`}
  ... referring to the string `"\0"` is redundant
- `sizeof` is a *static* property; string length is a *dynamic* property.
  ... in *(e.g.,)* `textbuffer_new`:
  ... `sizeof text` $=$ `sizeof (char *)` $= 4$
  ... `sizeof *text` $=$ `sizeof (char)` $= 1$
  ... use *strlen(3)* or *strnlen(3)* or similar

Assignment 1: Textbuffer

Pitfalls and Pointers (II)

- Making a (heap-allocated, mutable) copy of a string?
  … *strdup(3)*, *strndup(3)* get it right — did you?
- Splitting a string using *strsep(3)* or *strtok(3)*?
  … do you know what's going on?
- HINT read the forum answers!
  … they tend to be filled with all kinds of useful wisdom
- ANTI-HINT the challenge exercises are *challenging*
  … you will need to do your own reading and thinking
  … undo/redo hint: see week01thu lecture
  … diff hint: Levenshtein, but is it optimal?
- Cryptic crossword hint: 'shaken players shift the load'.

Priority Queues

Priority

Not all queues are created equal…
ever been to a hospital?

FIFO doesn't always cut it!
Sometimes, we need to process
in order of *key or priority*.

Priority Queues (PQueues or PQs)
provide this with
altered enqueue and dequeue.

Priority Queue

Operations

$\textsc{EnPQueue} :: \mathcal{Q}' \to (\texttt{Item}, \texttt{prio}) \to \texttt{void}$

join or requeue an item with a priority to pqueue $\mathcal{Q}'$

$\textsc{DePQueue} :: \mathcal{Q}' \to \texttt{Item}$

remove the item with highest priority from pqueue $\mathcal{Q}'$
(potentially including the priority; $\to (\texttt{Item}, \texttt{prio})$)

Priority Queue

<pqueue.h>

```
typedef struct pqueue *PQueue;
typedef int pq_prio;

/** Create a new, empty PQueue. */
PQueue pqueue_new (void q);

/** Destroy a PQueue, releasing its resources. */
void pqueue_drop (PQueue pq);

/** Add an item with a priority to a PQueue. */
void pqueue_en (PQueue pq, Item it, pq_prio prio);

/** Remove the highest-priority item from a PQueue. */
Item pqueue_de (PQueue pq, pq_prio *prio);

/** Get the number of items in a PQueue. */
size_t pqueue_size (PQueue pq);
```

Priority Queue

Implementations

ordered array or ordered list:
insert $O(n)$, delete $O(1)$

unordered array or unordered list:
insert $O(1)$, delete $O(n)$

there must be a better way!

Heaps of Fun

Heaps are a good solution.
Commonly viewed as trees;
commonly implemented with arrays.

Two important properties:
heap order property,
a 'top-to-bottom' ordering of values;
complete tree property,
every level is as filled as possible

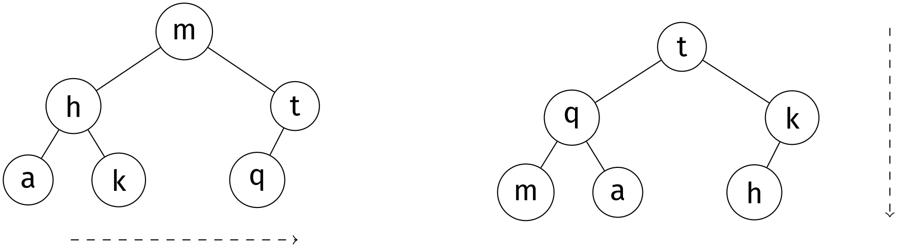
Heaps of Fun

Heap-Order Property

Binary search trees have left-to-right ordering.

Heaps have a top-to-bottom ordering:
for all nodes, both subtrees are $\leq$ the root
(*i.e.*, the root contains the largest value)

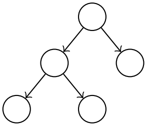
Inserting $[\mathrm{m}, \mathrm{t}, \mathrm{h}, \mathrm{q}, \mathrm{a}, \mathrm{k}]$ into a BST and heap:



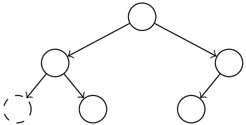
Heaps of Fun

Complete Tree Property

Heaps are *complete trees*:
every level is filled before adding nodes to the next level
nodes in a given level are filled left-to-right, with no breaks



complete



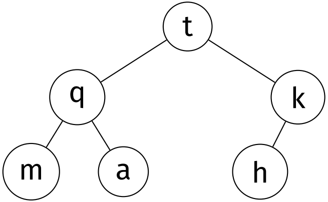
incomplete

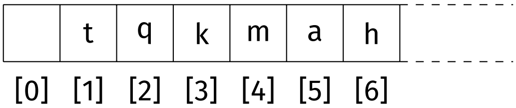
Heap Implementation

BSTs are typically implemented as linked data structures.

Heaps *can* be implemented as linked structures...
but are more commonly implemented as arrays.
complete tree $\Rightarrow$ array implementation

$$\textsc{left}\,(i) := 2i \qquad \textsc{right}\,(i) := 2i+1 \qquad \textsc{parent}\,(i) := i/2$$



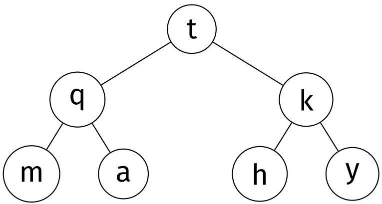


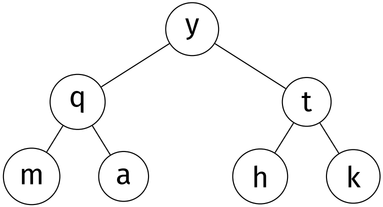
Heap Implementation

Insertion into an Array Heap (I)

Insertion is a two-step process:

1. add new element at the bottom-most, right-most position
(to ensure it is still a complete tree)
2. reorganise values along the path to the root
(to ensure it is still maintains heap order)





Heap Implementation

Insertion into an Array Heap (II)

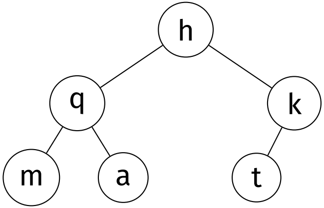
```
// move value at a[k] to correct position
void heap_fixup (Item a[], size_t k)
{
    while (k > 1 && item_cmp (a[k/2], a[k]) < 0) {
        swap (a, k, k/2);
        k /= 2; // integer division!
    }
}
```

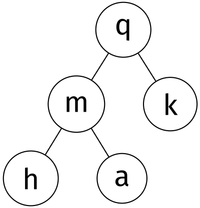
Heap Implementation

Deletion from an Array Heap (I)

Deletion is a three-step process:

1. swap root value with bottom-most, right-most value
2. remove bottom-most, right-most value
   (to ensure it is still a complete tree)
3. reorganise values along path from root
   (to ensure it is still maintains heap order)





Heap Implementation

Deletion from an Array Heap (II)

```
// move value at a[k] to correct position
void heap_fixdown (Item a[], size_t k)
{
    while (2 * k <= N) {
        size_t j = 2 * k; // choose greater child
        if (j < N && item_cmp (a[j], a[j+1]) < 0)
            j++;
        if (item_cmp (a[k], a[j]) >= 0)
            break;
        swap (a, k, j);
        k = j;
    }
}
```

Heap Complexity

Lots of work, surely?

height: always $\lfloor \log_2 n \rfloor$ (complete!)

insert: fixup is $O\left(\log_2 n\right)$
delete: fixdown is $O\left(\log_2 n\right)$

... worth it!

Now you try!

Exercise: Heaps of Fun!

Show the construction of the max-heap produced by inserting

$$[H, E, A, P, S, F, U, N]$$

Delete an item. What does the heap look like now?

Delete another item. What does the heap look like now?

Graph Fundamentals

Collections of Related Things

Up to this point, we've seen a few collection types...

lists: a *linear* sequence of items
each node knows about its next node
trees: a *branched* hierarchy of items
each node knows about its child node(s)

what if we want something more general?
...each node knows about its *related* nodes

Collections of Related Things

... Related Nodes? (I)

Many applications need to model relationships between items.

... on a map: cities, connected by roads
... on the Web: pages, connected by hyperlinks
... in a game: states, connected by legal moves
... in a social network: people, connected by friendships
... in scheduling: tasks, connected by constraints
... in circuits: components, connected by traces
... in networking: computers, connected by cables
... in programs: functions, connected by calls
... etc. etc. etc.

Collections of Related Things

... Related Nodes? (II)

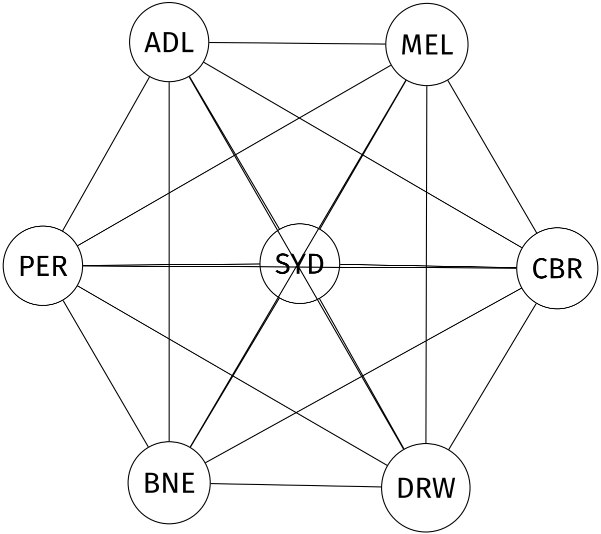
Questions we could answer with a graph:

- what items are connected? how?
- are the items fully connected?
- is there a way to get from A to B?
  what's the best way? what's the cheapest way?
- in general, what can we reach from A?
- is there a path that lets me visit all items?
- can we form a tree linking all vertices?
- are two graphs "equivalent"?

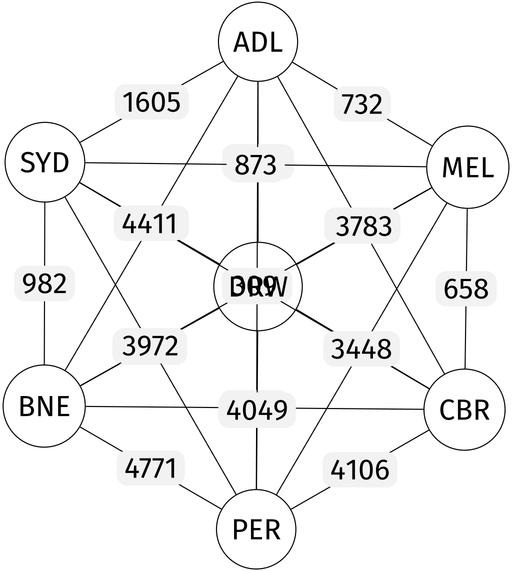
Road Distances

	ADL	BNE	CBR	DRW	MEL	PER	SYD
ADL	—	2055	1390	3051	732	2716	1605
BNE	2055	—	1291	3429	1671	4771	982
CBR	1390	1291	—	4441	658	4106	309
DRW	3051	3429	4441	—	3783	4049	4411
MEL	732	1671	658	3783	—	3448	873
PER	2716	4771	4106	4049	3448	—	3972
SYD	1605	982	309	4411	873	3972	—

Road Distances



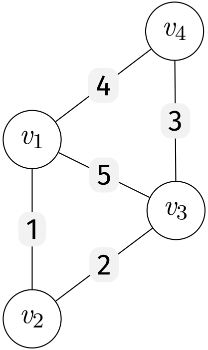
Road Distances



Graphs

A graph G is a set of vertices V and edges E.

$$E := \{(v, w) | v, w \in V, (v, w) \in V \times V\}$$

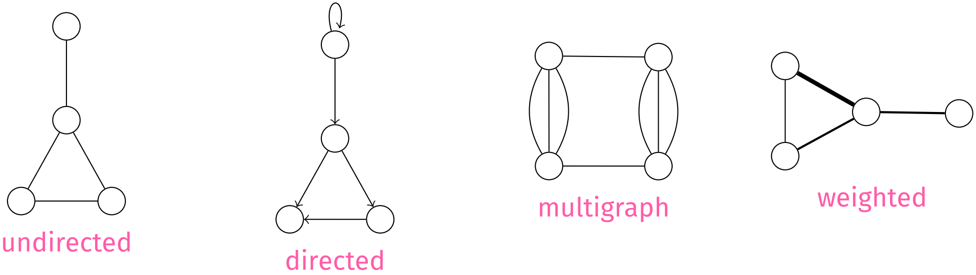


$$V = \{v_1, v_2, v_3, v_4\}$$

$$E = \left\{ \begin{array}{ccc} e_1 & := & (v_1, v_2), \\ e_2 & := & (v_2, v_3), \\ e_3 & := & (v_3, v_4), \\ e_4 & := & (v_1, v_4), \\ e_5 & := & (v_1, v_3) \end{array} \right\}$$

Graphs

Types of Graphs



Directed Graphs

If edges in a graph are directed,
the graph is a directed graph or digraph.

The edge $(v, w) \neq (w, v)$.
A digraph with V vertices can have at most V^2 edges.
Digraphs can have self loops ($v \to v$)

Unless otherwise specified,
graphs are undirected in this course.

Multigraphs and Weighted Graphs

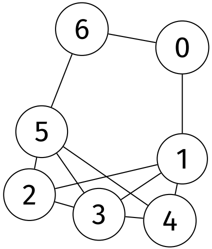
Multi-Graphs...
allow multiple edges between two vertices
(e.g., callgraphs; maps)

Weighted Graphs...
each edge has an associated weight
(e.g., maps; networks)

Simple Graphs

At this point,
we'll only consider **simple graphs**:

- a set of vertices
- a set of undirected edges
- no self loops
- no parallel edges



$|V| = 7; |E| = 11.$

How many edges can a
7-vertex simple graph have?
$7 \times (7-1)/2 = 21$

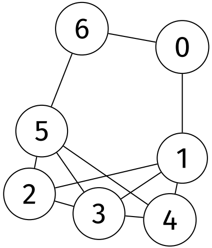
Graph Terminology
(I)

For a simple graph:

$$|E| \leq (|V| \times (|V| - 1))/2$$

- if $|E|$ closer to $|V|^2$, *dense*
- if $|E|$ closer to $|V|$, *sparse*
- if $|E| = 0$, we have a set

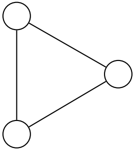
These properties affect our choice of representation and algorithms.



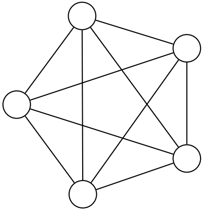
$|V| = 7$; $|E| = 11$.

Graph Terminology
(II)

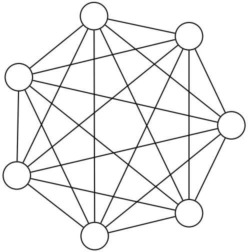
A complete graph is a graph where
every vertex is connected to all other vertices:
$|E| = (|V| \times (|V| - 1))/2$



K_3



K_5



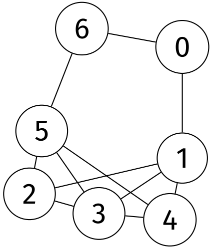
K_7

Graph Terminology (III)

A vertex v has degree $\deg(v)$
of the number of edges
incident on that vertex.

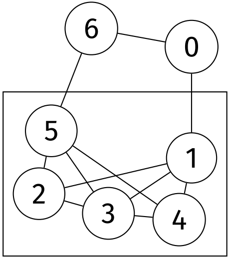
$\deg(v) = 0$ — an isolated vertex
$\deg(v) = 1$ — a pendant vertex

Two vertices v and w are adjacent
if an edge $e := (v, w)$ connects them;
we say e is incident on v and w



Graph Terminology (IV)

A subgraph is a subset of vertices and associated edges

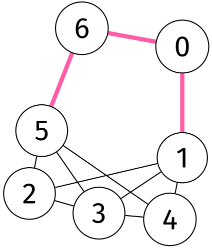


Graph Terminology (V)

A path is
a sequence of
vertices and edges
... $1, 0, 6, 5$

a path is simple
if it has no repeating vertices

a path is a cycle
if it is simple *except*
for its first and last vertex,
which are the same.



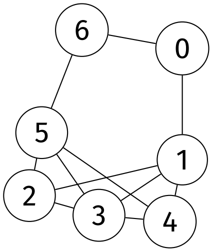
Graph Terminology

(VI)

A connected graph
has a path from every vertex
to every other vertex

A connected graph
with no cycles is a tree.

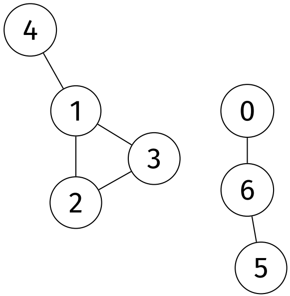
A tree has exactly one path
between each pair of vertices.



(not a tree)

Graph Terminology (VII)

A graph that is not connected consists of a set of connected components: maximally connected subgraphs

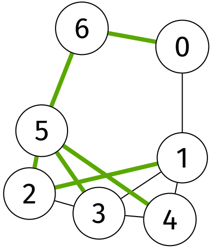


Graph Terminology
(VIII)

A spanning tree of a graph
is a subgraph that
contains all its vertices
and is a single tree

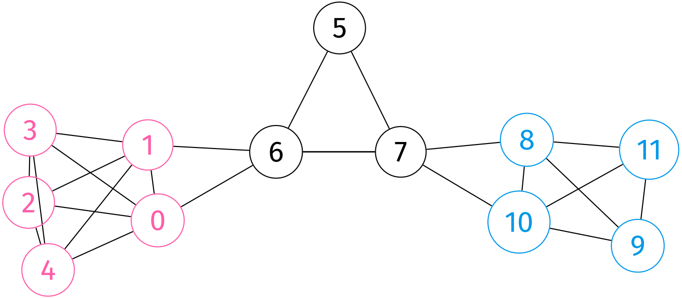
A spanning forest of a graph
is a subgraph that
contains all its vertices
and is a set of trees

There isn't necessarily *only* one
spanning tree/forest for a graph.



Graph Terminology
(IX)

A clique is a complete subgraph.



COMP2521 19T0
Week 3, Thursday: Graphic Content (II)!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

graph representation
graph search

Graph Representation

Graphs

What do we need to represent?

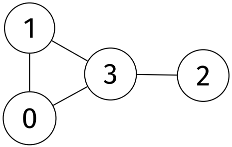
A graph G is a set of vertices $V := \{v_1, \cdots, v_n\}$,
and a set of edges $E := \{(v, w) \mid v, w \in V;\ (v, w) \in V \times V\}$.
Directed graphs: $(v, w) \neq (w, v)$.
Weighted graphs: $E := \{(v, w, \sigma)\}$.
Multigraphs: E is a list, not a set.

What operations do we need to support?

create/destroy graph;
add/remove vertices, edges;
get #vertices, #edges;

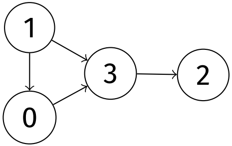
Adjacency Matrices

A $|V| \times |V|$ matrix; each cell represents an edge.



$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

undirected



$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

directed

Adjacency Matrices

Advantages and Disadvantages

Advantages

- Easy to implement!
  two-dimensional array of
  `bool`/`int`/`float`/...
- Works for:
  graphs! digraphs!
  weighted graphs!
  (unweighted) multigraphs!
- Efficient!
  $O(1)$ edge-insert, edge-delete
  $O(1)$ is-adjacent

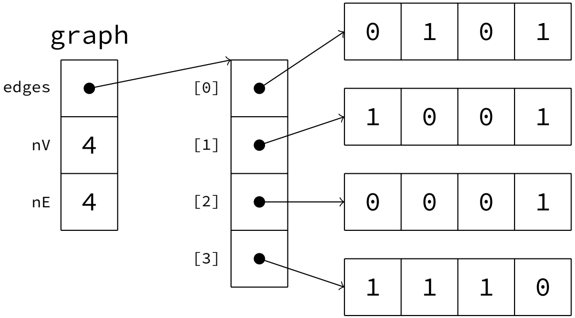
Disadvantages

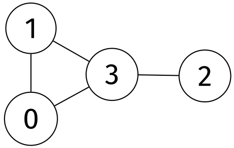
- Huge space overheads!
  V^2 cells of some type
  sparse graph $\Rightarrow$ wasted space!
  undirectd graph $\Rightarrow$ wasted space!
- Inefficient!
  $O(V^2)$ initialisation
  $O(V^2)$ vertex-insert/-delete

Adjacency Matrices

Implementation in C

```c
struct graph {
    size_t nV, nE;
    bool **matrix;
};
```





Exercise: Time Complexity

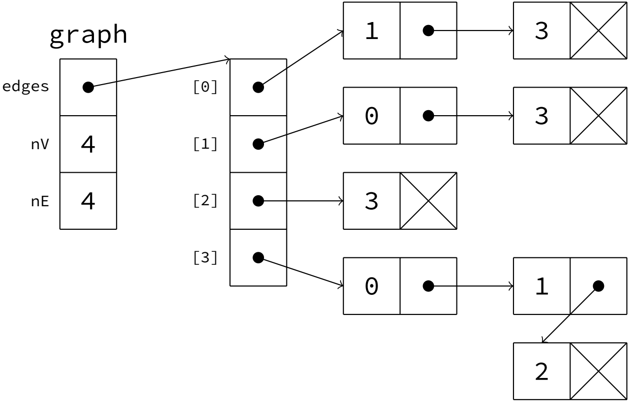
Given an adjacency matrix representation,
find the time complexity, and implement, these functions

- `bool graph_adjacent (Graph g, vertex v, vertex w);`
  ... returns true if vertices v and w are connected, false otherwise
- `size_t graph_degree (Graph g, vertex v);`
  ... return the degree of a vertex v

Adjacency Lists

Implementation in C

```
typedef
    struct adjnode
    adjnode;
struct graph {
    size_t nV, nE;
    adjnode **edges;
};
struct adjnode {
    vertex w;
    adjnode *next;
};
```



Comparison

Adjacency Lists vs Adjacency Matrices

- Space: matrix: V^2; adjlist: $V + E$
- Initialise: matrix: V^2, adjlist: V
- Destroy: matrix: V, adjlist: E
- Insert edge: matrix 1, adjlist: V
- Find/remove edge: matrix: 1, adjlist: V
- is isolated? matrix: V, adjlist: 1
- Degree: matrix: V, adjlist: E
- is adjacent? matrix: 1, adjlist: V

Designing an ADT

What do we need to represent?
What operations do we need to support?
What behaviours are we trying to model?
How do we interact with other types?

A Graph ADT

<graph.h> — Create, Destroy

```
typedef struct graph *Graph;

/** A concrete edge type. */
typedef struct edge { vertex v, w; weight n; } edge;

/** Create a new instance of a Graph. */
Graph graph_new (
    size_t max_edges,       /**< maximum value hint */
    size_t max_vertices,    /**< maximum value hint */
    bool directed,          /**< true if a digraph */
    bool weighted           /**< true if edges have weight */
);

/** Deallocate resources used by a Graph. */
void graph_drop (Graph g);
```

A Graph ADT

<graph.h> — Simple Facts

```
/** Get the number of vertices in this Graph. */
size_t graph_num_vertices (Graph g);

/** Get the number of edges in this Graph. */
size_t graph_num_edges (Graph g);

/** Is this graph directed? */
bool graph_directed_p (Graph g);

/** Is this graph weighted? */
bool graph_weighted_p (Graph g);
```

A Graph ADT

<graph.h> — Add/Remove

```
/** Add vertex with index `v' to the Graph.
 * If the vertex already exists, a no-op returning false. */
bool graph_vertex_add (Graph g, vertex v);

/** Add edge `e', from `v' to `w' with weight `n', to the Graph.
 * If the edge already exists, a no-op returning false. */
bool graph_edge_add (Graph g, edge e);

/** Remove edge `e' between `v' and `w' from the Graph. */
void graph_edge_remove (Graph g, edge e);

/** Remove vertex `v' from the Graph. */
void graph_vertex_remove (Graph g, vertex v);
```

A Graph ADT

<graph.h> — Answering Questions

```
/** Does this Graph have this vertex? */
bool graph_has_vertex_p (Graph g, vertex v);

/** What is the degree of this vertex on this Graph? */
size_t graph_vertex_degree (Graph g, vertex v);

/** Does this Graph have this edge? */
bool graph_has_edge_p (Graph g, edge e);
```

Graph Search

Searching on Graphs

We learn properties of a graph by
systematically examining
each of its edges and vertices —

… to compute the degree of all vertices,
we visit each vertex, and count its edges

… for path-related properties
we move from vertex to vertex along edges
choosing edges as we go

we implement general graph-search algorithms
which can solve a wide range of graph problems

Simple Path Search

PROBLEM
does a path exist between vertices v and w?

- examine vertices adjacent to v;
- if any of them is w, we're done!
- otherwise, check from all of the adjacent vertices
  ... rinse and repeat moving away from v

What order do we visit nodes in?
'Breadth-first' (BFS): adjacent nodes first
'Depth-first' (DFS): longest paths first
Dijkstra: lowest-cost paths first
'Greedy Best-First' (GBFS): shortest-heuristic-distance
A*: lowest-cost *and* shortest-heuristic-distance

Simple Path Search

Path searches on graphs tend to follow a simple pattern:

- create a structure that will tell us what next
- add the starting node to that structure
- while that structure isn't empty:
  - get the next vertex from that structure;
  - mark that vertex as visited; and
  - add its neighbours to the structure

What data structure should we use?

BFS: a queue!

DFS: a stack!

Depth-First Search

`count` number of vertices traversed so far

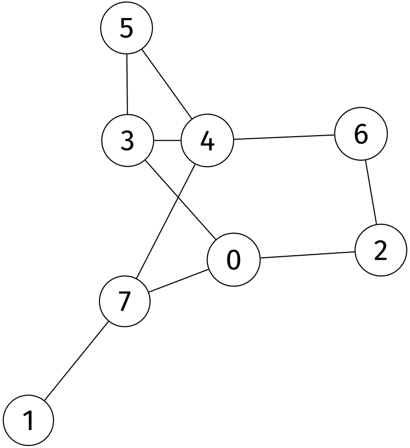
`pre[]` order in which vertices were visited (for 'pre-order')

`st[]` predecessor of each vertex (for 'spanning tree')

the edges traversed in all graph walks form a spanning tree, which has —

- has edges corresponding to call-tree of recursive function
- is the original graph sans cycles/alternate paths
- (in general, a spanning tree has all vertices
  and a minimal set of edges to produce a connected graph;
  no loops, cycles, parallel edges)

Depth-First Search



Depth-First Search

... on an unconnected graph

If a graph is not connected,
DFS will produce
a spanning forest

An edge connecting a vertex with
an ancestor in the DFS tree
that is not its parent is a back edge

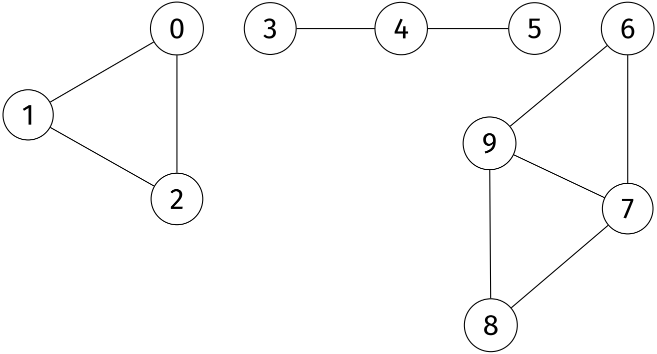
Depth-First Search

... recursively, using the call stack (I)

```
void dfsR (Graph g, edge e) {
    // ... set up `pre' array of `g->nV' items set to -1
    // ... set up `st' array of `g->nV' items set to -1
    // ... set up `count' = 0
    pre[w] = count++;
    st[w] = e.v;
    vertex w = e.w;
    for (vertex i = 0; i < g->nV; i++)
        if (g->edges[w][i] && pre[i] == -1)
            dfsR (g, (edge){.v = w, .w = i});
}
```

Depth-First Search

... recursively, using the call stack (II)



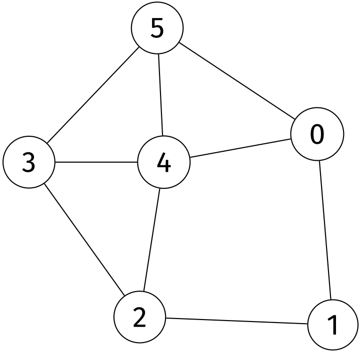
How can we ensure that all vertices are visited?

Depth-First Search

... recursively, using the call stack (III)

```
void dfs (Graph g)
{
    count = 0;
    pre = calloc (g->nV * sizeof (int));
    st = calloc (g->nV * sizeof (int));
    for (vertex v = 0; v < g->nV; v++)
        pre[v] = st[v] = 1;
    for (vertex v = 0; v < g->nV; v++)
        if (pre[v] == -1)
            dfsR (g, (edge){.v = v, .w = v});
}
```

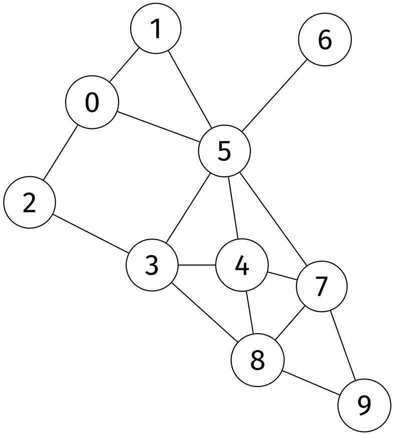
Depth-First Search



Let's do a DFS!

Does a path exist from $0 \cdots 5$?
Yes: $0, 1, 2, 3, 4, 5$.

Depth-First Search



Let's do a DFS!

What do `pre[]` and `st[]` look like?

Depth-First Search

... iteratively

```
void dfs (Graph g, edge e)
{
    // ... set up `pre' array of `g->nV' items set to -1
    // ... set up `st' array of `g->nV' items set to -1
    // ... set up `count' = 0
    Stack s = stack_new ();
    stack_push (s, e);
    while (stack_size (s) > 0) {
        e = stack_pop (s);
        if (pre[e.w] != -1) continue;
        pre[e.w] = count++; st[e.w] = e.v;
        for (int i = 0; i < g->nV; i++)
            if (has_edge (e.w, i) && pre[i] == -1)
                stack_push (s, (edge){.v = e.w, .w = i });
    }
}
```

Breadth-First Search

... iteratively

```
void bfs (Graph g, edge e)
{
    // ... set up `pre' array of `g->nV' items set to -1
    // ... set up `st' array of `g->nV' items set to -1
    // ... set up `count' = 0
    Queue q = queue_new ();
    queue_en (q, e);
    while (queue_size (q) > 0) {
        e = queue_de (q);
        if (pre[e.w] != -1) continue;
        pre[e.w] = count++; st[e.w] = e.v;
        for (int i = 0; i < g->nV; i++)
            if (has_edge (e.w, i) && pre[i] == -1)
                queue_en (q, (edge){.v = e.w, .w = i });
    }
}
```

COMP2521 19T0
Week 5, Tuesday: Graphic Content (IV)!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

weighted graphs
directed graphs

Administrivia

prac exam #1 **10 January**
at 10am, see WebCMS3 for details
(probably) no sample questions released

census date **13 January**
if you hate me and/or the course
prac exam marks back before then

assignment 2 part 1 is out now:
the Fury of Dracula: the View
make sure you have a group on WebCMS 3

Assignment 2, Part 1

the Fury of Dracula: the View

use a version control system
like Fossil, Git, SVN, etc.

use documentation tools like Doxygen

start sooner rather than later;
write some tests before you begin

Directed Graphs

Directed Graphs

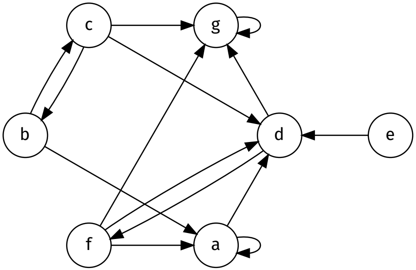
We've mostly considered *undirected* graphs:
an edge relates two vertices equivalently.

Some applications require us to consider
directional edges: $v \to w \neq w \to v$
e.g., 'follow' on Twitter, one-way streets, etc.

In an directed graph or digraph:
edges have direction;
self-loops are allowed;
'parallel' edges are allowed.

Directed Graphs

Example



Where can we get to from g?
Can we get to e from anywhere else?

Directed Graphs

Common Domains

domain	vertex is...	edge is...
WWW	web page	hyperlink
chess	board state	legal move
scheduling	task	precedence
program	function	function call
journals	article	citation

Directed Graphs

Problems and Applications

- Is there a directed path from s to t? (transitive closure)
- What is the shortest path from s to t? (shortest path search)
- Are all vertices mutually reachable? (strong connectivity)

- How can I organise a set of tasks? (topological sort)
- How can I crawl the web? (graph traversal)
- Which web pages are important? (PageRank)

Digraph Terminology (I)

in-degree or $d^{-1}(v)$: the number of directed edges leading into a vertex

out-degree or $d(v)$: the number of directed edges leading out of a vertex

sink a vertex with out-degree 0;

source a vertex with in-degree 0

Digraph Terminology (II)

reachability indicates existence of directed path:
if a directed path $v, \ldots, w$ exists,
w is reachable from v

strongly connected indicates mutual reachability:
if both paths $v, \ldots, w$ and $w, \ldots, v$ exist,
v and w are strongly connected

strong connectivity every vertex reachable from every other vertex;

strongly-connected component maximal strongly-connected subgraph

Digraph Representation

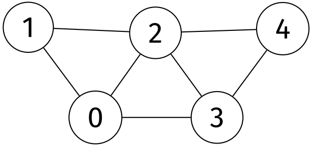
Similar choices as for undirected graphs:

- adjacency matrix … asymmetric, sparse; less space efficient
- adjacency lists … fairly common solution
- edge lists … order of edge components matters
- linked data structures … pointers inherently directional

Can we make our undirected graph implementations directed? Yes!

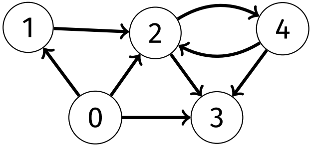
Directed Graphs

Implementation: Adjacency Matrix



$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

unweighted, undirected



$$\begin{bmatrix} - & 1 & 1 & 1 & - \\ - & - & 1 & - & - \\ - & - & - & 1 & 1 \\ - & - & - & - & - \\ - & - & 1 & 1 & - \end{bmatrix}$$

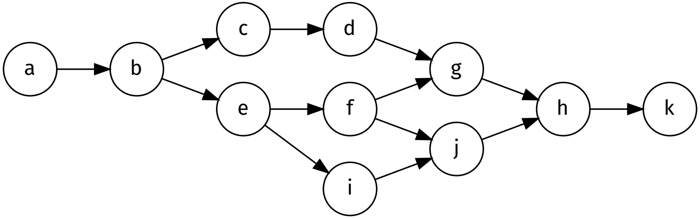
unweighted, directed

Digraph Complexity

	storage	edge add	has edge	outdegree
adj.matrix	$O\left(V + V^2\right)$	$O(1)$	$O(1)$	$O(V)$
adj.list	$O(V + E)$	$O(d(v))$	$O(d(v))$	$O(d(v))$

Overall, adjacency lists tend to be ideal:
real digraphs tend to be sparse
(large V, small average $d(v)$);
algorithms often iterate over v's edges

Directed Acyclic Graphs



Is it a tree? Is it a graph?
No: it's a DAG, a directed acyclic graph.

Tree-like: each vertex has 'children'.
Graph-like: a child vertex may have multiple parents.

Directed Acyclic Graphs

Application: the Topological Sort

NOT EXAMINABLE (and not taught until '4128)

The most common application of a DAG is *topological sorting*:
ordering vertices such that, for any vertices u and v,
if u has a directed edge to v, then v comes after u in the ordering.

Computable with a DFS, tracking *post-order sequence*:
vertices only added after their children have been visited
$\Rightarrow$ a valid topological ordering

dependency problems: *make(1)*, spreadsheets
version-control systems: Git, Fossil, etc.

Digraph Traversal

Mostly the same algorithms as for undirected graphs:
DFS and BFS should all Just Work

e.g., Web crawling: visit every page on the web.
BFS with implicit graph;
on visit, scans page for content, keywords, links
... assumption: WWW is fully connected.

Weighted Graphs

Weighted Graphs

Some applications require us to consider a cost or weight assigned to a relation between two nodes.

Often, we use a geometric interpretation:
low weight $\Rightarrow$ short edge;
high weight $\Rightarrow$ long edge;

Weights aren't always geometric:
some weights are negative.
(We assume we have non-negative weights,
as graphs with negative weights tend to cause problems...)

Weighted Graphs
Implementation

Adjacency matrix:

- store *weight* in each cell, not just true/false.
- need some "no edge exists" value: zero might be a valid weight.

Adjacency list

- add weight to each list node

Edge list:

- add weight to each edge

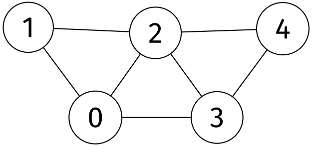
Linked data structure:

- links become link/weight pairs

Works for directed and undirected graphs!

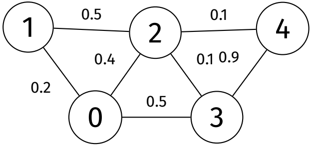
Weighted Graphs

Implementation: Adjacency Matrix



$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

unweighted, undirected



$$\begin{bmatrix} - & 0.2 & 0.4 & 0.5 & - \\ 0.2 & - & 0.5 & - & - \\ 0.4 & 0.5 & - & 0.1 & 0.1 \\ 0.5 & - & 0.1 & - & 0.9 \\ - & - & 0.1 & 0.9 & - \end{bmatrix}$$

weighted, undirected

Weighted Graph Problems

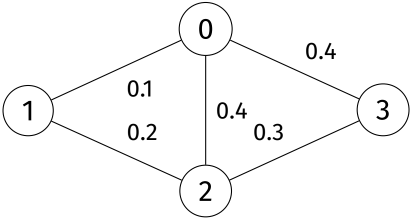
The shortest path problem:

- find the minimum cost path between two vertices
- edges may be directed or undirected
- assuming non-negative weights!

minimum spanning trees (MST):

- find the *weight-minimal* set of edges
  that connect all vertices in a weighted graph
- multiple solutions may exist!
- assuming undirected, non-negatively-weighted graphs

Weighted Graph Problems



What's the shortest path from 0 to 3?
What's the least-hops (shortest unweighted path) from 0 to 2?
What is the minimum spanning tree?

Shortest-Path Search

Shortest-path is useful in navigation and route-finding on physical maps, in computer networks, etc.

Several flavours of shortest-path searches exist:

source-target	the shortest path from v to w;
single-source	the shortest path from v to all other vertices;
all-pairs	the shortest paths for all pairs of v, w

Shortest-Path Search

Formally

On graph G, the weight of p (as $\mathrm{weight}(p)$)
is the sum of weights of p's edges.

The shortest path between v and w
is a simple path $p = [v, \ldots, w]$,
where no other simple path $q = [v, \ldots, w]$, with $q \neq p$,
has a lesser weight (i.e., $\forall q, \mathrm{weight}(p) < \mathrm{weight}(q)$).

Assuming a weighted graph, with no negative weights.
(On an unweighted graph, devolves to least-hops.)

Single-Source Shortest-Path Search

Given a weighted graph G, and a start vertex v,
we want shortest paths from v to all other vertices.

ASIDE how do we represent it?
we get a vertex-indexed array of distances from v,
and a vertex-indexed array of shortest-path predecessors
... it's a spanning tree rooted at v.
(Spanning trees can have weighted and/or directed edges, too!)

Single-Source Shortest-Path Search

A Sketch of the Algorithm

`sssp` (Graph g, vertex v):
  `dists[]` $:= [\infty, \cdots]$
  `dists[v]` $= 0$
  pq $:= \textsc{newPQueue}$
  for each $e := (s, t, \omega)$ **in** $\textsc{adjacent}(v)$,
    $\textsc{enPQueue}(\mathsf{pq}, (s, t), \omega)$

  while $\textsc{length}(pq) > 0$:
    $(\mathtt{s}, \mathtt{t}), \omega := \textsc{dePQueue}(\mathsf{pq})$
    get edges that connect s and t
    relax along edge if new distance is better
    add edges with total path weights

Single-Source Shortest-Path Search

Edge Relaxation

"Edge relaxation" along edge e from s to t:

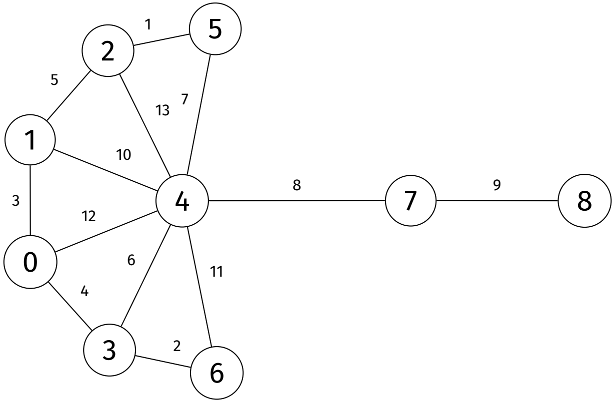
`dist[s]` is length of some path from v to s;
`dist[t]` is length of some path from v to t
if e gives shorter path v to t via s,
update `dist[t]` and `st[t]`.

Relaxation updates data on t, *if* we find a shorter path from v.

```
if (dist[s] + e.weight < dist[t]) {
    dist[t] = dist[s] + e.weight;
    pqueue_en (pq, t, dist[t]);
    st[t] = s;
}
```

Single-Source Shortest-Path Search

Demonstration



	0	1	2	3	4	5	6	7	8
dist									
st									

Single-Source Shortest-Path Search

Results; Complexity

Once this algorithm has run:
shortest path distances are in `dist`;
predecessors in `st` array; trace for a path

COMPLEXITY:
using an adjacency list and a heap: $O(E \log V)$;
using an adjacency matrix: $O(V^2)$.

Just a graph traversal (*a la* BFS, DFS),
but using a PQueue, instead of a Stack/Queue.

This algorithm is usually known as
Dijkstra's algorithm.
Sedgewick calls this a PRIORITY-FIRST SEARCH.

Single-Source Shortest-Path Search

Situation Overview

constraint	algorithm	cost	remark
single-source shortest:			
non-negative weights	Dijkstra	V^2	optimal (dense)
non-negative weights	Dijkstra	$E \log V$	conservatively
acyclic	source-queue	E	optimal
no negative cycles	Bellman-Ford	VE	improvements?
(none)	?	?	NP-hard

All-Pairs Shortest-Path Search

Do Dijkstra's SSSP at every vertex.
(This sucks as much as it sounds like it does.)

Floyd-Warshall.
(Out of scope, see '4121/'4128).

All-Pairs Shortest-Path Search

Situation Overview

constraint	algorithm	cost	remark
all-pairs shortest:			
non-negative weights	Floyd	V^3	same for all
non-negative weights	Dijkstra (PFS)	$VE \log V$	conservatively
acyclic	DFS	VE	same for all
no negative cycles	Floyd	V^3	same for all
no negative cycles	Johnson	$VE \log V$	conservatively
(none)	?	?	NP-hard

Minimum Spanning Trees

History

originally, Otakar Borůvka in 1926:
most economical construction of electric power network
(*O jistém problému minimálním*, 'On a certain minimal problem')

routing and network layout:
electricity, telecommunications, electronic, road, ...
widely applicable $\Rightarrow$ intensely studied problem

Minimum Spanning Trees

The Rules of the Game

A spanning tree ST of a graph $G(V, E)$
is a subgraph $G'(V, E')$, such that $E' \subseteq E$.
ST is connected (spanning) and acyclic (tree)

A minimum spanning tree MST of a graph G
is a spanning tree of G,
where the sum of edge weight
is no larger than any other spanning tree.

So: how do we (efficiently) find a MST for G?

Kruskal's Algorithm

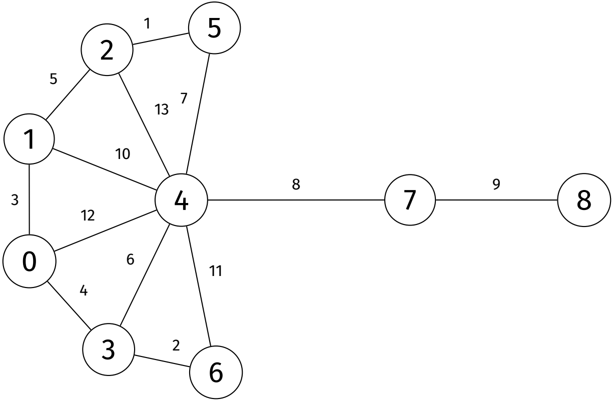
take all edges, sorted according to their weight;
then, for each edge:
add it to the proto-MST;
unless it would introduce a cycle,

Cycle-checking is really expensive
(DFS everything!)
Sedgewick has a 'union-find' that works fine here

Sorting dominates overall:
$O\left(E \log E\right)$.

Kruskal's Algorithm

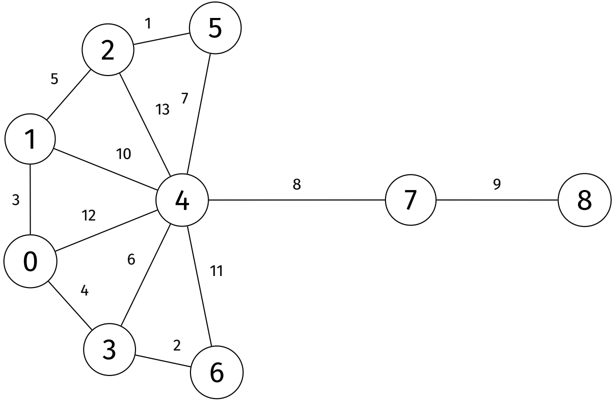
Demonstration (I)



v	0	0	0	1	1	2	2	3	3	4	4	4	7
w	1	3	4	2	4	4	5	4	6	5	6	7	8
ω	3	4	12	5	10	13	1	6	2	7	11	8	9

Kruskal's Algorithm

Demonstration (II)



v	2	3	0	0	1	3	4	4	7	1	4	0	2
w	5	6	1	3	2	4	5	7	8	4	6	4	4
ω	1	2	3	4	5	6	7	8	9	10	11	12	13

Prim-Jarník-Dijkstra Algorithm

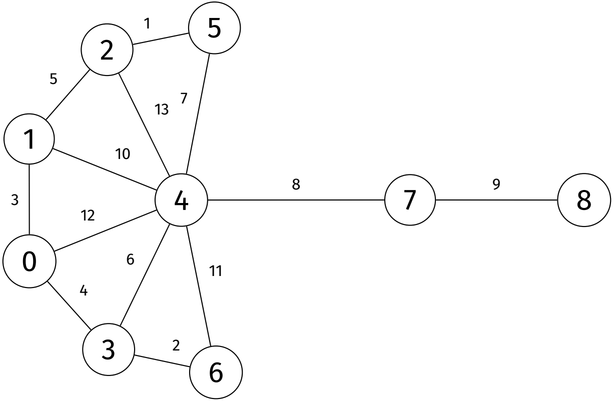
Another approach to computing an MST for graph $G(V, E)$
discovered by Prim (1957), Jarník (1930), Dijkstra (1959)

1. start from any vertex s and with an empty MST
2. choose edge not already in MST to add
   - must not contain a self-loop
   - must connect to a vertex already on MST (on the *fringe*)
   - must have minimal weight of all such edges
3. check to see whether adding the new edge brought any of the non-tree vertices closer to the MST
4. repeat until MST covers all vertices

basically just Dijkstra's SSSP algorithm,
just a graph search but using a PQueue;
$O\left(E \log V\right)$ (adjacency lists, heap)
or $O(V^2)$ (adjacency matrix).

Prim-Jarník-Dijkstra Algorithm

Demonstration



v	0	0	0	1	1	2	2	3	3	4	4	4	7
w	1	3	4	2	4	4	5	4	6	5	6	7	8
ω	3	4	12	5	10	13	1	6	2	7	11	8	9

Other MST Algorithms

- Kruskal: grow many forests
- Prim/Jarník/Dijkstra: maintain connectivity on frontier
- Borůvka/Sollin: component-wise
- Tarjan/Karger/Klein: randomised
- Chazelle: deterministic; best-performing

COMP2521 19T0
Week 4, Thursday: Graphic Content (III)!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

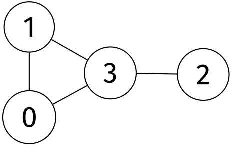
computability
directed graphs

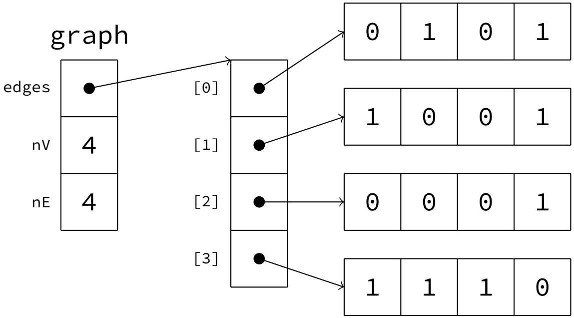
Graph Representation

Recap: Ways of Representing Graphs

Adjacency Matrices

```
struct graph {
    size_t nV, nE;
    bool **matrix;
};
```

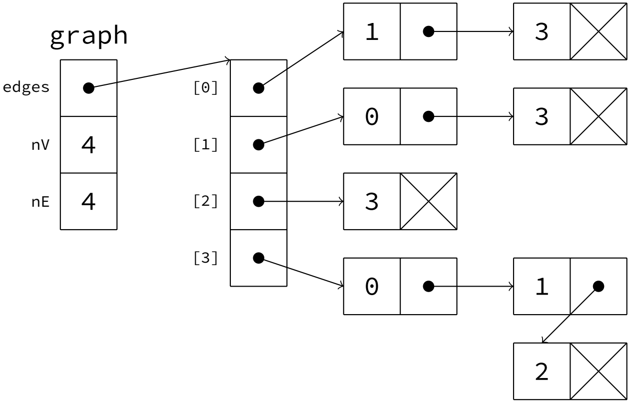




Recap: Ways of Representing Graphs

Adjacency Lists

```
typedef
    struct adjnode
    adjnode;
struct graph {
    size_t nV, nE;
    adjnode **edges;
};
struct adjnode {
    vertex w;
    adjnode *next;
};
```



Recap: Ways of Representing Graphs

Edge Lists

```
struct graph {
    size_t nV, nE;
    edge *edges;
};
```

v	w
0	1
0	3
1	3
2	3

Recap: Ways of Representing Graphs

Linked Data Structure

```
typedef struct vertex {
    Item it;
    size_t degree;
    vertex *neighbours;
} vertex;

struct graph {
    size_t nV, nE;
    vertex *root;
};
```

Recap: Ways of Representing Graphs

Comparison

	matrix	adj.list	edge list	node links
space	V^2	$V + E$	E	$V + E$
initialise	V^2	V	1	V
destroy	V	E	E	$V + E$
insert edge	1	V	1	E
find/remove edge	1	V	E	E
is isolated?	V	1	E	1
degree	V	E	E	E
is adjacent?	1	V	E	E

Hamilton Paths and Tours

Hamilton Path:
a simple path connecting two vertices
that visits each vertex in the graph exactly once

Hamilton Tour:
a cycle
that visits each vertex in the graph exactly once

* * *

Given a list of vertices or edges, easy to check.
Given a graph … how do we know if one exists?
Do we have to find one? If so, how do we find one?

Hamilton Paths and Tours

Brute Force Is Best Force

IDEA brute force!
enumerate every possible path, and check each one.

hack a BFS or DFS to do it:
keep a counter of vertices visited in the current path;
only accept a path only if count is
equal to the number of vertices.

PROBLEM how many paths?
given a simple path:
no path from t to w implies no path from v to w via t...
so there's no point visiting a vertex twice on a simple search
... but that's not true for a Hamilton path!

we must inspect every possible path in the graph.
in a complete graph, we have $V!$ different paths ($\approx (V/e)^V$)

there are well-known, well-defined subsets of this problem
which are easy to solve (Dirac, Ore) ... but in general
this is a non-deterministic polynomial, or NP problem

Euler Paths and Tours

Euler Path:
a simple path connecting two vertices
that visits each edge in the graph exactly once
... exists iff the graph is connected
and has exactly two vertices of odd degree

Euler Tour:
a cycle
that visits each edge in the graph exactly once ... exists iff the graph is connected
and all vertices are of even degree

... these can be found in linear time.

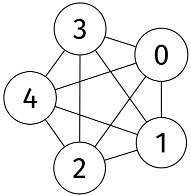
Graph Problems

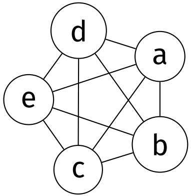
Tractable and Intractable

- tractable: can we find a simple path connecting two vertices in a graph?
  tractable: what's the shortest such path?
  intractable: what's the longest such path?
- tractable: is there a clique in a given graph?
  intractable: what's the largest clique?
- tractable: given two colours, can we colour every vertex in a graph such that no two adjacent vertices are the same colour?
  intractable: what about three colours?

Graph Problems

Bonus Round!





Graph isomorphism:
Can we make two given graphs identical by renaming vertices?

No general solution exists.
We don't know if one can exist.

xkcd 1425 “Tasks” // CC BY-NC 2.5

COMP2521 19T0
Week 6, Tuesday: Order! Order! (I)

Jashank Jeremy

jashank.jeremy@unsw.edu.au

basic sorting algorithms
more sorting algorithms

Administrivia

MYEXPERIENCE now open!
`myexperience.unsw.edu.au`

PRAC EXAM #1 results look pretty good
a majority of people passed the exam!
no problem required >10 LoC;
if you just threw code at the wall,
consider a different strategy next time.

ASSIGNMENT 2 part 1 is underway!
views due **20 Jan 2019**, no extensions.
view dryruns out now — how does your code do?
hunt spec to be released during week07tue lecture

Sorting

Sorting: The Problem

Sorting
... arranging a collection of items in order,
... based on some property of an item (a 'key'),
... using an ordering relation on that property.

Sorting: The Problem

Why? What? Where?

Why are we interested?

- speeds up subsequent searches;
- arranges data in useful ways (human- or otherwise)
  ... *e.g.*, a list of students in a tutorial
- provides useful intermediate for other algorithms
  ... *e.g.*, duplicate detection/removal; DBMS operations

What contexts?

- arrays, linked lists (in-memory, internal)
- files (external, on-disk)
- ... distributed across a network (map/reduce)

We'll focus on sorting arrays (and lists)

Sorting: The Problem

(More) Formally

Pre-conditions:
array `a[N]` of `Items`
`lo`, `hi` are valid indices on `a`
(roughly, $0 \leq \texttt{lo} < \texttt{hi} \leq \texttt{N} - 1$)

Post-conditions:
array `a'[lo..hi]` contains same values
$\texttt{a}'[\texttt{lo}] \leq \texttt{a}'[\texttt{lo} + 1] \leq \texttt{a}'[\texttt{lo} + 2] \leq \cdots \leq \texttt{a}'[\texttt{hi}]$

Sorting: The Problem

Properties: Stability, Adaptive, In-Place

Properties: stable sorts

let $x = \texttt{a[i]}$, $y = \texttt{a[j]}$, where $\text{KEY}(x) \equiv \text{KEY}(y)$
let the 'precedes' relation be that index $i \leq j$.
if x 'precedes' y in a, then x 'precedes' y in a′

Properties: adaptive sorts

where the algorithm's behaviour or performance
is affected by the input data —
that best/average/worst case performance differs
... and can take advantage of *existing order*

Properties: in-place sorts

sort data within original structure,
using only a constant additional amount of space

A Concrete Framework

```c
// we deal with generic `Item's
typedef int Item;

// abstractions to hide details of items
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define eq(A,B) (key(A) == key(B))
#define swap(A,B) { Item t; t = A; A = B; B = t; }
#define cas(A,B) { if (less (A, B)) swap (A, B); }
// cas == Compare And Swap, often hardware assisted

/// Sort a slice of an array of Items.
void sort (Item a[], int lo, int hi);

/// Check for sortedness (to validate functions).
bool sorted_p (Item a[], int lo, int hi);
```

A Concrete Framework

This framework can be adapted by...
defining a different data structure for `Item`;
defining a method for extracting sort keys;
defining a different ordering (`less`);
defining a different `swap` method for different `Item`

```
typedef struct { char *name; char *course; } Item;
#define key(A) (A.name)
#define less(A, B) (strcmp(key(A), key(B)) < 0)
#define swap(A,B) { Item t; t = A; A = B; B = t; }
// ... works because struct assignment works in C
```

Complexity of Sorting Algorithms

In analysing sorting algorithms:

- N: the number of items ($\texttt{hi} - \texttt{lo} + 1$)
- C: the number of comparisons between items
- S: the number of times items are swapped

(We usually aim to minimise C and S.)

Cases to consider for input order:

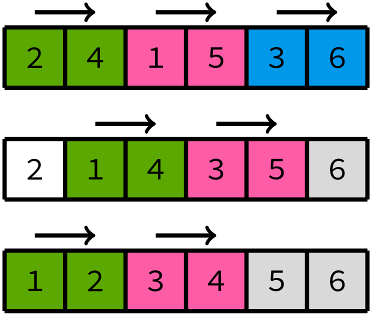
- random order: `Items` in `a[lo..hi]` have no ordering
- sorted-ascending order: $\texttt{a}[\texttt{lo}] \leq \texttt{a}[\texttt{lo} + 1] \leq \cdots \leq \texttt{a}[\texttt{hi}]$
- sorted-descending order: $\texttt{a}[\texttt{lo}] \geq \texttt{a}[\texttt{lo} + 1] \geq \cdots \geq \texttt{a}[\texttt{hi}]$

Three Simple Sorts

- Bubble Sort (oblivious and early-exit)
- Selection Sort
- Insertion Sort

Bubble Sort

Values ‘bubble up’ the array.



Bubble Sort

C Implementation — Oblivious

```c
void sort_bubble (Item items[], size_t lo, size_t hi)
{
    for (size_t i = hi; i > lo; i--)
        for (size_t j = lo + 1; j <= i; j++)
            if (less (items[j], items[j - 1]))
                swap_idx (items, j, j - 1);
}
```

Bubble Sort

C Implementation — Analysis

- Outer loop (C_0) $\Rightarrow N$
  ```
  for (size_t i = n - 1; i > 0; i--)
  ```
- Inner loop (C_1) $\Rightarrow N + (N-1) + (N-2) + \cdots + 2 = (N^2+N)/2 - 1$
  ```
  for (size_t j = 1; j <= i; j++)
  ```
- Comparisons (C_2) $\Rightarrow N + (N-1) + (N-2) + \cdots + 1 + 0 = (N^2-N)/2$
- Swaps (C_3) $\Rightarrow N + (N-1) + (N-2) + \cdots + 1 + 0 = (N^2-N)/2$
  (assuming the worst case: we *always* have to swap)

$$T(n) = NC_0 + \left(\frac{N^2+N}{2} - 1\right) C_1 + \frac{N^2-N}{2} C_2 + \frac{N^2-N}{2} C_3$$

$$\Rightarrow O(N^2)$$

Bubble Sort

Summary

How many steps does it take to sort a collection of N elements?

For the ith iteration, we have $N - i$ comparisons and best 0, worst $N - i$ swaps (depending on sortedness.)

Bubble sort is $O(n^2)$.
Stable, in-place, non-adaptive.

Improving Bubble Sort

‘oblivious’ bubble-sort continues, even if the list is sorted
so what’s a better stopping-case than ‘we ran out of array’?

if we complete a whole pass without swaps, we’re ordered!
this is bubble sort with early exit, or adaptive bubble sort

Adaptive Bubble Sort

C Implementation — Adaptive

```c
void sort_bubble_ee (Item items[], size_t lo, size_t hi)
{
    bool no_swaps = false;
    for (size_t i = hi; i > lo && !no_swaps; i--) {
        no_swaps = true;
        for (size_t j = lo + 1; j <= i; j++)
            if (less (items[j], items[j - 1])) {
                swap_idx (items, j, j - 1);
                no_swaps = false;
            }
    }
}
```

Adaptive Bubble Sort

Analysis; Summary

How many steps does it take
to sort a collection of N elements?

Each traversal does N comparisons.
Best case: exit after one iteration
(if the collection is already sorted.)
Worst case: N traversals still necessary.

$$T_{\text{worst}}\left(N\right) = N - 1 + N - 2 + \cdots + 1 \approx N^2$$
$$T_{\text{best}}\left(N\right) = N$$

Bubble-sort with early exit is *still* $O(N^2)$.
Stable, in-place, adaptive (!).

Selection Sort

Select the smallest element.
Swap it with the first position.

Select the next smallest element.
Swap it with the second position.

… continue until sorted!

Selection Sort

4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	2	7	3	8	6	5	4
1	2	3	7	8	6	5	4
1	2	3	4	8	6	5	7
1	2	3	4	5	6	8	7
1	2	3	4	5	6	8	7
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Selection Sort

C Implementation

```
void sort_selection (Item items[], size_t lo, size_t hi)
{
    for (size_t i = lo; i < hi; i++) {
        size_t low = i;
        for (size_t j = i + 1; j <= hi; j++)
            if (less (items[j], items[low]))
                low = j;
        swap_idx (items, i, low);
    }
}
```

Selection Sort

Analysis; Summary

How many steps does it take
to sort a collection of N elements?

... picking the minimum of a sequence of N elements: N steps.
... inserting at the right place: 1.

$$T\left(N\right) = N + (N-1) + (N-2) + \cdots + 1 = \frac{1}{2}N(N+1)$$

Selection sort is $O\left(N^2\right)$.
Unstable, in-place, oblivious.s

Insertion Sort

Take the first element, insert into the first position.
This starts our 'sorted sublist'.

Take the next element.
Insert it into the sorted sublist
in the right spot!

Repeat until sorted!

Insertion Sort

4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	6	7	8	5	2
1	3	4	5	6	7	8	2
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Insertion Sort

C Implementation

```
void sort_insertion (Item items[], size_t lo, size_t hi)
{
    for (size_t i = lo + 1; i <= hi; i++) {
        Item item = items[i];
        size_t j = i;
        for (/* j */; j > lo; j--) {
            if (! less (item, items[j - 1])) break;
            items[j] = items[j - 1];
        }
        items[j] = item;
    }
}
```

Insertion Sort

Analysis; Summary

How many steps does it take
to sort a collection of N elements?

For every element (of N elements):
1 step to pick an element;
insert into a $N' \leq N$ sequence: up to N steps.

$$T_{\text{worst}}\left(N\right) = 1 + 2 + \cdots + N = \frac{N}{2}(N+1)$$
$$T_{\text{best}}\left(N\right) = 1 + 1 + \cdots + 1 = N$$

Insertion sort is $O\left(N^2\right)$.
Stable, in-place, adaptive.

Shell Sort

One Small Swap for a Sort ...

Bubble- and Insertion-Sort
really only consider *adjacent* elements.

If we make longer-distance exchanges,
can we be more efficient?

What if we consider elements that are some distance apart?
... sort sublists of mod-h indices,
for decreasing h until $h = 1$?

Shell Sort

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
unsorted	4	1	7	3	8	6	5	2
$h = 3$ passes	3			4			5	
		1			2			8
			6			7		
3-sorted	3	1	6	4	2	7	5	8
$h = 2$ passes	2		3		5		6	
		1		4		7		8
2-sorted	2	1	3	4	5	7	6	8
$h = 1$ pass	1	2	3	4	5	6	7	8

Shell Sort

C Implementation

```
void sort_shell (Item items[], size_t lo, size_t hi)
{
    size_t h;
    for (h = 1; h <= (n - 1) / 9; h = (3 * h) + 1);

    for (/* h */; h > 0; h /= 3) {
        // when `h' = 1, this is an insertion sort.
        for (size_t i = h; i < n; i++) {
            Item item = items[i];
            size_t j = i;
            for (/* j */; j >= h && item < items[j - h]; j -= h)
                items[j] = items[j - h];
            items[j] = item;
        }
    }
}
```

Shell Sort

... One Giant Leap for Complexity?

The exact complexity-class depends on the h-sequence.
Probably safe to assume that $O\left(\leq n^2\right)$,
because otherwise what's the point?
Lots of h-value sequences are $O\left(n^{\frac{3}{2}}\right)$.

No 'general' analysis exists.

Shell Sort is $O\left(\leq n^2\right)$.
It is unstable, adaptive, in-place.

Aside: Sorting Linked Lists

Bubble traverse list; if curr > next, swap.

Selection delete selected element, insert at head of sorted list.

Insertion delete first element, do order-preserving insertion.

Shell *(screaming)*

COMP2521 19T0
Week 6, Thursday: Order! Order (II)

Jashank Jeremy

jashank.jeremy@unsw.edu.au

more sorting algorithms
non-comparing sorts

Sorting

Divide-and-Conquer Algorithms

divide-and-conquer algorithms
break up, or shard, the problem
into (easier) computations on smaller pieces,
and combine the results.

(usually) easy to implement recursively!
(usually) easy to implement in parallel!

Divide-And-Conquer Sorting

1. If a collection has less than two elements, it’s sorted.
   Otherwise, split it into 2 parts.
2. Sort both parts separately.
3. Combine the sorted collections to return the final result.

Sorted Merge

Copy elements from the inputs one at a time,
giving preference to the smaller of the two.
When one list is empty,
copy the rest of the elements from the other.

```
/// In-place merge on array `a',
/// of slices `a[lo..mid-1]' and `a[mid..hi]'.
void merge (Item a[], size_t lo, size_t mid, size_t hi);

/// Out-of-place merge, into `dest[0..ndest-1]',
/// from `a[0..nA-1]' and `b[0..nB-1]'.
void merge (Item dest[], size_t ndest,
        Item a[], size_t nA, Item b[], size_t nB);
```

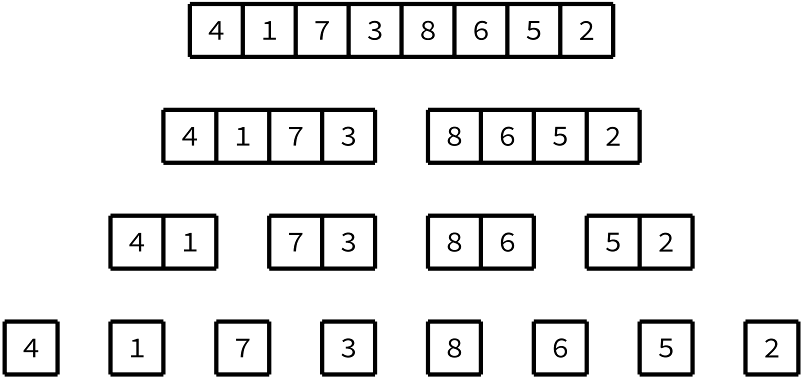
Merge Sort

A divide-and-conquer sort:

partition the input into two equal-sized parts.
recursively sort each of the partitions.
merge the two now-sorted partitions back together.

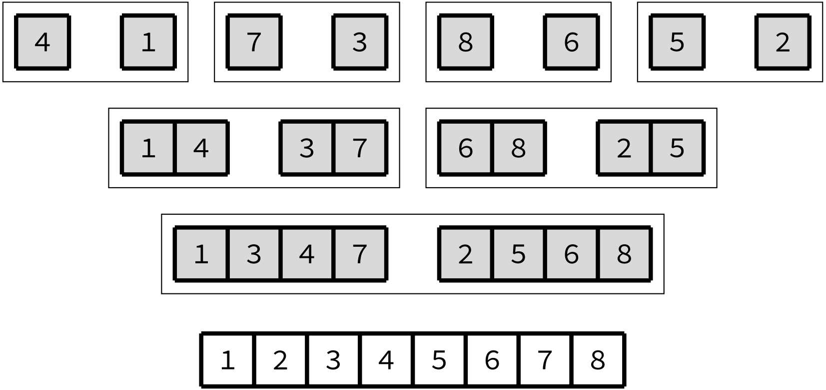
Merge Sort

Demo (I)



Merge Sort

Demo (II)



Merge Sort

C Implementation: Sort

```
void sort_merge (Item a[], size_t lo, size_t hi)
{
    if (hi <= lo) return;
    size_t mid = (lo + hi) / 2;
    sort_merge (a, lo,    mid);
    sort_merge (a, mid+1, hi);
    merge (a, lo, mid, hi);
}
```

Merge Sort

C Implementation: Merge

```
void merge (Item a[], size_t lo, size_t mid, size_t hi)
{
    Item *tmp = calloc (hi - lo + 1, sizeof (Item));
    size_t i = lo, j = mid + 1, k = 0;

    // Scan both segments, copying to `tmp'.
    while (i <= mid && j <= hi)
        tmp[k++] = less (a[i], a[j]) ? a[i++] : a[j++];

    // Copy items from unfinished segment.
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi)  tmp[k++] = a[j++];

    // Copy `tmp' back to main array.
    for (i = lo, k = 0; i <= hi; a[i++] = tmp[k++]);

    free (tmp);
}
```

Merge Sort

Analysis (I)

How many steps does it take
to sort a collection of N elements?

Splitting arrays into two halves: constant time.
To re-combine, N steps.

$$T\left(N\right) = N + 2\,T\left(N/2\right)$$

substitute $N := 2^N$; then:

$$T\left(2^N\right) = 2^N + 2\,T\left(2^N/2\right)$$
$$T\left(2^N\right) = 2^N + 2\,T\left(2^{N-1}\right)$$

Merge Sort
Analysis (II)

divide out 2^N; then:

$$T\left(2^N\right)/2^N = 1 + 2\,T\left(2^{N-1}\right)/\left(2^N\right)$$
$$T\left(2^N\right)/2^N = 1 + T\left(2^{N-1}\right)/\left(2^{N-1}\right)$$

expanding, we get:

$$1 + \left(1 + T\left(2^{N-2}\right)/\left(2^{N-2}\right)\right)$$
$$1 + \left(1 + \left(1 + T\left(2^{N-3}\right)/\left(2^{N-3}\right)\right)\right)$$
$$\ldots = N$$

$$T\left(2^N\right)/2^N = N$$
$$T\left(2^N\right) = 2^N N$$

$$T\left(N\right) = N\log_2 N$$

Merge Sort

Analysis (III)

How many steps does it take
to sort a collection of N elements?

- split array into equal-sized partitions
  halving at each level $\Rightarrow \log_2 N$ levels
- same operations happen at every recursive level
- each 'level' requires $\leq N$ comparisons —
  worst case: two arrays exactly interleaved, N comparisons

Merge Sort

Analysis; Summary

Merge sort is $O(n \log n)$.

Generally, stable...
... as long as the merge is stable.

Not in-place:
$O(n)$ memory for merge; $O(\log n)$ stack space.

Oblivious:
$O(n \log n)$ best case, average case, worst case

Merge Sort on Lists

Straightforward!

- Traverses input in sequential order.
- Don't need extra space for merging list.
- Works top-down and … bottom-up?

Bottom-Up Merge Sort

An approach that works non-recursively!

- on each pass, our array contains sorted *runs* of length m.
- initially, N sorted runs of length 1.
- The first pass merges adjacent elements into runs of length 2.
- The second pass merges adjacent elements into runs of length 4.
- ... continue until we have a single sorted run of length N.

Can be used for *external* sorting;
e.g., sorting disk-file contents

Bottom-Up Merge Sort

C Implementation

```
#define MIN(a,b) ((a) < (b) ? (a) : (b))

void sort_merge_bu (Item a[], size_t lo, size_t hi)
{
    for (size_t m = 1; m <= lo - hi; m *= 2)
        for (size_t i = lo; i <= hi - m; i += 2 * m) {
            size_t end = MIN (i + 2*m - 1, hi);
            merge (a, i, i + m - 1, end);
        }
}
```

Merge Sort is Slow

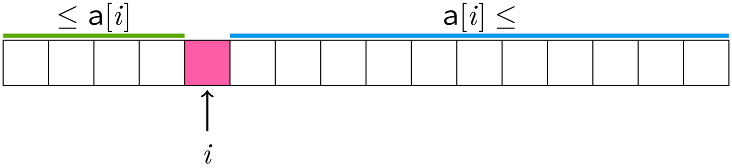
Merge sort uses a trivial split operation;
all the heavy lifting is in the *merge* operation.

Can we split the collection in a more intelligent way,
so combining the results is easier?

...*e.g.*, making sure all elements in one part
are less than elements in the second part?

Better Partitioning

to partition array a
at some index i (the 'pivot'),
we need to swap elements such that,
for other indices j and k,
$j < i$ implies $a[j] \leq a[i]$
$k > i$ implies $a[i] \leq a[k]$



Quick Sort

C Implementation: Sort

Assuming we have a partition function,
this looks very similar to merge sort.

```
void sort_quick_naive (Item a[], size_t lo, size_t hi)
{
    if (hi <= lo) return;
    size_t part = partition (a, lo, hi);
    sort_quick_naive (a, lo, part ? (part - 1) : 0);
    sort_quick_naive (a, part + 1, hi);
    // look, ma! no merge!
}
```

Quick Sort

C Implementation: Naïve Partition

```
size_t partition (Item a[], size_t lo, size_t hi)
{
    Item v = a[lo]; // our `pivot' value.
    size_t i = lo + 1, j = hi;
    for (;;) {
        while (less (a[i], v) && i < j) i++;
        while (less (v, a[j]) && i < j) j--;
        if (i == j) break;
        swap_idx (a, i, j);
    }
    j = less (a[i], v) ? i : i - 1;
    swap_idx (a, lo, j);
    return j;
}
```

Quick Sort

Analysis

How many steps does it take to sort a collection of N elements?

N steps to partition an array...
constant-time combination of sub-results.

best-case (equal sized partitions): $O(N \log N)$
worst-case (one part contains all elements):
$T(N) = N + T(N-1) = N + (N-1) + T(N-2)$
... $= N(N+1)/2$, which is $O\left(N^2\right)$

Quick Sort

Summary

Quick sort with naïve partition is...

Unstable (in this implementation)...
... but can be made stable.

In-place: partitioning is done in-place;
stack depth is $O(N)$ worst-case, $O(\log N)$ average

Oblivious.

Problems with Quick Sort

Picking the first or last element as pivot
is an absolutely terrible life choice.

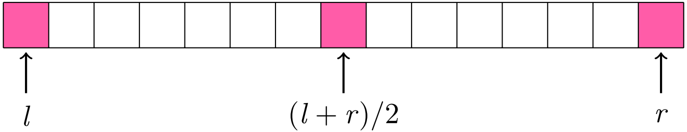
... existing order is a worst case.
... existing *reverse* order is a worst case.
`partition` always gives us parts of size $N - 1$ and 0.

Our ideal pivot is the *median* value.
Our worst pivot is the largest/smallest value.
We can reduce the probability of picking a bad pivot...

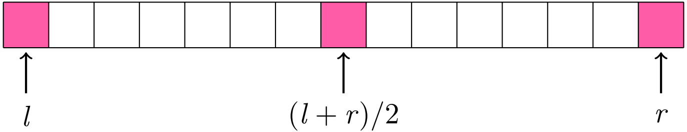
Quick Sort with Median-of-Three Partition

Pick three values: left-most, middle, right-most.
Pick the median of these three values as our pivot.

Ordered data is no longer a worst-case scenario.
In general, doesn't eliminate the worst-case ...
... but makes it much less likely.



Quick Sort with Median-of-Three Partitioning



1. Pick a$[l]$, a$[r]$, a$[(l+r)/2]$
2. Swap a$[r-1]$ and a$[(l+r)/2]$
3. Sort a$[l]$, a$[r-1]$, a$[r]$, such that a$[l] \leq$ a$[r-1] \leq$ a$[r]$
4. Partition on a$[l+1]$ to a$[r-1]$.

Quick Sort with Median-of-Three Partitioning

C Implementation

```c
void qs_median3 (Item a[], size_t lo, size_t hi)
{
    size_t mid = (lo + hi) / 2;
    if (less (a[mid], a[lo])) swap_idx (a, lo, mid);
    if (less (a[hi], a[mid])) swap_idx (a, mid, hi);
    if (less (a[mid], a[lo])) swap_idx (a, lo, mid);
    // now, we have a[lo] <= a[mid] <= a[hi]
    // swap a[mid] to a[lo+1] to use as pivot
    swap_idx (a, lo+1, mid);
}

void sort_quick_m3 (Item a[], size_t lo, size_t hi)
{
    if (hi <= lo) return;
    qs_median3 (a, lo, hi);
    size_t part = partition (a, lo + 1, hi - 1);
    sort_quick_m3 (a, lo, part ? (part - 1) : 0);
    sort_quick_m3 (a, part + 1, hi);
}
```

Quick Sort Optimisations

Sub-file Cutoff

For small sequences (when $n < 5$, say),
quick sort is expensive
because of the recursion overhead.

With a sub-file cutoff, we have two choices:
use a different algorithm on small partitions; or
do a second sort after the quicksort finishes.
(Insertion sort is a good choice: lots of almost-sorted data!)

Quick Sort Optimisations

Bentley-McIlroy's Three-Way Partition

For sequences with many duplicate keys,
partitioning can screw up badly.

instead, do a three-way partition:
keys $< a[i]$, $= a[i]$, $> a[i]$.

Quick Sort on Lists

Straightforward to do...
if we just use the first or last element as pivot
(which means we're vulnerable to ordered data again)

using a random or median-of-three pivot
is now $O(n)$ not $O(1)$

Quick Sort vs Merge Sort

Design of modern CPUs mean,
for sorting arrays in RAM
quicksort *generally* outperforms mergesort.

quicksort is more ‘cache friendly’:
good locality of access on arrays

on the other hand, mergesort is
readily stable, readily parallel,
more efficient with slower data;
a good choice for sorting linked lists

The $n \log n$ Lower Bound

How Low Can We Go? (I)

If we have 3 items,
then $3! = 6$ possible permutations as input.
(n items implies $n!$ possible permutations.)

If we do 1 comparison,
we can form two categories (true, false).
(k comparisons implies 2^k categories.)

The $n \log n$ Lower Bound

How Low Can We Go? (I)

n items implies $n!$ possible permutations.
k comparisons implies 2^k categories.

We need to do enough comparisons so
$n! \leq 2^k$.
$$log_2 n! \leq log_2 2^k$$
$$log_2 n! \leq k$$
... applying Stirling's approximation, and waving our hands:
$n \log n < k$.

the theoretical lower bound on
worst-case execution time
for comparison-based sorts is $O(n \log n)$.
(Quicksort, mergesort are pretty much
as good as it gets, for unknown data.)

Non-Comparison-Based Sorting

All the sorts so far have been
comparison-based sorts.
(They compare things, using some ordering relation $\leq$.)
Works on *any* data, so long as we have $\leq$.

What if we know more about the keys?
... could we get down to $O(n)$ time?

Key-Indexed Counting Sort

count up the number of times each key appears;
this indexes where each item belongs in the sorted array

FOR EXAMPLE:
assuming my key domain is numbers $[0 \ldots 10]$,
if we have three ‘0’s, and two ‘1’s,
‘2’s must go at index 5 and onwards

look, ma! no comparisons!
look, ma! **an $O(n)$ sort!**
terms and conditions apply, see in store for details

Key-Indexed Counting Sort

we must know our sequence is of size N,
and the domain of keys in that sequence.

pumped-up KICS pretty efficient
... if M is small compared to N.
actually, $O(n + M)$ ... so if we have $1, 2, 999999$ ...

Not in-place — uses a temporary array.
Can be stable! Not really adaptive.

Heap Sort

We already have a data structure
which has element ordering as an invariant:
the heap or priority queue.

We *could* just dump all n elements
into a priority queue, and dequeue them —
n operations of $O(\log n)$ complexity.
no gain.

What if we used the heap-fix-down mechanism
on the whole array, popping off the maximum item,
and shrinking the heap each time? That's $O(n)$!
The catch: the inner loop is expensive.

Heap Sort

C Implementation

```
void sort_heap (Item a[], size_t lo, size_t hi)
{
    size_t N = hi - lo + 1;
    Item *pq = &a[lo - 1];
    for (size_t k = N/2; k >= 1; k--)
        heap_fixdown (pq, k, N);
    while (N > 1) {
        swap_idx (pq, 1, N);
        heap_fixdown (pq, 1, --N);
    }
}
```

COMP2521 19T0
Week 7, Tuesday: A Question of Balance

Jashank Jeremy

jashank.jeremy@unsw.edu.au

radix sort
balanced trees

Sorting

Radix Sorting

Can we decompose our keys?
Radix sorts let us deal with this case.

Keys are values in some base-R number system.
e.g., binary, $R = 2$; decimal, $R = 10$;
ASCII, $R = 128$ or $R = 256$; Unicode, $R = 2^{16}$

Sorting individually on each part of the key at a time:
digit-by-digit, character-by-character, rune-by-rune, *etc.*

Radix Sorting, Most-Significant-Digit First

Consider characters, digits, bits, runes, *etc.*,
from left to right;
partitioning input into R pieces according to `key.0`;
recurse into each piece, using succesive keys —
`key.1`, `key.2`, ..., `key.`w

1019	2301	3129	2122
1019	**2**301	**3**129	**2**122
1019	**23**01	**21**22	**31**29
1019	2122	2301	3129

with $R = 2$, roughly a quicksort.

Radix Sorting, Least-Significant-Digit First

Consider characters, digits, bits, runes, *etc.*,
from right to left;
use a **stable** sort using the dth digit as key,
using (*e.g.*,) key-indexed counting sort.

1019	2301	3129	2122
101**9**	230**1**	312**9**	212**2**
23**0**1	21**2**2	10**1**9	31**2**9
2**3**01	1**0**19	2**1**22	3**1**29
1019	**2**122	**3**129	**2**301
1019	2122	2301	3129

this *will not work* if the sort is not stable!

Radix Sort

Analysis; Summary

Complexity: $O\left(w\left(n+R\right)\right) \approx O(n)$,
where w is the 'width' of data;
the algorithm makes w passes over n keys

LSD

Not in-place: $O(n+R)$ extra space required.
May be stable! Usable on variable length data.

MSD

Not in-place: $O(n+DR)$ extra space required.
(D is the recursion depth.)
May be stable! Usable on variable length data.
Can complete *before* examining all of all keys.

Balanced Trees

Recap: The Search Problem

input a key value
output item(s) containing that key

Common variations:

- keys are unique; key value matches 0 or 1 items
- multiple keys in search, items containing any key
- multiple keys in search/item, items containing all keys

We assume: keys are unique, each item has one key.

Recap: Trees; Binary Trees

Trees are branched data structures,
consisting of nodes and edges, with no cycles.

Each node contains a value.
Each node has edges to $\leq k$ other nodes.
For now, $k = 2$ — binary trees

Trees can be viewed as a set of nested structures:
each node has k (possibly empty) subtrees.

Recap: Binary Search Trees

For all nodes in the tree:

the values in the left subtree
are less than the node value the

values in the right subtree
are greater than the node value

A binary tree of n nodes
is degenerate if its height is
at most $n - 1$.

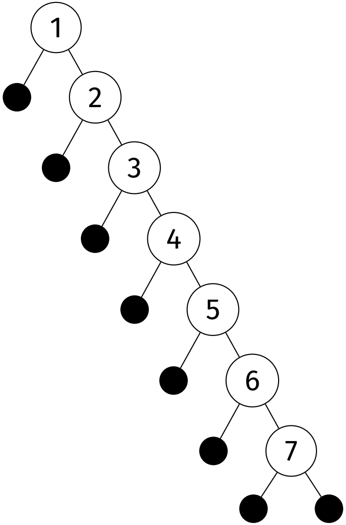
A binary tree of n nodes
is balanced if its height is
at least $\lfloor \log_2 n \rfloor$.

Structure tends to be determined
by order of insertion:
$[4, 2, 1, 3, 6, 5, 7]$ vs $[6, 5, 2, 1, 3, 4, 7]$

Binary Search Trees

The Worst Case

Ascending-ordered or
descending-ordered data
is a pathological case:
we always right- or left-insert
along the spine of the tree.



Binary Search Trees

Performance

Cost for insertion:
balanced $O(\log_2 n)$, degenerate $O(n)$
(we always traverse the height of the tree)

Cost for search/deletion:
balanced $O(\log_2 n)$, degenerate $O(n)$
(worst case, key $\notin \tau$; traverse the height)

We want to build balanced trees.

What is ‘Balance’, then?

PERFECTLY BALANCED

a *weight-balanced* or *size-balanced* tree has, for every node,

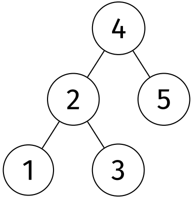
$$|\textsc{size}\,(l) - \textsc{size}\,(r)| < 2$$

LESS STRINGENTLY

a *height-balanced* tree has, for every node,

$$|\textsc{height}\,(l) - \textsc{height}\,(r)| < 2$$

Balanced or Not? (I)



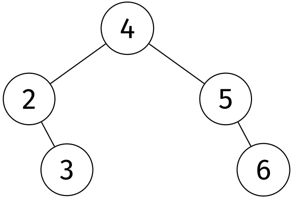
$$\text{SIZE}\left(\tau_4\right) = 5$$
$$\text{SIZE}\left(\tau_2\right) = 3$$
$$\text{SIZE}\left(\tau_5\right) = 1$$
$$\text{SIZE}\left(\tau_1\right) = 1$$
$$\text{SIZE}\left(\tau_3\right) = 1$$
$$\text{SIZE}\left(\tau_2\right) - \text{SIZE}\left(\tau_5\right) = 2$$

NOT SIZE BALANCED

$$\text{HEIGHT}\left(\tau_4\right) = 2$$
$$\text{HEIGHT}\left(\tau_2\right) = 1$$
$$\text{HEIGHT}\left(\tau_5\right) = 0$$
$$\text{HEIGHT}\left(\tau_1\right) = 0$$
$$\text{HEIGHT}\left(\tau_3\right) = 0$$
$$\text{HEIGHT}\left(\tau_2\right) - \text{HEIGHT}\left(\tau_5\right) = 1$$

HEIGHT BALANCED

Balanced or Not? (II)



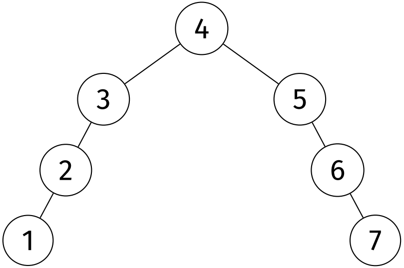
$$\text{SIZE}\left(\mathcal{T}_4\right) = 5$$
$$\text{SIZE}\left(\mathcal{T}_2\right) = 2$$
$$\text{SIZE}\left(\mathcal{T}_5\right) = 2$$
$$\text{SIZE}\left(\mathcal{T}_3\right) = 1$$
$$\text{SIZE}\left(\mathcal{T}_6\right) = 1$$

SIZE BALANCED

$$\text{HEIGHT}\left(\mathcal{T}_4\right) = 2$$
$$\text{HEIGHT}\left(\mathcal{T}_2\right) = 1$$
$$\text{HEIGHT}\left(\mathcal{T}_5\right) = 1$$
$$\text{HEIGHT}\left(\mathcal{T}_3\right) = 0$$
$$\text{HEIGHT}\left(\mathcal{T}_6\right) = 0$$

HEIGHT BALANCED

Balanced or Not? (III)



Let's look at τ_3.

$$\text{SIZE}\left(\tau_2\right) = 2$$
$$\text{SIZE}\left(\tau_\varnothing\right) = 0$$
$$2 - 0 = 2 \not< 2$$

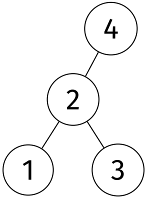
NOT SIZE BALANCED

Let's look at τ_5.

$$\text{HEIGHT}\left(\tau_\varnothing\right) = 0$$
$$\text{HEIGHT}\left(\tau_6\right) = 1$$
$$|0 - 1| = 1 < 2$$

HEIGHT BALANCED

Balanced or Not?
(IV)



Let's look at τ_4.

$$\text{SIZE}\,(\tau_2) = 3$$
$$\text{SIZE}\,(\tau_\varnothing) = 0$$
$$3 - 0 = 3 \not< 2$$

NOT SIZE BALANCED

Let's look at τ_4.

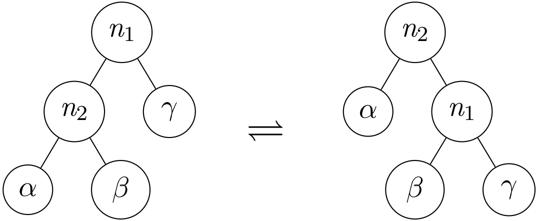
$$\text{HEIGHT}\,(\tau_2) = 1 \;\; \text{HEIGHT}\,(\tau_\varnothing) = 0$$
$$1 - 0 = 1 < 2$$

HEIGHT BALANCED

Rotations

Rebalancing Primitives

LEFT ROTATION and RIGHT ROTATION:
a pair of 'primitive' operations
that change the balance of a tree
whilst maintaining a search tree.



$$(n_1, (n_2, \alpha, \beta), \gamma) \rightleftharpoons (n_2, \alpha, (n_1, \beta, \gamma))$$

Rotations

Rotating Right

```
btree_node *btree_rotate_right (btree_node *n1)
{
    if (n1 == NULL) return NULL;
    btree_node *n2 = n1->left;
    if (n2 == NULL) return n1;
    n1->left = n2->right;
    n2->right = n1;
    return n2;
}
```

n_1 starts as the root of this subtree and is demoted;
n_2 starts as the left subtree of this tree, and is promoted.

Rotations

Rotating Left

```
btree_node *btree_rotate_left (btree_node *n2)
{
    if (n2 == NULL) return NULL;
    btree_node *n1 = n2->right;
    if (n1 == NULL) return n2;
    n2->right = n1->left;
    n1->left = n2;
    return n1;
}
```

n_2 starts as the root of this subtree and is demoted;
n_1 starts as the right subtree of this tree, and is promoted.

Partition

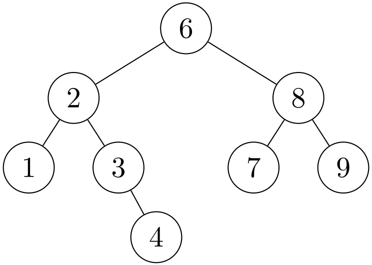
Rotation in Context

A way to brute-force some balance into a tree:
lifting some kth index to the root.

$$
\begin{aligned}
&\textsc{partition} :: \mathrm{BTree} \to \mathrm{Word} \to \mathrm{BTree}\\
&\textsc{partition}\ \mathrm{Empty}\ k = \mathrm{Empty}\\
&\textsc{partition}\ \ (\mathrm{Node}\ n\ l\ r)\ \ k\\
&\quad \mid k < \textsc{size}\, l = \textsc{rotate}\mathrm{R}\ \ (\mathrm{Node}\ n\ \ (\textsc{partition}\, l\, k)\ \ r)\\
&\quad \mid \textsc{size}\, l < k = \textsc{rotate}\mathrm{L}\ \ (\mathrm{Node}\ n\ l\ \ (\textsc{partition}\, r\ \ (k - 1 - \textsc{size}\ l)))\\
&\quad \mid \textsf{otherwise} = \mathrm{Node}\ n\ l\ r
\end{aligned}
$$

Partition

Partition in Context



What happens if we partition at index 3 (node 4)?

Partition

C Implementation

```
btree_node *btree_partition (btree_node *tree, size_t k)
{
    if (tree == NULL) return NULL;
    size_t lsize = size (tree->left);
    if (lsize > k) {
        tree->left = btree_partition (tree->left, k);
        tree = btree_rotate_right (tree);
    }
    if (lsize < k) {
        tree->right = btree_partition (tree->right, k - 1 - lsize);
        tree = btree_rotate_left (tree);
    }
    return tree;
}
```

Primitives

With our primitive operations in hand —

$$\textsc{rotateL} :: \mathrm{BTree} \to \mathrm{BTree}$$
$$\textsc{rotateR} :: \mathrm{BTree} \to \mathrm{BTree}$$
$$\textsc{partition} :: \mathrm{BTree} \to \mathrm{Word} \to \mathrm{BTree}$$

— let's go balance some trees!

Approach #1: Global Rebalancing

Move the median node to the root,
by partitioning on $\textsc{size}\,\tau/2$;
then, balance the left subtree,
and balance the right subtree.

```
btree_node *btree_balance_global (btree_node *tree)
{
    if (tree == NULL) return NULL;
    if (size (tree) < 2) return tree;
    tree = partition (tree, size (tree) / 2);
    tree->left = btree_balance_global (tree->left);
    tree->right = btree_balance_global (tree->right);
    return tree;
}
```

Approach #1: Global Rebalancing

Problems

- cost of rebalancing:
  for many trees, $O(n)$; for degenerate trees, $O(n \log n)$
- what if we insert more keys?
  - rebalance on every insertion
  - rebalance every k insertions; what k is good?
  - rebalance when imbalance exceeds threshold.

  we either have more costly instertions
  or degraded performance for (possibly unbounded) periods.
  … given a sufficiently dynamic tree, sadness.

Global vs Local Rebalancing

GLOBAL REBALANCING

walks every node, balances its subtree;
$\Rightarrow$ perfectly balanced tree — at cost.

LOCAL REBALANCING

do small, incremental operations
to improve the overall balance of the tree
... at the cost of imperfect balance

Local Rebalancing Approaches

amortisation: do (a small amount) more work now to avoid more work later

randomisation: use randomness to reduce impact of BST worst cases

optimisation: maintain structural information for performance

Root Insertion

How do we insert a node at the root of a tree?
(Without having to rearrange all the nodes?)

We do a leaf insertion ...
... and rotate the new node up the tree.

More work? **No!**
Same complexity as leaf insertion,
but more actual work is done: amortisation.

(Side-effect: recently-inserted items are close to the root.
Depending on what you're doing, this might be very useful!)

Root Insertion

C Implementation

```
btree_node *btree_insert_root (btree_node *tree, Item it)
{
    if (tree == NULL)
        return btree_node_new (it, NULL, NULL);
    if (less (it, tree->value)) {
        tree->left = btree_insert_root (tree->left, it);
        tree = btree_rotate_right (tree);
    } else {
        tree->right = btree_insert_root (tree->right, it);
        tree = btree_rotate_left (tree);
    }
    return tree;
}
```

Randomised Insertion

BSTs don't have control over insertion order.
worst cases — (partially) ordered data — are common.

to minimise the likelihood of a degenerate tree,
we randomly choose which level to insert a node;
at each level, probability depends on remaining tree size.

do a 'normal' leaf insertion, most of the time.
randomly (with a certain probability),
do a root insertion of a value.

Randomised Insertion

C Implementation

```
btree_node *btree_insert_rand (btree_node *tree, Item it)
{
    if (tree == NULL)
        return btree_node_new (it, NULL, NULL);
    if (rand () < (RAND_MAX / size (tree)))
        return btree_insert_root (tree, it);
    else if (less (it, tree->value))
        tree->left = btree_insert_rand (tree->left, it);
    else
        tree->right = btree_insert_rand (tree->right, it);
    return tree;
}
```

Randomised Insertion

Properties

building a randomised BST is equivalent to
building a standard BST with
a random initial permutation of keys.

worst-case, best-case, average-case performance:
same as a standard BST —
but with no penalty for ordering!

Randomised Deletion

We could do something similar for deletion:
when choosing a node to promote,
choose randomly from the
in-order predecessor or successor

Splay Trees

Root insertion can still leave us
with a degenerate tree.

Splay trees vary root-insertion,
by considering *three* levels of the tree
— parent, child, grandchild —
and performing double-rotations based on p-c-g orientation;
the idea: double-rotations improve balance.

No guarantees, but *improved* performance.

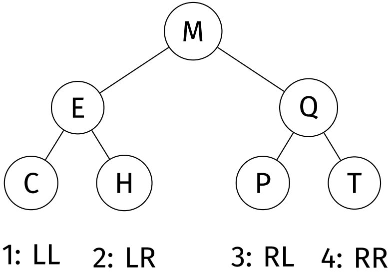
"... their performance is amortised by
the amount of effort required to understand them."

— me, 2016

Splay Trees

Double-Rotation Cases

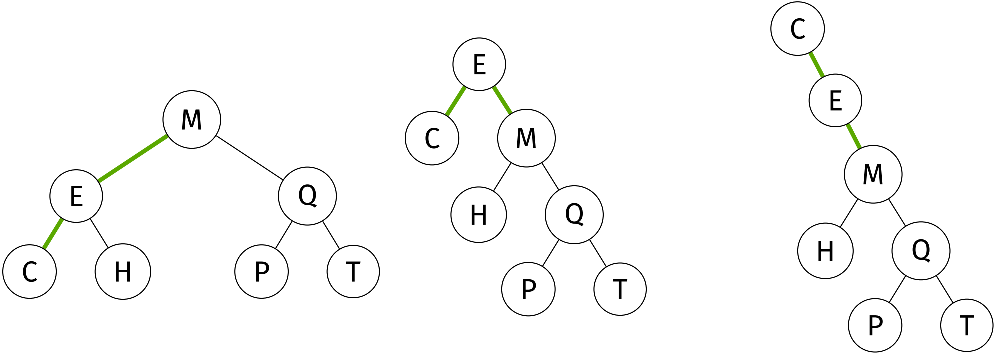
Four choices to consider for a double-rotation:



Splay Rotations

Double-Rotation: Left, Left

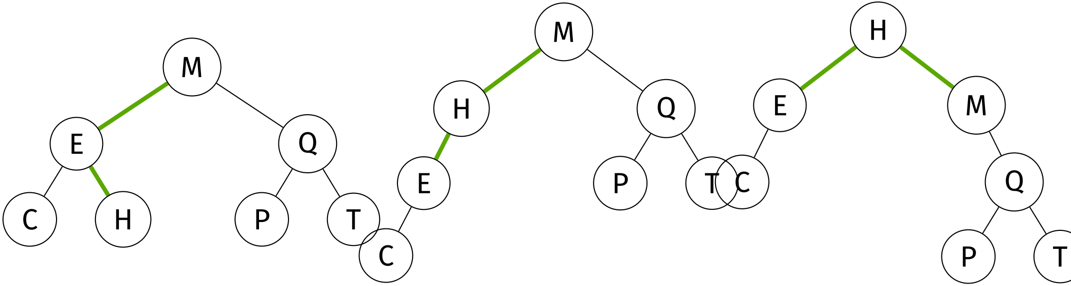
$\textsc{rotateR}\ \tau_M \qquad \textsc{rotateR}\ \tau_E$



Splay Rotations

Double-Rotation: Left, Right

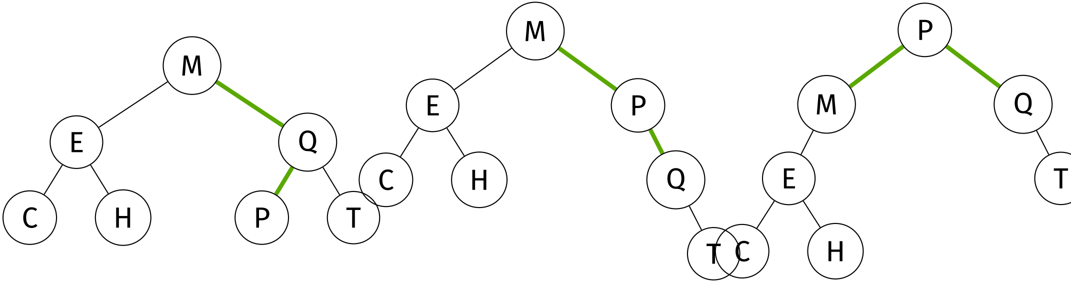
$\textsc{rotateL}\ \tau_{\mathsf{E}}$ $\textsc{rotateR}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Right, Left

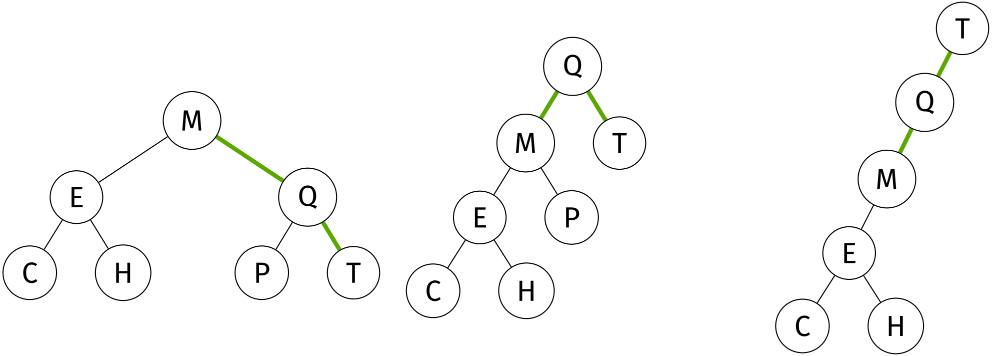
$\textsc{rotateR}\ \tau_{\mathsf{Q}} \qquad \textsc{rotateL}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Right, Right

$\textsc{rotateL}\ \tau_M \qquad \textsc{rotateL}\ \tau_Q$



COMP2521 19T0
Week 7, Thursday: Tropical Paradise

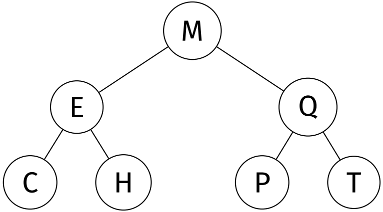
Jashank Jeremy

jashank.jeremy@unsw.edu.au

exotic trees

Balanced Trees

Four choices to consider for a double-rotation:

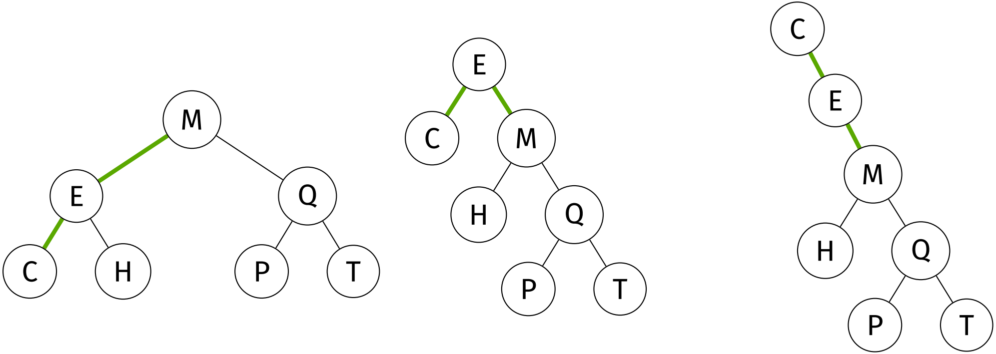


1: LL 2: LR 3: RL 4: RR

Splay Rotations

Double-Rotation: Left, Left

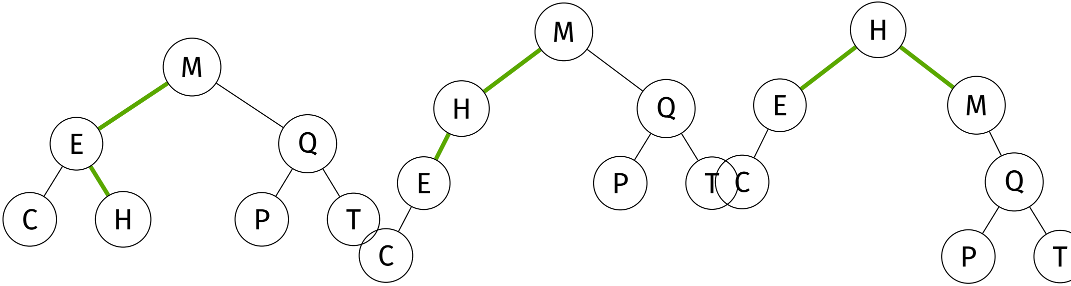
ROTATER τ_{M} ROTATER τ_{E}



Splay Rotations

Double-Rotation: Left, Right

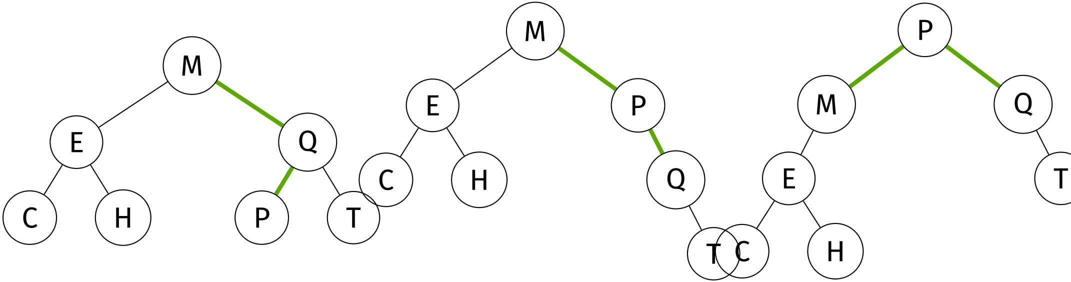
$\textsc{rotateL}\ \tau_{\mathsf{E}} \qquad \textsc{rotateR}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Right, Left

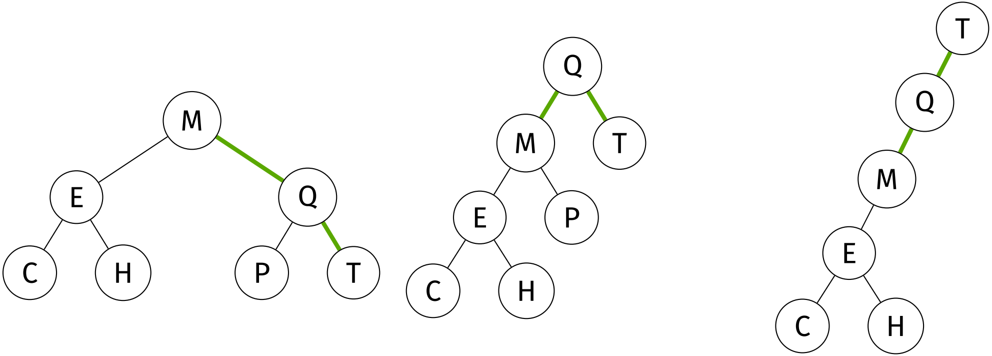
$\textsc{rotateR}\ \tau_{\mathsf{Q}} \qquad \textsc{rotateL}\ \tau_{\mathsf{M}}$



Splay Rotations

Double-Rotation: Right, Right

$\text{ROTATEL } \tau_{M}$ $\text{ROTATEL } \tau_{Q}$



Splay Search

Some implementations do rotation-on-search,
which has a similar effect to periodic rotations;
increases search cost by doing more work,
decreases search cost by moving likely keys closer to root.

Even on a degenerate tree,
splay search massively improves the balance of the tree.

Splay Trees

C Implementation (I)

```
btree_node *btree_insert_splay (btree_node *tree, Item it)
{
    if (tree == NULL) return btree_node_new (it, NULL, NULL);
    int diff = item_cmp (it, tree->value);
    if (diff <  0) {
        if (tree->left == NULL) {
            tree->left = btree_node_new (it, NULL, NULL);
            return tree;
        }
        int ldiff = item_cmp (it, tree->left->value);
        if (ldiff < 0) {
            // Case 1: left-left
            tree->left->left = btree_insert_splay (tree->left->left, it);
            tree = btree_rotate_right (tree);
        } else {
            // Case 2: left-right
            tree->left->right = btree_insert_splay (tree->left->right, it);
            tree->left = btree_rotate_left (tree->left);
        }
        return btree_rotate_right (tree);
```

Splay Trees

C Implementation (II)

```
// ... btree_insert_splay continues ...
    } else if (diff  > 0) {
        int rdiff = item_cmp (it, tree->right->value);
        if (rdiff < 0) {
            // Case 3: right-left
            tree->right->left = btree_insert_splay (tree->right->left, it);
            tree->right = btree_rotate_right (tree->right);
        } else {
            // Case 4: right-right
            tree->right->right = btree_insert_splay (tree->right->right, it);
            tree = btree_rotate_left (tree);
        }
        return btree_rotate_left (tree);
    } else
        tree->value = it;
    return tree;

}
```

without insertion-specific code, we might call this `btree_splay`

Splay Search

C Implementation

```
btree_node *btree_search_splay (btree_node **root, Item it)
{
    assert (root != NULL);
    if (*root == NULL) return NULL;
    *root = btree_splay (*root, it);
    if (item_cmp ((*root)->value, it) == 0)
        return *root;
    else
        return NULL;
}
```

Splay Trees: Why Bother?

Insertion Time Complexity

worst case (for work):
item inserted at the end of a degenerate tree.
$O(n)$ steps necessary here...
but overall tree height now halved

worst case (from resulting tree):
item inserted at root of degenerate tree.
$O(1)$ steps necessary. surprise!

even in the worst case,
not possible to *repeatedly* have
$O(n)$ steps to insert

Splay Trees: Why Bother?

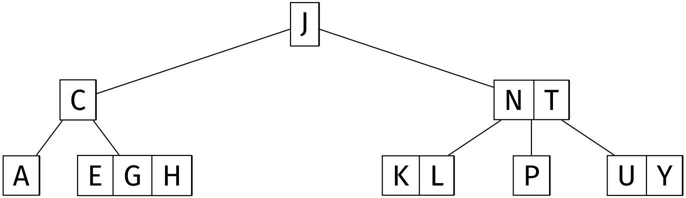
Overall Time Complexity

Assuming we do splay operations on insert and search,
assuming we have N nodes and M inserts/searches:
average $O\left(\left(N+M\right)\log_2\left(N+M\right)\right)$

A good (amortised) cost overall …
but no guarantees of improved individual operations:
some may still be $O\left(N\right)$.

2–3–4 Trees

2–3–4 trees have three types of nodes:
2–nodes have one value and two children;
3–nodes have two values and three children;
4–nodes have three values and four children;

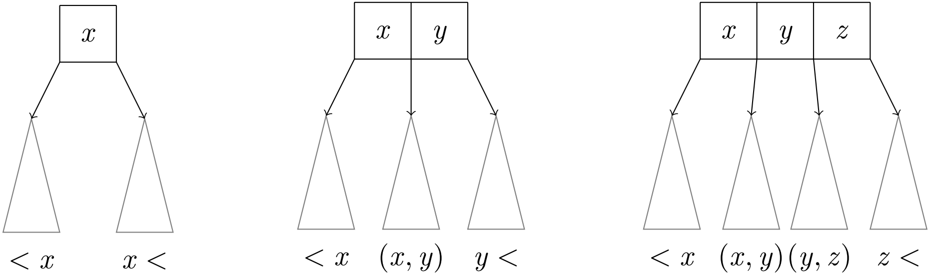


2–3–4 trees grow ‘upwards’ from the leaves,
all of which are equidistant to the root.

2–3–4 Trees

Node Value Ordering

A similar ordering to a conventional BST:



2–3–4 Tree Search

2–3–4 trees are always balanced; depth is $O(\log n)$

worst case for depth: all nodes are 2-nodes
same case as for balanced BSTs, i.e. $d \simeq \log_2 n$

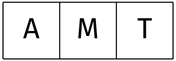
best case for depth: all nodes are 4-nodes
balanced tree with branching factor 4, i.e. $d \simeq \log_4 n$

2–3–4 Tree Insertion

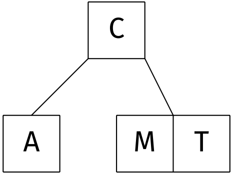
The Algorithm

1. find leaf node where item belongs (via `search`)
2. if node is not full (i.e., order < 4),
   insert item in this node, order++.
3. if node is full (i.e., contains 3 Items):
   1. split into two 2-nodes as leaves
   2. promote middle element to parent
   3. insert item into appropriate leaf 2-node
   4. if parent is a 4-node,
      continue split/promote upwards
   5. if promote to root, and root is a 4-node,
      split root node and add new root

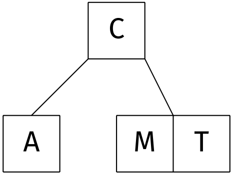
2–3–4 Tree Insertion (I)



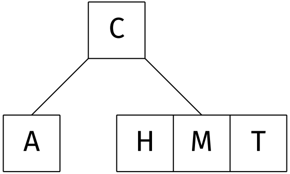
INSERT C



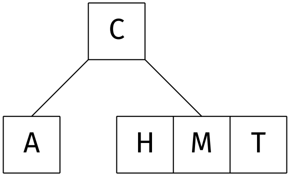
2–3–4 Tree Insertion (II)



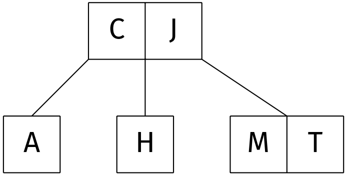
INSERT H



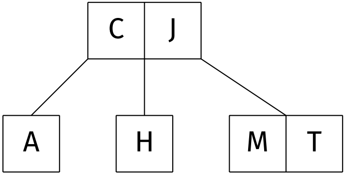
2–3–4 Tree Insertion
(III)



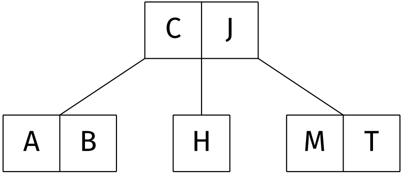
INSERT J



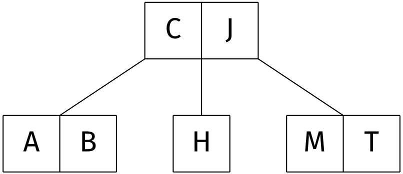
2–3–4 Tree Insertion (IV)



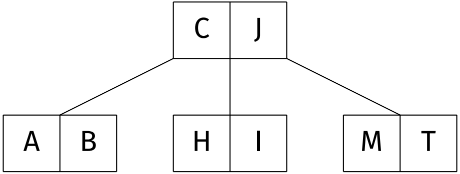
INSERT B



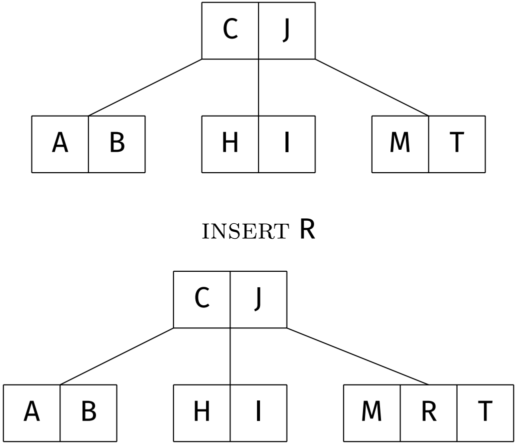
2–3–4 Tree Insertion (V)



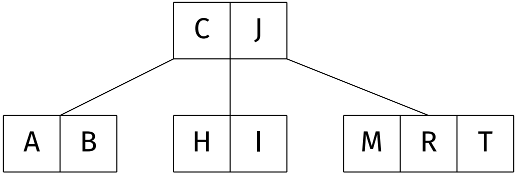
INSERT I



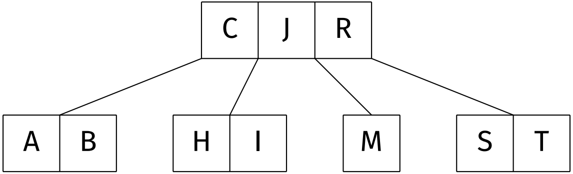
2–3–4 Tree Insertion (VI)



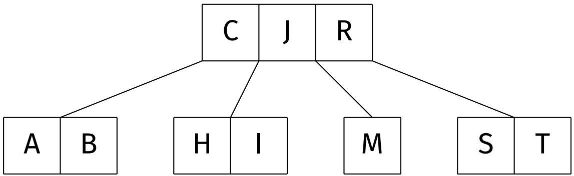
2–3–4 Tree Insertion (VII)



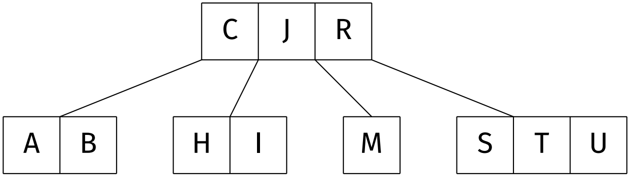
INSERT S



2–3–4 Tree Insertion (VIII)

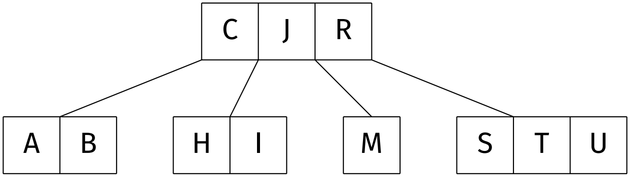


INSERT U

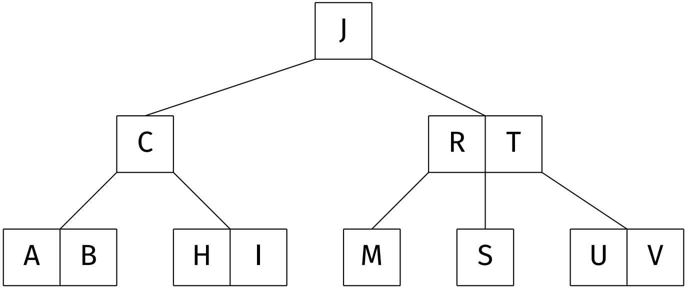


2–3–4 Tree Insertion

(IX)



INSERT V



2–3–4 Trees

C Implementation: Node Structure

```
typedef struct t234_node t234_node;
struct t234_node {
    int order;       // 2, 3, 4
    Item data[3];    // items in node
    t234_node *child[4]; // links to subtrees
};
```

2–3–4 Tree Search

C Implementation

```
Item *t234_search (t234_node *tree, Item it)
{
    if (tree == NULL) return NULL;
    int i, diff = 0;
    for (i = 0; i < tree->order - 1; i++) {
        diff = item_cmp (it, tree->data[i]);
        if (diff <= 0) break;
    }
    if (diff == 0) return &(t->data[i]);
    else return t234_search (t->child[i], k);
}
```

2–3–4–M

Why stop with just 2-, 3-, and 4-nodes?
If we allow nodes to hold $M/2$ to M items,
we have a B-tree.

commonly used in DBMS, FS, ...
where a node represents a disk page.

Red-Black Trees

2–3–4, unplugged (I)

red-black trees are a representation of 2–3–4 trees
using only plain old BST nodes;
each node needs one extra value to encode link type,
but we no longer have to deal with different kinds of nodes.

plain old binary search tree search works, unmodified
get benefits of 2–3–4 tree self-balancing on insert, delete
… with great complexity in insertion/deletion.

Red-Black Trees

2–3–4, unplugged (II)

red links combine nodes to represent 3- and 4-nodes;
effectively, child along red link is a 2–3–4 neighbour.
black links are analogous to ‘ordinary’ child links.
some texts call these ‘red nodes’ and ‘black nodes’

THE RULES:
each link is either red or black
no two red links appear consecutively on any path
all paths from root to leaf have same number of black links

COMP2521 19T0

Week 8, Tuesday: Hash Tables

Jashank Jeremy

jashank.jeremy@unsw.edu.au

hashing
performance

Hash Tables

Searching

The State Of Play

So far we've seen...
linked list: insert $O(1)$, search $O(n)$
ordered linked list: insert $O(n)$, search $O(n)$
array: insert $O(1)$, search $O(n)$
ordered array: insert $O(n)$, search $O(\log n)$
search tree: insert $O(\log n)$, search $O(\log n)$

... but these are still all pretty slow,
and perform less-than-ideally on modern architectures
(due to cache locality effects)

Searching

Even Faster?

In an ideal world, we can index on arbitrary keys,
and get constant-time $O(1)$ access.

Key-indexed arrays get some of the way there,
but have downsides:
... requires dense range of index values
... uses fixed-size array; sizing it is hard
... can't use arbitrary keys!

Hashing

Hashing lets us approximate this:
arbitrary keys! (so long as we can hash them)
map keys into a compact range of index values!
store items in array, accessed by index value!
$O(1)$!

We need three things:
an array of Items, of size N
a hash function,
$\textsc{hash} :: \texttt{Key} \to \texttt{size} \to [0 \cdots N)$,
a collision resolution method,
for when $k_1 \neq k_2 \land \textsc{hash}\,(k_1, N) = h\,(k_2, N)$;
collisions are inevitable when $\textsc{dom}\,(k) \gg N$

Hash Functions

Properties we want h to have:

- for a table of size N, output range is 0 to $N-1$;
- pure, deterministic: $h(k, N)$ gives the same result;
- spreads key values uniformly over index range (assuming keys are unformly distributed)
- cheap (enough) to compute … otherwise, what's the point?

Ideally all of the above, *and*

pre-image resistant:
for $h = \textsc{Hash}(m)$, given h, hard to pick m;

second pre-image resistant:
for $\textsc{Hash}(m_1) = \textsc{Hash}(m_2)$,
given m_1, hard to find $m_2 \neq m_1$;

collision resistant:
for $\textsc{Hash}(m_1) = \textsc{Hash}(m_2)$,
hard to find m_1 and m_2.

For our purposes, we don't need cryptographic hash functions.
(COMP6[48]41, MATH3411 go into detail.)

Hash Functions

Example (I)

A simple hash function for single characters, if $N = 128$:

```
size_t hash (char key, size_t N)
{
    return key; // N redundant
}
```

Not really useful:
key range is usually much larger than N.

Hash Functions

Example (II)

Another simple hash function, for integers:

```
size_t hash (int key, size_t N)
{
    return key % N;
}
```

How big is N?
small $N \Rightarrow$ too many collisions!

Hash Functions

Example (III)

A simple hash function, for strings:

```
size_t hash (char *key, size_t N)
{
    return strlen (key) % N;
}
```

(You should never *actually* do this.)

Hash Functions

Example (IV)

A better string hash function:

```
size_t hash (char *key, size_t N)
{
    size_t h = 0;
    for (size_t i = 0; key[i] != '\0'; i++)
        h += key[i];
    return h % N;
}
```

Hash Functions

Example (V)

A more sophisticated hash function:

```
size_t hash (char *key, size_t N)
{
    size_t h = 0;
    unsigned a = 127; // prime
    for (size_t i = 0; key[i] != '\0'; i++)
        h = ((a * h) + key[i]) % N;
    return h;
}
```

Hash Functions

Example (VI)

Using *universal hashing*,
which introduces randomness while using the entire key:

```
size_t hash (char *key, size_t N)
{
    size_t h = 0;
    unsigned a = 31415, b = 21783;
    for (size_t i = 0; key[i] != '\0'; i++) {
        a = (a * b) % (N - 1);
        h = ((a * h) + key[i]) % N;
    }
    return h;
}
```

Collisions

What happens if two keys hash the same?
We go to the same array index … then what?

… allow multiple Items in a single location,
via *e.g.*, array of item arrays
array of linked lists

… systematically compute new indices
by various *probing* strategies

… resize the array
by adjusting the hash function,
and moving everything (!)

Collisions

Separate Chaining

Given N slots and M items:
best case, all lists have length M/N
worst case, one list with length M, all others 0

with a good hash and $M \leq N$, cost $O(1)$;
with a good hash and $M > N$, cost $O(M/N)$

(The M/N ratio is called *load*.)

Collisions

Linear Probing

If the table is not close to being full,
there are still many empty slots;
we could just use the next available slot along;
open-address hashing.

to reach the first item is $O(1)$;
search for subsequent items depends on load;
successful search cost: $\frac{1}{2}\left(1 + 1/\left(1-\alpha\right)\right)$
unsuccessful search cost: $\frac{1}{2}\left(1 + 1/\left(1-\alpha\right)^2\right)$
(assuming reasonably uniform data, good hash function)

… but tends towards $O(N)$ when α is high.

Collisions

Double-Hash Probing

We switch from $\textsc{hash}$ to $\textsc{hash2}$
(which should not return 0!),
and use it as the step to the 'next' item.
$\textsc{hash}$ and $\textsc{hash2}$ should be
relatively prime to each other, and to N.
(Easy, if we pick a prime N.)

Significantly faster than linear probing for high α

Hash Table Performance

Choosing a good HASH is critical.
Choosing a good N for M is critical.
Choosing a good resolution approach is critical.

linear probing: fastest, given big N!
double hashing: fastest for higher α, more efficient
chaining: possible for $\alpha \geq 1$, but degenerates

Performance

Why do we care, anyway?

good performance $\Rightarrow$ less hardware, happy users.
bad performance $\Rightarrow$ more hardware, unhappy users.

generally, performance is proportional to execution time;
we may be interested in other things (memory, i/o, ...)

Premature optimisation

is the root

of all evil.

Developing Efficient Programs

1. Design the program well[1]
2. Implement the program well[2]
3. Test the program well
4. Only after you're sure it's working, measure performance
5. If (and only if) performance is inadequate, find the 'hot spots'
6. Tune the code to fix these
7. Repeat measure-analyse-tune cycle until performance ok

---

[1] See, e.g., *Algorithms* by Sedgewick, *Algorithms* by Cormen/Leierson/Rivest/Stein.
[2] See, e.g., *Programming Pearls*, *the Practice of Programming*.

Performance Estimates

Complexity analysis give info on most appropriate algorithm.
We can also consider an experimental approach to performance:

- determine the *critical operations* in the program
- determine *classes of input* data and *likelihood* of each
- *estimate* the cost (#crit.ops) for each class of data
- produce a weighted sum estimate for *overall cost*

Often, however...

- assumptions made in estimating performance are invalid
- we overlook some frequent and/or expensive operation

Performance Measurements

Basis of performance evaluation:
measure program execution.

empirical study suggests the '80/20' rule:
most programs spend
most of their execution time
in a small part of their code.

most code has little impact on overall performance
small parts account for most execution time

To improve performance: focus on *bottlenecks* first.

Performance Measurement

Profiling Execution

We need a way to measure how much
each block of code costs:
a *profiler*.

gprof(1) displays execution profiles
for programs compiled with `-pg`;
profiling info is left behind in `gmon.out`,
which *gprof(1)* can read.

gprof(1) gives a table (a *flat profile*) containing:
number of times each function was called,
% of total execution time spent in the function,
average execution time per call to that function,
execution time for this function and its children

Performance Improvement

Once you have a profile, you can identify hot points.
To improve the performance:

- change the algorithm and/or data-structures
  - may give orders-of-magnitude better performance
  - but it is extremely costly to rebuild the system
- use simple efficiency tricks to reduce costs
  - may improve performance by one order-of-magnitude
- use the compiler's optimization switches (e.g., -O, -O2, -O3)
  - may improve performance by one order-of-magnitude

Conclusions

Time and profile your code
only when you are done.

Don’t optimise code unless you have to.
(You almost never will.)

Fixing your algorithm is
almost always the solution

Using compiler optimisations
is usually good enough.

COMP2521 19T0

Week 8, Thursday: The Course in Review

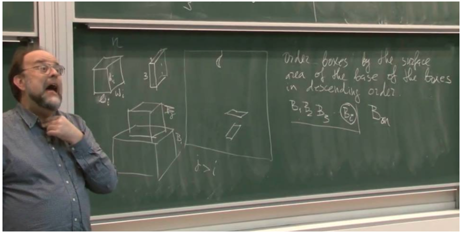
Jashank Jeremy

jashank.jeremy@unsw.edu.au

course review
exam information

Where Next?

The Algorithms Stream



COMP3121/3821 **Algorithms and Programming Techniques** (T1/T2)
dynamic/linear/greedy programming, flow networks, strings, ...

COMP4121 **Advanced and Parallel Algorithms** (T3)
pure theory: PageRank, Markov models, error-correction, ...

COMP4128 **Programming Challenges** (T3)
pure practice: puzzles, challenges, contests; applications!

Where Next?

The Databases Stream

COMP3311 **Database Systems**
COMP9315 **Database Systems Implementation**
COMP9313 **Big Data Management**
COMP4317 **XML and Databases**
COMP9318 **Data Warehousing and Data Mining**
COMP9319 **Web Data Compression and Search**
COMP6714 **Information Retrieval and Web Search**

Where Next?

The Formal Methods Stream

COMP2111 **System Modelling and Design**
COMP3141 **Software System Design and Implementation**
COMP3151 **Foundations of Concurrency**
COMP3153 **Algorithmic Verification**
COMP3161 **Concepts of Programming Languages**
COMP4141 **Theory of Computation**
COMP4161 **Advanced Software Verification**
COMP6721 **(In-)Formal Methods: The Lost Art**
COMP6752 **Parameterised and Exact Computation**

Where Next?

Other Streams

The Systems Stream

COMP3231/3891 **Operating Systems**
COMP9242 **Advanced Operating Systems**
COMP9243 **Distributed Systems**

The Networks Stream

COMP3331 **Computer Networks**
COMP9332 **Network Routing and Switching**
COMP9334 **Capacity Planning**
COMP9336 **Mobile Networks**
COMP4337 **Securing Wireless Networks**