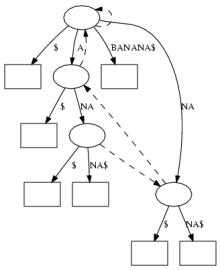
Further Mathematics and Algorithms

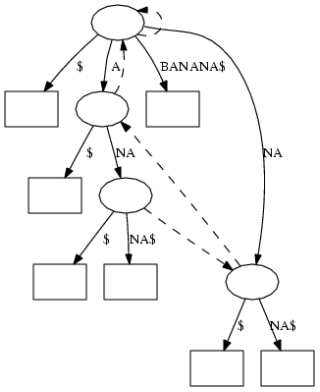
Lesson 1: *Use Data Structures and Algorithms!*



Course structure, examples of data structures and algorithms

Outline

1. Course structure
2. Example of Using DSA
3. Sophisticated Program
4. State-of-the-Art



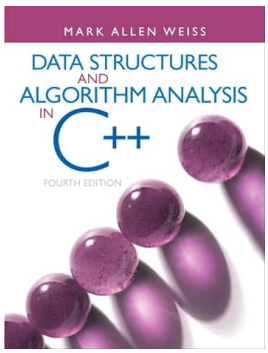
Welcome to Further Mathematics and Algorithms

- First 7 weeks Daniela and I will be teaching you about algorithms
- The last 4 weeks you will learn some further maths
- I'm teaching you algorithms (and data structures) in C++
- My ambition is not only to teach you data structures and algorithms academically, but also to get to a new level of coding in C++

Quick Survey on C++

- Who considers themselves a competent coder in C++?
- Who knows what a class constructor is?
- Who knows what a default constructor is?
- Who is happy using templates?
- Who understands pointers?
- Who knows what the key word `explicit` means?
- Who has heard of *resource acquisition is initialisation (RAII)*?

Recommended Course Text



- *Data Structures and Algorithm Analysis in C++* by M. A. Weiss
  - ⋆ Best introduction to Data Structures and Algorithms
  - ⋆ Not huge, but covers all the basics
- Available in the library

What is a Data Structure?

any of various methods of organising data items (as records) in a computer

- Container for data
- E.g. sets, stacks, lists, trees, graphs
- Clean interface, e.g. `push`, `pop`, `delete`
- Usually designed for fast or convenient access

What is an Algorithm?

a sequence of unambiguous instructions for solving a problem, i.e. for obtaining a required output for a legitimate input in a finite amount of time

- E.g. `sort`, `search`, `match`
- Well defined and generic
- Guarantees on performance

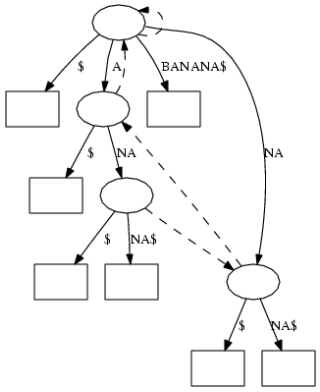
Exemplary OO-Software

- Abstraction from details of problem
- Declaration of intention
- Clean interfaces
- Hidden implementations
- Makes programs readable and maintainable
- Reuse code—don't even have to write it yourself

  Thou shall not re-implement common data structures

Outline

1. Course structure
2. **Example of Using DSA**
3. Sophisticated Program
4. State-of-the-Art



Example: Sort program

- Suppose we want to write a program to
  - ⋆ read an input file of integers
  - ⋆ sort the integers
  - ⋆ write a list of integers to standard out
- In Unix there is a command called `sort` which does just this
- Note that you don't know the number of inputs

Code for sort

```
#include <iostream>
#include <fstream>

int main(int argc, char** argv) {
  std::ifstream myfile(argv[1]);

  int array_size = 10;
  int* array = new int[array_size];
  int cnt = 0;
  while(myfile.good()) {
    if (cnt==array_size) {
      int* new_array = new int[2*array_size];
      for(int i=0; i<array_size; ++i)
        new_array[i] = array[i];
      delete[] array;
      array = new_array;
      array_size *= 2;
    }
    myfile >> array[cnt++];
  }
```



```
  for(int i=0; i<cnt; ++i) {
    int index = 0;
    for(int j=1; j<cnt-i; ++j) {
      if (array[j]<array[index])
        index = j;
    }
    std::cout << array[index] << std::endl;
    array[index] = array[cnt-i-1];
  }
}
```


Notes on Code

- Details of code don't matter
- Simple program ($\sim$ 20 lines of code)
- Uses a simple array
- Difficult to see what is going on
- On 100 000 inputs it takes 10 seconds to run

Using Data Structures and algorithms

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;

int main(int argc, char *argv[])
{
  ifstream in(argv[1]);
  vector<int> data;
  copy(istream_iterator<int>(in), istream_iterator<int>(),
       back_inserter(data));
  sort(data.begin(), data.end());
  copy(data.begin(), data.end(), ostream_iterator<int>(cout,"\n"));
}
```


Sorting Doubles

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;

int main(int argc, char *argv[])
{
  ifstream in(argv[1]);
  vector<double> data;
  copy(istream_iterator<double>(in), istream_iterator<double>(),
       back_inserter(data));
  sort(data.begin(), data.end());
  copy(data.begin(), data.end(), ostream_iterator<double>(cout,"\n"));
}
```


Notes on C++

- `vector<int>` is the C++ standard resizable array
- input/output is treated as a copy
- Code is easy to read
  - ⋆ Declare `vector<int>` or `vector<double>`
  - ⋆ copy input file into vector
  - ⋆ sort vector
  - ⋆ copy sorted vector to standard output stream
- On 100 000 inputs takes 10ms to run

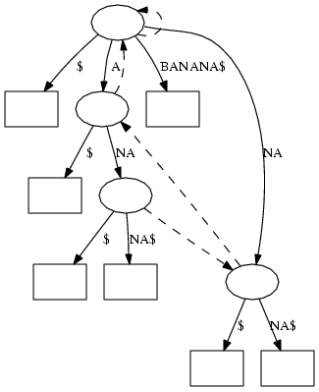
Summary: Why use Data Structures?

Data structure version is

- Easier/quicker to code
- More readable (less bugs)
- Easier to modify and change
- Easier to port to another language
- Better (in this case faster)

Outline

1. Course structure
2. Example of Using DSA
3. **Sophisticated Program**
4. State-of-the-Art



Sophisticated Programs

- Data structures and algorithms allow moderately competent programmers to write some very impressive programs
- E.g. consider a program to count all occurrences of words in a document
- We want to output the words in sorted order

countWords

```
#include <stuff>

int main(int argc, char** argv) {
  ifstream in(argv[1]);
  map<string, int> words;

  string s;
  while(in >> s) {
    ++words[s];
  }

  vector<pair<string,int> > pairs;
  copy(words.begin(), words.end(), back_inserter(pairs));
  sort(pairs.begin(), pairs.end(),
    [](auto& a, auto&b){return a.second>b.second;});

  for(auto w=pairs.begin(); w!=pairs.end(); ++w) {
    cout << w->first << "␣occurs␣" << w->second << "␣times\n";
  }
}
```


Using countWords

```
> countWords text.dat | more
the occurs 97 times
of occurs 96 times
to occurs 57 times
and occurs 42 times
a occurs 36 times
be occurs 31 times
will occurs 26 times
we occurs 23 times
that occurs 23 times
is occurs 21 times
have occurs 19 times
freedom occurs 18 times
```


Programming Challenge

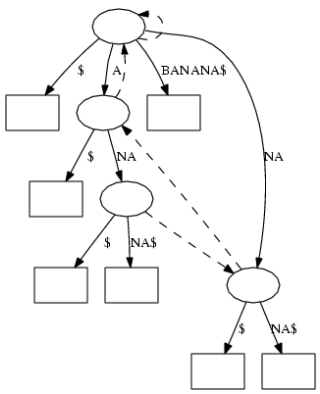
- Run on "I have a dream" speech with 1550 words in 0.02 seconds
- Challenge for good programmers

  Write a program without use data structures in less that 10 times as much code that runs in less than 10 times as long

- Probably possible, but certainly not easy—almost certainly take you 10 times longer to code

Outline

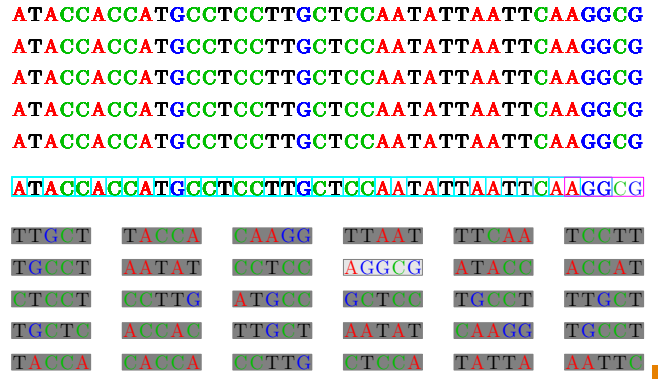
1. Course structure
2. Example of Using DSA
3. Sophisticated Program
4. **State-of-the-Art**



DNA Sequencing

- In modern whole shotgun genome sequencing the full genome is broken into small pieces
- The pieces are then read by a sequencing machine
- This reads short sections (around 1000) bases
- The reads are then assembled to construct the full genome

Sequencing and Assembly



New Generation Sequencers

- The estimated cost of sequencing the human genome in 2005 was $\$10\,000\,000$
- To reduce the cost there was and is a drive to produce new sequencing machines
- These tend to read much shorter sections of DNA (e.g. 20-100nt)
- Can these be assembled?

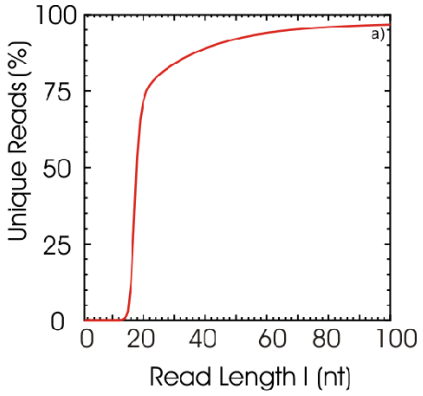
Repeats

- The difficulty of assembly is caused by repeats

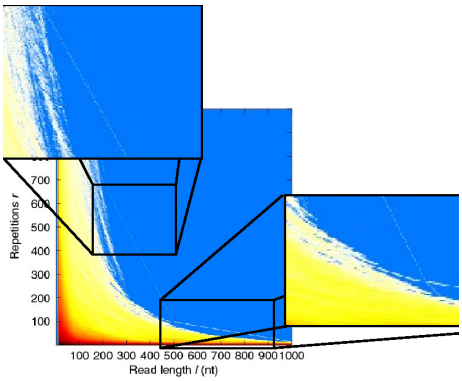
ATACCACCATGCCTCCTTGCTCCAATCCACCATCAAGGCG

- How many repeats are there in the human genome? (Incidentally the human genome is 3.2 billion base **pairs**)
- This is an important question for developing new sequencing technologies

Repeats in Human Genome



Repeats Structure



Computing Repeats

- A naive program would take n^2 operations where $n = 6.4 \times 10^9$
- If we used this we would still be waiting for the program to finish
- Could not answer this question a few years ago—not because computers weren't powerful, but because the algorithms had not been developed
- Used state-of-the-art suffix arrays
- Smart algorithms allow you to do things which you cannot do otherwise

To Use DSA You Need To

- Know what common data structures and algorithm do
- Understand the implementations enough to modify existing data structures to be fit for purpose
- Understand time/space complexity to select the right data structure or algorithm
- Understand software interfaces for DSA
- Be able to combine data structures
- The rest of this course teaches you these skills

Further Mathematics and Algorithms

Lesson 2: *Know How Long A Program Takes*

TSP, Sorting, time complexity, Big-Theta, Big-O, Big-Omega

Outline

1. **TSP**
2. Sorting
3. Big O

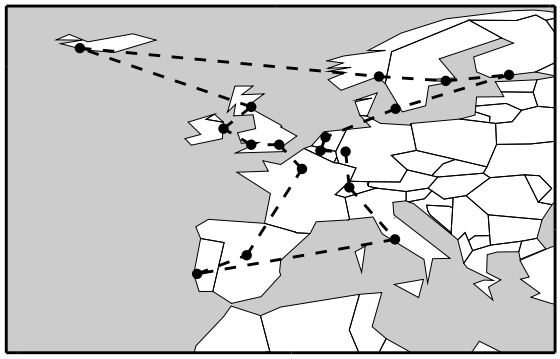
Travelling Salesperson Problem

- Given a set of cities
- A table of distances between cities
- Find the shortest tour which goes through each city and returns to the start

Example of Distance Table

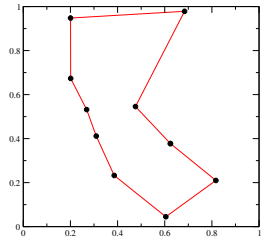
	Lon	Car	Dub	Edin	Reyk	Oslo	Sto	Hel	Cop	Amst	Bru	Bonn	Bern	Rome	Lisb	Madr	Par
London	0	223	470	538	1896	1151	1426	1816	950	349	312	503	743	1429	1587	1265	337
Cardiff	223	0	290	495	1777	1277	1589	1985	1139	564	533	725	927	1600	1492	1233	492
Dublin	470	290	0	350	1497	1267	1628	2026	1239	756	775	956	1207	1886	1638	1449	777
Edinburgh	538	495	350	0	1374	933	1314	1708	984	662	758	896	1243	1931	1964	1728	872
Reykjavik	1896	1777	1497	1374	0	1746	2134	2418	2104	2020	2130	2255	2617	3304	2949	2892	2232
Oslo	1151	1277	1267	933	1746	0	416	788	481	917	1088	1048	1459	2011	2739	2390	1343
Stockholm	1426	1589	1628	1314	2134	416	0	398	518	1126	1281	1181	1542	1978	2987	2593	1543
Helsinki	1816	1985	2026	1708	2418	788	398	0	881	1504	1650	1530	1856	2203	3360	2950	1910
Copenhagen	950	1139	1239	984	2104	481	518	881	0	625	769	662	1036	1538	2479	2076	1030
Amsterdam	349	564	756	662	2020	917	1126	1504	625	0	173	235	629	1296	1860	1480	428
Brussels	312	533	775	758	2130	1088	1281	1650	769	173	0	194	489	1174	1710	1315	262
Bonn	503	725	956	896	2255	1048	1181	1530	662	235	194	0	422	1067	1843	1420	400
Bern	743	927	1207	1243	2617	1459	1542	1856	1036	629	489	422	0	689	1630	1156	440
Rome	1429	1600	1886	1931	3304	2011	1978	2203	1538	1296	1174	1067	689	0	1862	1365	1109
Lisbon	1587	1492	1638	1964	2949	2739	2987	3360	2479	1860	1710	1843	1630	1862	0	500	1452
Madrid	1265	1233	1449	1728	2892	2390	2593	2950	2076	1480	1315	1420	1156	1365	500	0	1054
Paris	337	492	777	872	2232	1343	1543	1910	1030	428	262	400	440	1109	1452	1054	0

Example Tour

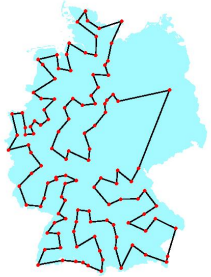


Brute Force

- I wrote a program to solve TSP by enumerating every path and finding the shortest
- I checked that it worked on some problems with 10 cities
- It takes just under half a second to solve this problem
- I set the program running on a 100 city problem—**How long will it take to finish?**

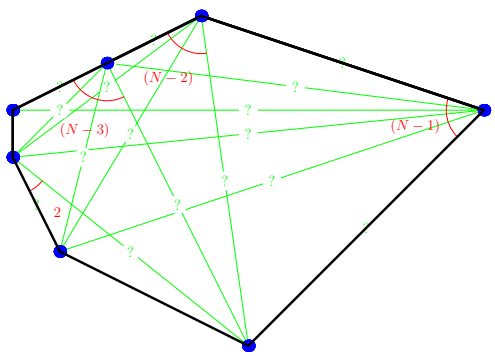


How Many Possible Tours Are There?



- For 100 cities how many possible tours are there?
- It doesn't matter where we start
- Starting from Berlin there are 99 cites we can try next

Counting Tours



Number of tours = $(N-1) \times (N-2) \times (N-3) \times \cdots 2 \times 1 = (N-1)!$

How Long Does It Take?

- The direction we go in is irrelevant
- Total number of tours is $99!/2$
- **Any more guesses how long it will take?**

How Big is 99 Factorial?

- $99! = 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 = ?$
- Upper bound

$$\begin{aligned} 99! &= 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 \\ 99! &< 99 \cdot 99 \cdot 99 \cdots 99 \cdot 99 = 99^{99} \end{aligned}$$

- Lower bound

$$\begin{aligned} 99! &= 99 \cdot 98 \cdot 97 \cdots 50 \cdot 49 \cdots 2 \cdot 1 \\ 99! &> 50 \cdot 50 \cdot 50 \cdots 50 \cdot 1 \cdots 1 \cdot 1 = 50^{50} \end{aligned}$$

How Long Does It Take?

- For $N > 1$

$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

- $99!/2 = 4.666 \times 10^{155}$
- How long does it take to search all possible tours?
  - We computed about $200\,000$ tours in half a second
  - 3.15×10^7sec $= 1$ year
  - Age of Universe ≈ 15 billion years

Answer

- 2.72×10^{132} ages of the universe!
- Incidental

$$\begin{aligned} 99!/2 = {} & 46663107721972076340849619428133350 \\ & 24535798413219081073429648194760879 \\ & 99966149578044707319880782591431268 \\ & 48960413611879125592605458432000000 \\ & 0000000000000000 \end{aligned}$$

Record TSP Solved—15 112 and 24 978 Cities

In Case You're Curious

- Number of tours: $15111!/2 = 7.3 \times 10^{56592}$
- Current record $24\,978$ cities with 1.9×10^{98992} tours
- The algorithm for finding the optimum path does not look at every possible path
- If your interested look for the TSP homepage on the web `http://www.math.uwaterloo.ca/tsp/`

Lessons

- Even relatively small problems can take you an astronomical time to solve using simple algorithms
- As a professional programmer you need to have an estimate for how long an algorithm takes—otherwise you can look silly
- For the 100 city problem, if
  - I had 10^{87} cores, one for every particle in the Universe
  - I could compute a tour distance in 3×10^{-24} seconds, the time it takes light to cross a proton
  - It would still take $10^{39}\times$ the age of the universe
- Smart algorithms can make a much larger difference than fast computers!

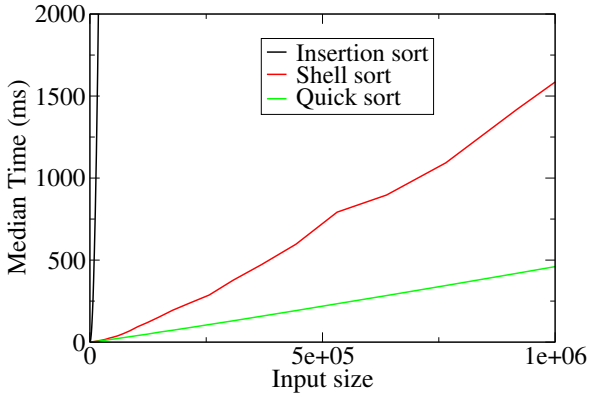
Outline

1. TSP
2. **Sorting**
3. Big O

Sort

- Comparison between common sort algorithms
  - Insertion sort—an easy algorithm to code
  - Shell sort—invented in 1959 by Donald Shell
  - Quick sort—invented in 1961 by Tony Hoare
- These take an array of numbers and returns a sorted array
- Sort is very commonly used algorithm so you care about how long it takes

Empirical Run Times



Lessons

- There is a right and wrong way to do easy problems
- You only really care when you are dealing with large inputs
- Good algorithms are difficult to come up with, but they exist
- We would like to quantify the performance of an algorithm—how much better is quick sort than insertion sort?

Outline

1. TSP
2. Sorting
3. **Big O**

Estimating Run Times

- We would like to estimate the run times of algorithms
- This depends on the hardware (how fast is your computer)
- We could count number of elementary operations but
  - different machines have different elementary operations
  - many algorithms use complex functions such as `sqrt` (matrix inversion using Cholesky decomposition) or `sin` and `cos` (FFT)
  - would need to count memory accesses which you shouldn't need to think about
  - code after compiling can be very different from code before compiling

Engineering Solution

- Compute the **asymptotic leading functional behaviour**
- Lets take that statement to pieces
- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ operations (clock cycles)
  - **asymptotic**: what happens when n becomes very large
  - **leading**: ignore the $12n + 199$ part as it is dominated by $4n^2$ (i.e. for large enough n we have $4n^2 \gg 12n + 199$)
  - **functional behaviour**: ignore the constant 4
- We call this an order n^2, or quadratic time, algorithm
- We can write this in 'Big-Theta' notation as $\Theta(n^2)$
- This notion of 'run time' is known as **time complexity**

Advantages of Big-Theta Notation

- Doesn't depend on what computer we are running
- Don't need to know how many elementary operations are required for a non-elementary operation
- Can estimate run times by measuring run time on a small problem
  - If I have a $\Theta(n^2)$ algorithm
  - It takes x seconds on an input of 100
  - It will take about $\frac{x \times n^2}{100^2}$ seconds on a problem of size n
    ($T(100) \approx c\,100^2 = x$ therefore $c = x/100^2$
    thus $T(n) = c\,n^2 = x\,n^2/100^2$)

Counting Instructions

- Big-Theta run times are often easy to calculate
- a $\Theta(n)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
}
// clean up
```

- a $\Theta(n^2)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  for (int j=0; j<n; j++) {
    // do other stuff
  }
}
// clean up
```

Disadvantage with Big-Theta notation

- Can't compare algorithms with the same Big-Theta time complexity
- For small inputs Big-Theta time complexity can be misleading. E.g.
  - algorithm A takes $n^3 + 2n^2 + 5$ operations
  - algorithm B takes $20n^2 + 100$ operations
  - algorithm A is $\Theta(n^3)$ and algorithm B is $\Theta(n^2)$
  - algorithm A is faster than algorithm B for $n < 18$

  but who cares?
- In some cases Big-Theta time complexity is hard to compute

Not So Sure

- Some algorithms are harder to compute

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- Time complexity now depends on the `if` statement
- If the condition is often satisfied we have a $\Theta(n^2)$ algorithm
- If the condition is true only rarely then we have a $\Theta(n)$ algorithm

Bounds

- To avoid having to think really hard we define upper and lower bounds
- The upper bound we write using **big-O** notation
  - The above algorithm is an $O(n^2)$ algorithm
  - I.e. it runs in no more than order n^2 operations
- The lower bound we write using **big-Omega** notation
  - The above algorithm is a $\Omega(n)$ algorithm
  - I.e. it runs in no less than order n operations

Precise Definitions of $O(n)$

- An algorithm that runs in $f(n)$ operations is $O(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \qquad \text{where } c \text{ is a constant (could be zero)}$$

- E.g.. $f(n) = 3\,n^2 + 2\,n + 12$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3\,n^2 + 2\,n + 12}{n^2} = 3 \quad \Rightarrow 3\,n^2 + 2\,n + 12 = O(n^2)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3\,n^2 + 2\,n + 12}{n^3} = 0 \quad \Rightarrow 3\,n^2 + 2\,n + 12 = O(n^3)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3\,n^2 + 2\,n + 12}{n} = \infty \quad \Rightarrow 3\,n^2 + 2\,n + 12 \neq O(n)$$

Lower Bound Definition

- An algorithm that runs in $f(n)$ operations is $\Omega(g(n))$ if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c \qquad \text{where } c \text{ is a constant (could be zero)}$$

- E.g. $f(n) = 3\,n^2 + 2\,n + 12$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^2}{3\,n^2 + 2\,n + 12} = \frac{1}{3} \Rightarrow 3\,n^2 + 2\,n + 12 = \Omega(n^2)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^3}{3\,n^2 + 2\,n + 12} = \infty \Rightarrow 3\,n^2 + 2\,n + 12 \neq \Omega(n^3)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n}{3\,n^2 + 2\,n + 12} = 0 \quad \Rightarrow 3\,n^2 + 2\,n + 12 = \Omega(n)$$

Big-Theta

- An algorithm that runs in $f(n)$ operations is $\Theta(g(n))$ if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c \qquad \text{where } c \text{ is a non-zero constant}$$

- That is, $f(n) = \Theta(g(n))$ if

$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$

- I.e. the lower bound is identical to the upper bound
- Often the most straightforward way of obtaining big-Theta is to show the upper and lower bounds are the same

Use and Misuse

- Note: big-O notation is most commonly used
- often people say they have a $O(n^2)$ when in fact they mean they have a $\Theta(n^2)$ algorithm (a much stronger result)
- Note that an $O(n^2)$ algorithm is also a $O(n^3)$ algorithm
- Strictly a $O(n^2)$ algorithm **may not** be faster than a $O(n^3)$ algorithm when n becomes larger
- A $\Theta(n^2)$ algorithm **will** be faster than a $\Theta(n^3)$ algorithm when n becomes larger

Lessons to Learn

- Run times (computational time complexity) matters
- Choosing an algorithm with the best time complexity is important
- Understand the meaning of big-Theta, big-O and big-Omega
- Know how to estimate time complexity for simple algorithms (loop counting)

Further Mathematics and Algorithms

Lesson 3: *Declare your intentions (not your actions)*

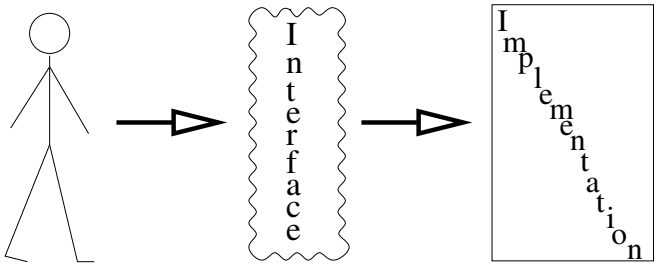
ADTs, stacks, queues, priority queues, sets, maps

Outline

Object Oriented Programming

- OO-programming allows you to build large systems reliably
- In the OO-methodology you separate the interface from the implementation
- The **interface** is the public methods (functions) of a class
- The implementation is hidden (**encapsulated**) and may be changed without affecting how the class is used
- There exist other ways of programming, but C++ is designed to support the OO-methodology—for building systems it is brilliant

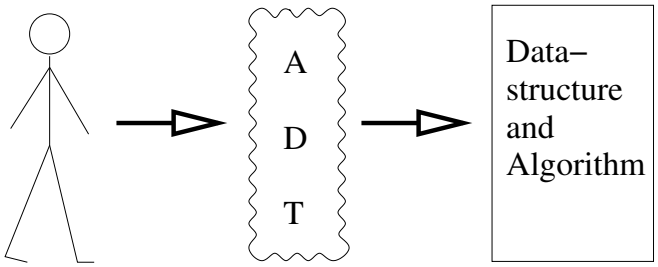
Object-Oriented Classes



Abstract Data Types

- With data structures there are some traditional interfaces called **Abstract Data Types** or ADTs
- These are implementation free data structures
- They are mathematical abstractions of the data structure
- Their purpose is to allow you to declare you intentions
- You are entering into an agreement that you only intend to use the underlying data structure in the way specified by the interface

ADTs



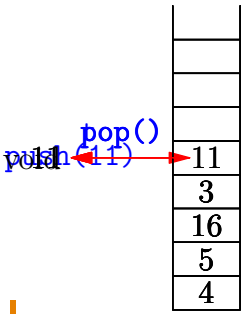
Say it with an ADT

- Common ADTs include stacks, queues, priority queues, sets, multisets and maps
- There are many possible implementations of these ADTs (some far from obvious)
- Each ADT has a limited set of methods associated with it
- They are an abstraction away from the implementation
- By declaring your intentions you are making your code easier to understand and maintain

Outline

Stacks

- Last In First Out (LIFO) memory
- Standard functions
  - `push(item)`
  - `T top()`
  - `T pop()` except in C++ `pop()` doesn't return the top of the stack
  - `boolean empty()`
- Implemented using an array (or a linked-list)



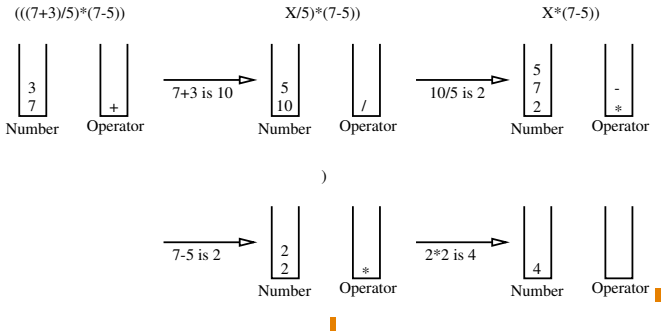
Why Use a Stack?

- Stacks reduces the access to memory—no longer random access
- Seems counter intuitive to reduce what you can do
- Gives you a very simple interface
- Prevents another programmer from using memory in a way that will break existing code
- Sufficient for large number of algorithms

Uses of Stacks

- Reversing an array
- Parsing expression for compilers
  - balancing parentheses
  - matching XML tags
  - evaluating arithmetic expression
- Clustering algorithm

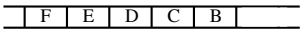
Evaluating Arithmetic Expressions



Outline

Queues

- First-in-first-out (FIFO) memory model
- `enqueue(elem)`
- `peek()`
- `dequeue()`
- C++ has a double ended queue (`deque`) with `push_front()`, `push_back()`, etc.

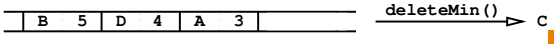


Uses of Queues

- Queues are heavily used in multi-threaded applications (e.g. operating systems)
- Multi-threaded applications need to minimise waiting and ensure the integrity of the data structure (for instance when an exception is thrown)
- Because of this they are more complicated than most data structures
- They can be implemented using linked-lists or circular arrays

Priority Queues

- Queue with priorities
- `insert(elem,priority)` (in C++ `push()`)
- `findMin()` (in C++ `top()`)
- `deleteMin()` (in C++ `pop()`)



Uses of Priority Queues

- Queues with priorities (e.g. which threads should run)
- Real time simulation
- Often used in "greedy algorithms"
  - Huffman encoding
  - Prim's minimum spanning tree algorithm

Implementation of Priority Queue

- Could be implemented using a binary tree or linked list
- Most efficient implementation uses a heap
- A heap is a binary tree implemented using an array

Outline

Lists

- In C++ the standard list is known as `vector<T>`
- That is, it is a collection where the order in which you put items into the list counts
- You can have repetitions of elements
- It has random access, e.g. `v[i]`
- You can `push_back(i)`, `insert`, `erase`, etc.
- C++ has a linked list class `List<T>`

Sets

- Models mathematical sets
- Container with no ordering or repetitions
- Methods include `insert`, `find`, `size`, `erase`
- Provides fast search (`find`)
- This is the class to use when you have to rapidly find whether an object is in the set or not—don't use a list like `vector<T>`!

Iterators

- Wish to act on all members of the set
- Performed using an `iterator`
- Iterators are used by many collections
- In C++ iterators follow the pointer convention

```
set<string> words;

words.insert("hello");
words.insert("world");

for(auto iter = words.begin(); iter != words.end(); ++iter) {
  cout << *iter << endl;
}
```


Implementation of Sets

- Sets are very important and there are many implementations depending on their usage
- Two common implementations of sets are
  - hash tables: `unordered_set<T>`
  - binary trees: `set<T>`
- Which is most efficient depends on the application
- Binary trees allow you to iterate in order (iterating over a hash table will give you outputs in random order)
- `multiset<T>` are sets with repetition

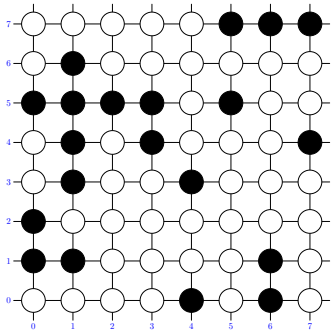
Maps

- A map provides a content addressable memory for pairs *keyword: data*
- It provides fast access to the data through the keyword
- Implement as tree or hash table
- Multimaps allows different data to be stored with the same keyword

Outline

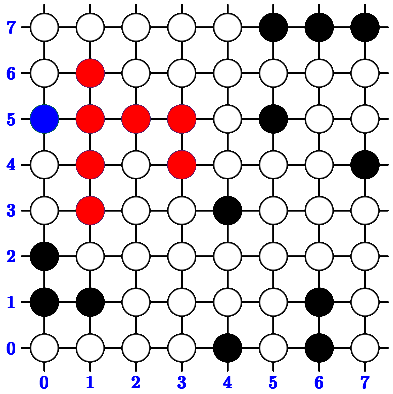
1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. **Putting it Together**

Connected Nodes



- A frequent problem is to find clusters of connected cells
- Applications in computer vision, computer go, graph connectedness, . . .

Connected Nodes



next = (2,5)

uncheckedNodes =

(1,6)
(3,4)
(0,5)

clusterNodes =
{(2,5),(1,5),(3,5),}
(3,4),(0,5),(1,4),
(1,6),(1,3)}

Connected Node Algorithm

```
set<Node> findCluster(Node startNode, Graph graph)
{
  stack<Node> uncheckedNodes = new Stack<Node>();
  set<Node> clusterNodes = new HashSet<Node>();

  uncheckedNodes.push(startNode);
  clusterNodes.add(startNode);

  while (!uncheckedNodes.empty()) {
    Node next = uncheckedNodes.top();   uncheckedNodes.pop();
    vector<Node> neighbours = graph.getNeighbours(next);
    for (Node neigh: neighbours) {
      if (graph.isOccupied(neigh) && !clusterNodes.contains(neigh) ) {
        uncheckedNodes.push(neigh);
        clusterNodes.insert(neigh);
      }
    }
  }

  return clusterNodes;
}
```


Lessons

- Abstract Data Types (ADT) are interfaces to data
- Their purpose is to allow the programmer to declare their intentions
- They often have different implementations with different properties
- The most efficient implementation is not always obvious—we will see many of these implementations as we go through this course
- You need to know the common ADTs (e.g. Stack, Queue, List, Set, Map) and how and when to use them

Lesson 4: *Use Arrays*

Variable length arrays, implementing stacks

Outline

Use Arrays

- An array is a contiguous chunk of memory
- In C we can create arrays using
  `int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance, although never use raw arrays!

Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
  - E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are—annoying

Outline

Variable Length Arrays

- We want a variable length array
- Initially a variable length array would have length zero
- We should be able to
  - Add an element to an array
  - Access any element in the array
  - Change an element
  - Delete elements
  - Know how many elements we have

ADT for a List

- What do we want of a list of **int**s?
  - `void push_back(int value)`
  - random access `array[i]`
  - `int size()`
- It would be useful if it resized
- It would be great to have some algorithms (e.g. sort) that can be run on a list

Implementation

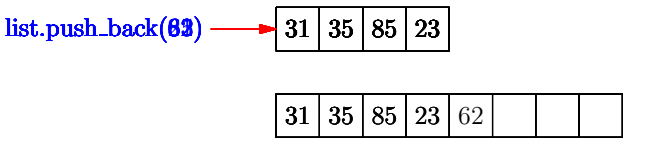
- How should we implement a list?
- Use an array, of course!
- We need to distinguish between
  - the number of elements in the list `size()`
  - the number of elements in the array `capacity()`
- If the number of elements grows larger than the capacity then we need to increase the capacity

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use cn bits of memory where we want to keep c small

Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
  - create a new array with double the capacity of the old array
  - copy the existing elements from the old array to the new array



Amortised Time Analysis

- How efficient is resizing?
- Most `push_back(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- Adding to a full array is slow but it is **amortised** by other quick adds

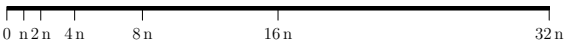
  amortised: effect of a single operations 'deadened' by other operations

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
  - adds: 100
  - copies: 10+20+40+80=150
  - **new int[]**: 4
- 250 adds and copies operations + 4 **new** operations

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}n = n(1 + 2 + \cdots + 2^{m-1}) = n\,(2^m - 1)$$

- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n\,(2^m - 1) = N + n2^{\left\lceil \log_2\left(\frac{N}{n}\right)\right\rceil} - n < N + 2N - n < 3N$$

Insertion and Deletion

- `vector<T>` is very useful and very fast for lots of things
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order (binary trees will do that for you much more efficiently)
- Linked lists allow you to splice in a sublist into a list in constant time although linked lists have a lot of drawbacks

Outline

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C), it pays the price of allowing bugs that hard to detect
- **Java** was designed to be vary safe (avoiding lots of bugs), but is not fast and a bit long winded
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code, but it is not fast or safe

Problems with C++

- Amongst a number of issues that make C++ dangerous are
  - Memory management
  - Writing to parts of memory that you should not
  - Multiple inheritance, although you seldom need to do this
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue

Memory Management

- Most programming languages have two types of memory

  The Stack: is the area of memory controlled by compiler for local variables, function calls, etc.

  The Heap: is area that the programmer (you) can request, which is nice
- In C++ you are given the **right** to ask for memory

```
int *storage = new int[n];
```

- You have **responsibility** to free the memory

```
delete[] storage;
```


Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapid exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use: `delete[] storage;`
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

Resource Acquistion is Initialisation (RAII)

- Java and Python use garbage collectors which automatically checks whether memory can be accessed and if not it is removed
- In C++ this is your responsibility
- But the is a standard **programming pattern** to elevate the problem known as **Resource Acquistion is Initialisation (RAII)**

  Wrap all resources in classes. Request the resources in the constructor and release the resource in the destructor
- When the object goes out of scope (you leave a `for` loop, function call, etc.) the destructor is called and the resource is safely released

Writing over Memory

- In C++ the following will compile and run

```
int *array = new int[4];
int *a = new int[2];
double *darray = new double[4];
array[4] = 4;
```

- However `array[4]` has not been assigned (unlike `array[0]`, `array[1]`, `array[2]` and `array[3]`)
- The memory on the heap corresponding to the address of `array[4]` might have been assigned to `a[0]` in which case you may inadvertently have set `a[0]` to 4 leading to the program not doing what you want
- It might be that you have put an `int` into `darray[0]` which will then crash the system when you read `darray[0]`

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

Follow Programming Idioms

- Using common data structures and following common idioms will prevent most errors

```
int n = 5;
vector<int> array(n);

for(int i=0; i<array.size(); ++i) {
  array[i] = i;
}

for(auto pt=array.begin(); pt != array.end(); ++pt){
  *pt *= 2
}

for(int& element: array) {
  element += 2;
}
```


Outline

Stacks

- Lets look at implementing a stack
- Remember a stack has methods
  - `push(Object )`
  - `pop()`
  - `top()`
  - `empty()`

Implementation of Stack

```
template <typename T>
class MyStack
{
private:
  std::vector<T> stack;

public:
  void push(const T& obj) {stack.push_back(obj);}

  T top() const {return stack.back();}

  T pop() {
    T tmp = stack.back();
    stack.pop_back();
    return tmp;
  }

  T empty() {return stack.size()==0;}
};
```


Notes on Implementation

- I don't need to write a constructor as C++ generates a default constructor that will initialise the stack correctly
- I don't need to write a desctuctor because by default the destructor for `vector<T>` will be called which releases memory
- I've written the `pop` command, that I like, but if I run

```
stack<Widget> widget_stack;
Widget w;
widget_stack.push(w);
Widget w1(widget.pop());
```

  if the last command throws an exception then the last term on the stack is lost for ever

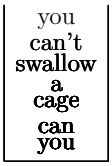
Why not use a `vector`

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory—I don't need to write comments as it is clear what I am doing

Using `MyStack`

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you can't swallow a cage can you



Reversing Strings in File

```
#include <stack>
#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>
using namespace std;

int main(int argc, char *argv[]) {
  ifstream in(argv[1]);

  stack<string> stack;

  string word;
  while (in >> word)
    stack.push(word);

  while(!stack.empty()) {
      cout << stack.top() << ' ';
      stack.pop();
  }
}
```


Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays frequently!

Lesson 5: *Writing an Arrays*

Writing data structures in C++

- This is not being taught as a lecture, but run as a practical session
- We are going to write a resizable array class in C++
- I will call the class Array although this is not a great name
- The point of this is to understand the subtleties of coding is C++

Complied code

- C++ like C must be compiled and linked
- Compiling turns the code into machine code (*.o files) with calls to external libraries
- Linking actually links the external libraries with the code to produce an executable file
- We use a `Makefile` to do the compile and linking

```
all: main run

main: array.h array.cc main.cc
    g++ main.cc array.cc -o main

run: main
    ./main
```


Cpp Style Classes: main.cc

```
#include <iostream>
#include "array.h"
using namespace std;

int main() {
  Array a(3);
  a.set(0,0);
  a.set(1,2);
  a.set(2,4);

  cout << a.get(0) << ", " << a.get(1) << ", " << a.get(2) << endl;

  return 0;
}
```


array.h

```
#ifndef ARRAY_H
#define ARRAY_H

class Array {
private:
  int *data;
public:
  Array(int n);
  void set(int index, int value);
  int get(int index);
};

#endif
```


array.cc

```
#include "array.h"

Array::Array(int n) {
  data = new int[n];
}

void Array::set(int index, int value) {
  data[index] = value;
}

int Array::get(int index) {
  return data[index];
}
```


Operator Overloading

- Cpp is just ugly
- C++ allows us to overload operators (e.g. +, +=, <<, etc.)
- One operator is indexing: `operator[int]()`
- We can use this to return a reference to `data[i]`

```
int& Array::operator[](int index) {
  return data[index];
}
```


Updated main.cc

```
#include <iostream>
#include "array.h"
using namespace std;

int main() {
  Array a(3);

  for(int i=0; i<3; i++) {
    a[i] = i*i;
  }

  cout << a[0] << ", " << a[1] << ", " << a[2] << endl;

  return 0;
}
```

Adding Power

- As we might want to print different arrays lets create a print function
- We want Array to know how many elements are in it

```
#ifndef ARRAY_H
#define ARRAY_H

class Array {
private:
  int *data;
  int length;
public:
  Array(int n);
  int& operator[](int index);
  int size();
};

#endif
```


main.cc

```
#include <iostream>
#include "array.h"
using namespace std;

void print(Array& a, string name) {
  cout << name;
  for(int i=0; i<a.size(); i++) {
    cout << " " << a[i];
  }
  cout << endl;
}

int main() {
  Array a(10);

  for(int i=0; i<a.size(); i++) {
    a[i] = i*i;
  }

  print(a, "a:");

  return 0;
}
```


Copy Constructor

- C++ conveniently generates a copy constructor

```
Array b(a);
```

- Unfortunately this copies the address to `data` and the `length`
- But his is a *shallow copy* which means that both arrays work on the same data array
- This would be deeply confusing. Instead we have to write our own *copy constructor* to do a deep copy

```
Array::Array(Array& other) {
  data = new int[other.size()];
  length = other.size();
  for(int i=0; i<size(); ++i) {
    data[i] = other[i];
  }
}
```

Lesson 6: *Writing an Arrays*

Common errors, memory leaks, templates

- These are notes on the tutorial session on writing a resizeable array
- We did not get very far with them in the first lecture
- In this lecture we are going to make are code more solid
- Add some functionality
- Make the code generic

Copy Constructor

- C++ conveniently generates a copy constructor

```
Array b(a);
```

- Unfortunately this copies the address to `data` and the `length`
- But his is a *shallow copy* which means that both arrays work on the same data array
- This would be deeply confusing. Instead we have to write our own *copy constructor* to do a deep copy

```
Array::Array(Array& other) {
  data = new int[other.size()];
  length = other.size();
  for(int i=0; i<size(); ++i) {
    data[i] = other[i];
```



```
  }
}
```


Assignment Constructor

- We can also generate a new array through assignment

```
Array a = b;
```

- As with the copy constructor this is generated by default
- However, it calls the copy constructor
- If we fix the copy constructor this now works as expected
- Almost . . .

Being Explicit

- One oddity of C++ is that the following code compiles

```
Array a = 4;
```

- We have not defined what happens when we set and array to an integer
- However, the compiler tries to make sense of this and sees that it can create an array on the right-hand side using the constructor

```
Array(int n);
```

- It sees this as a way of promoting an integer to an array
- This isn't what most people would expect. I expect a compile error
- To achieve this I can redefine the constructor

```
explicit Array(int n);
```


Compilers are our Friends

- Compile errors are our friends: they are quick to fix and prevent serious errors
- One little understood strength of C++ is the compiler allows us to determine what changes
- Defining the function

```
void print(const Array&, string name);
```

passes the array by const reference. This is efficient. Making it const means we know print won't change the reference

- But this triggers a whole lot of consequence because print is only allowed to use const member functions

Constant consistency

- At first it appears we have opened a can of works
- We have declare lot of member functions as constant

```
int size() const;

int& operator[](int index);
int operator[](int index) const;
```

- We have to declare a constant version of the access operator
- When you first do this it seems like a lot of unnecessary work
- But the is some satisfaction in specifying all the functions consistently
- And in the long run it will prevent many bugs

Memory Leaks

- Another "bug" in our code is that we are grabbing memory, but not giving it up
- This can become very expensive

```
for(int i=0; i<500000; i++) {
  Array a(10000000);
  if (i % 10000==0) {
    cout << i << endl;
    sleep(1);
  }
  cout << "Finished\n";
}
```

- In linux I can look at memory usage using `top -c $(pgrep -d',' 'main)$`

RAII

- The method for preventing memory leaks is known as "Resource Allocation is Initialisation"
- This means we take the resource (in this case memory) in the constructor of a class
- And give it back in the destruction

```
Array::~Array() {
  delete[] data;
}
```


Make it Generic

- To make an array for doubles or strings we only have to change the type of the data from `int` to `double` or `string`
- We can write a template with `T` (or any other name we want to use) as representing some generic type
- We can't compile the code as the compiler needs to know the type we are using
- We would therefore have to do a global replace of `T` by the type we want to use and create new `array.h` and `array.cc` for each type of array we use
- Fortunately the C++ compiler will do this for us

Template Programming

- To do this all we need to do is write **`template <typename T>`** in front of any class or function that uses a generic type
- These need to be included in the header file
- In the main code to ask for an array of type `string`, for example, we write

```
Array<string> string_array;
```

- The compiler invisibly creates the code for this class, but replacing the template variable by `string`

Template Code Example

```
template <typename T>
class Array {
private:
  T *data;
  unsigned length;
public:
  explicit Array(int n);
  Array(const Array& other);
  ~Array();
  T& operator[](unsigned index);
  T operator[](unsigned index) const;
  unsigned size() const;
};

template <typename T>
Array<T>::Array(int n) {
  data = new T[n];
  length = n;
}
```

Further Mathematics and Algorithms

Lesson 7: *Writing an Arrays*

Resizeable arrays, iterators

Resizable Arrays

- We are finally in a position where we can make our array resizable
- We need to have both a length (which the user know about) and a capacity which is invisible to the user, but allows us to add elements while not having to create too many new arrays
- We need to change the constructor

```
template <typename T>
Array<T>::Array(unsigned n=0) {
  length = n;
  if (n==0) {
    n = 8;
  }
  data = new T[n];
  capacity = n;
}
```

- If we don't give the constructor an argument it will be set to 0

Push Back

- Let's introduce a new command that allows us to add to the end of an array
- If the capacity is not big enough we need to increase the capacity

```
template <typename T>
T Array<T>::push_back(T value) {
  if (length == capacity) {
    capacity *= 2;
    T* new_data = new T[capacity];
    for (int i=0; i<length; ++i) {
      new_data[i] = data[i];
    }
    delete [] data;
    data = new_data;
  }
  data[length] = value;
  ++length;
  return value;
}
```


Refactoring

- It is possible to define functions inside the class definition
- This has the advantage of making the code much more compact
- It has the disadvantage of confusing the interface (a list of function that can be called) with the implementation
- It is unfortunate that when you use templates you tend to put the implementation details in the header file
- What you should do depend on whether you are after a quick solution or are writing code that many people might use

New Code

```
template <typename T>
class Array {
private:
  T *data;
  unsigned length;
  unsigned capacity;
public:
  Array(unsigned n=0) {
    length = n;
    if (n==0) {
      n = 8;
    }
    data = new T[n];
    capacity = n;
  }
  Array(const Array& other);
  ~Array() {delete[] data;}
  T& operator[](unsigned index) {return data[index];}
  const T& operator[](unsigned index) const {return data[index];}
  unsigned size() const {return length;}
  T push_back(T value);
};
```


Iterators

- Iterators are designed to follow the same semantics as pointers
- Because we are just wrapping an array where we can use pointers to navigate the array we can use pointers as iterators
- All we need to do is add two new methods

```
T* begin() {return data;}
T* end() {return data+length;}
```

- We can the use iterators to iterate over our array

vector

- C++ comes with a powerful library known as the **standard template library** (STL)
- This includes containers (resizable arrays, linked lists, double ended queues, sets, maps, etc.)
- The resizable array is called `vector<T>`
- We can just replace "`array.h`" with `<vector>` and `Array` with `vector` in our main function and everything will work the same
- Of course the STL vector is a bit more powerful and efficient than the `Array` class we wrote, although our class isn't bad

Iterators in Other Containers

- Iterators exist for many containers
- Because of iterators it is possible to write algorithms that work for any container that supports iterators
- The STL has a bunch of algorithms that work for different containers
- You can also use a pretty for loop

```
for(T entry: container) {
  ...
}
```

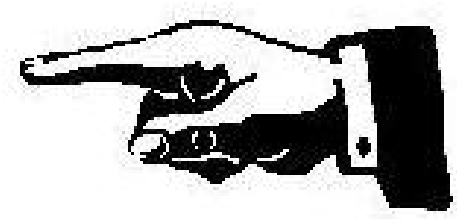
- This makes code easier to read

Writing Iterators

- For most classes we don't iterator by advancing a pointer
- Thus most iterators are more complex and we have to write a class (or structure—`struct`) which keeps the information we need to iterate
- We then have to define methods to make iteration look like iterating though an array
- We will see an example in the code for linked lists

Further Mathematics and Algorithms

Lesson 8: *Point to where you are going: links*



Linked lists

Outline

Non-Contiguous Data

- So far we have considered arrays where the data is stored in a contiguous chunk of memory
- This has the great advantage of allowing random access
- It has the disadvantage that it is expensive to add or remove data from the middle of the list or to rearrange the data
- A different approach is to use units of data that point to other units

Non-Contiguous Data Structures

- There are a lot of important data structures using non-contiguous memory
  - Binary trees
  - Graphs
- In this lecture we consider **linked-lists**
- This is a classic data structure which is almost entirely useless
- However, it serves as a good introduction to much more useful data structures

Self-Referential Classes

- The building block for a linked list is a node class

```
struct Node<T>
{
  Node(U value, Node<U> *node): value(value), next(node) {}
  T element;
  Node<T> *next;
}
```

- We create new nodes
  `Node<int> *node = new Node<int>(10, pt_to_next)`
- Note that `node` is the address of this node
- I make it a `struct` as this is a class where I want public access to the element and next
- I can make this class a private class of my linked list

Outline

Singly Linked List

- We can build a linked list by stringing nodes together

  We don't show the "pointer" to `element`
- A singly linked list has a single "pointer" to the next element
- A doubly linked list has "pointers" to the next and previous element—we will see this later
- We should be able to create a linked list, add elements, remove elements, see if an element exists, etc.

Implementation

- We consider a lightweight implementation
- The class will have a head, a size counter and have a `Node` as a nested class

```
class MyList {
private:
  template <typename U>
  struct Node{
    Node(U value, Node<U> *node): value(value), next(node) {}
    U value;
    Node<U> *next;
  };
  Node<T> *head;
  unsigned noElements;
```

Simple Methods

- The constructor is simple (and not strictly necessary)

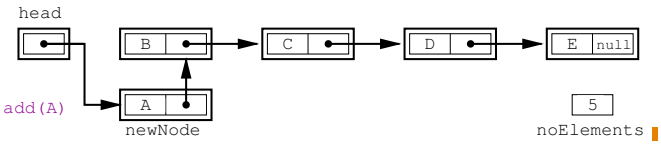
```
MyList(): n(0), head(0) {}
```

- Other simple methods are

```
unsigned size() const {return noElements;}

bool empty() const {
  return head == 0;
}
```

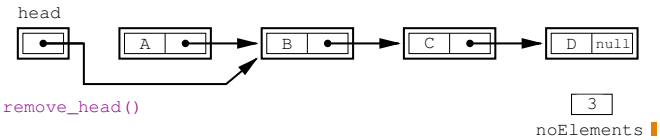

Adding elements

```
void add(T element)
{
    Node<T> *newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```



Remove Head of List

```
void remove_head()
{
    Node<T>* dead = head;
    head = head->next
    noElements--;
    delete dead;
}
```



Other Methods

- We can easily implement many other methods
  - `get(int i)`—return i^{th} item in list
  - `remove(T obj)`-remove `obj` from list
  - `insert(int position, element)`
- Note that `get(int i)` requires moving down the list so is $O(n)$ (i.e. not random access)

Outline

Stack

- It is easy to implement a stack using a linked list

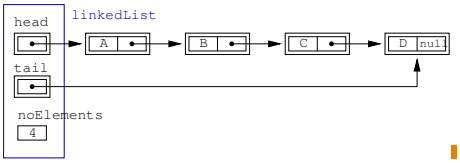
```
template <typename T>
class Stack<E>
{
  private Mylist<T> list = new mylist<T>();

  boolean push(E obj) {list.add(obj);}

  E top() {return list.get_head();} // throw exception

  E pop() {
    T tmp = list.get_head();
    list.remove_head();
    return tmp;
  }

  boolean empty() {return list.empty();}
}
```


Complexity of Stack

- All operations of the stack is constant time, i.e. $O(1)$
- This is the same time complexity as an array implementation
- Memory requirement is approximately $2 \times n$ reference and n objects—same as worst case for an array
- However, hidden cost of creating and destroying `Node` objects
- The array implementation is therefore slightly faster

Point to the Back

- To find the end of the queue takes n jumps
- Thus our linked list isn't the right data structure to implement a queue
- However, we could include a pointer to the end of the queue



Implementing a Queue

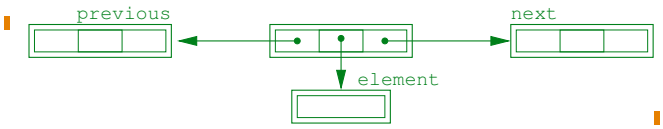
- We can then add elements to the tail in constant time
- We can the implement a queue in $O(1)$ time by
  - `enqueue`ing at the back
  - `dequeue`ing at the head
- I leave the implementation details as an exercise for you
- Note that although adding an element to the tail is constant time, removing an element from the tail is $O(n)$ as we have to find the new tail

Outline

Doubly linked list

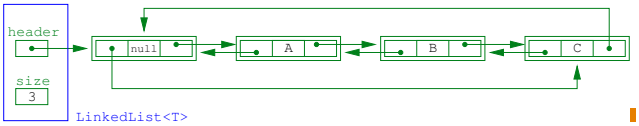
- In a more powerful linked list we would like to navigate the list in either direction
- To achieve this it uses a doubly-linked lists with elements to next and previous

```
class Node<T>
{
  T element;
  Node<T> *next;
  Node<T> *previous;
}
```



Dummy Node

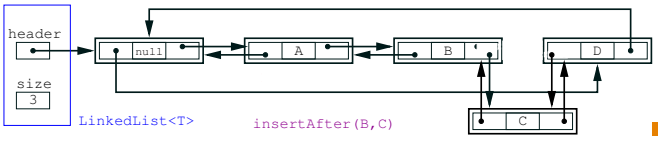
- List includes a dummy node—this make the implementations slicker



- Symmetric data structure so processing head and tail is equally efficient

Time Complexity

- `add` and `remove` from head and tail $O(1)$
- `find` $O(n)$ and slow
- `insert` and `delete` $O(1)$ (faster than an array list) once position is found



Outline

When To Use Linked Lists

- It is difficult to think of applications where linked lists are the best data structure
- lists—variable length arrays are usually better
- queues—linked list OK, but circular arrays are probably better
- sorted lists—binary trees much better
- linked lists have efficient insertion and deletion but it is difficult to think of an application where this matters

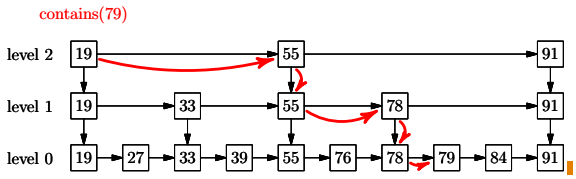
Line Editor

- One application where efficient insertion and deletion matters is a line editor
- We are usually working at a particular location in the text
- We often want to add or delete whole lines
- Storing the lines as strings in a linked list would allow a fairly efficient implementation

Outline

Skip Lists

- Linked lists have the disadvantage that to get to anywhere in the list takes on average $\Theta(n)$ steps
- Even if you kept an ordered list you still need to traverse it
- Skip lists are hierarchies of linked lists which allow binary search



Efficiency of Skip Lists

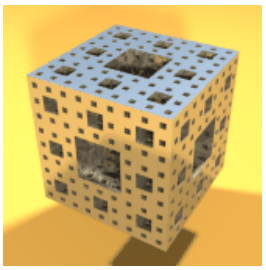
- Skip lists provide $\Theta(\log_2(n))$ search as opposed to $\Theta(n)$
- They have the similar time complexity to binary trees, although binary trees are slightly faster
- They have one advantage over binary trees—they allow efficient concurrent access
- The standard template library provides a doubly linked list, `list<T>` as well as a slingly linked-list `slist<T>`

Lessons

- Node structures that point to other Node structures are used in many important data structures
- Linked lists are the simplest examples of this kind of structure and consequently has a dominant position in most DSA books
- In practice linked lists are seldom the data structure of choice—before choosing to use a linked list consider the alternatives
- There are some important uses for linked lists, e.g. skip lists and hash tables (see lecture on hashing)

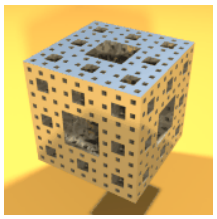
Further Mathematics and Algorithms

Lesson 9: *Recurse!*



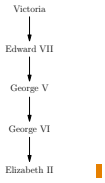
Induction, integer power, towers of hanoi, analysis

Outline



Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial: $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z



Structure of Recursion

- Notice that these are *self-referential* definitions
- A recursive definition consists of two elements
  - ⋆ **The Base Case:** or boundary cases where the problem is trivial
  - ⋆ **The Recursive Clause:** which is a self-referential part driving the problem towards the base case
- This should be reminiscent of proofs by induction—indeed the two are very closely related (many mathematical functions are defined recursively and there properties are proved by induction)

Recap on Proof by Induction

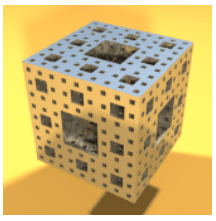
- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$ ✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} = S_n + 2^{n+1} = f(n) + 2^{n+1}$$
$$= 2^{n+1} - 1 + 2^{n+1} = 2 \times 2^{n+1} - 1 = 2^{n+2} - 1 \quad ✓$$

Outline



Programming Recursively

- Most modern programming languages, including C++, allow you to program recursively
- That is they allow functions/methods to be defined in terms of themselves

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    else if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

- This will work, is very intuitive, but is certainly not the best way to code a factorial function

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

$$0.95^{25} = 0.95 \times (0.95)^{24} = 0.95 \times \left((0.95)^{12}\right)^2 = 0.95 \times \left(\left((0.95)^6\right)^2\right)^2$$
$$= 0.95 \times \left(\left(\left((0.95)^3\right)^2\right)^2\right)^2 = 0.95 \times \left(\left(\left((0.95 \times (0.95)^2\right)^2\right)^2\right)^2$$

six multiplications rather than 24!

Implementing Integer Power

- Integer power looks rather intimidating to code
- However, the recursive definition is easy
- We can easily code this function recursively

```
double power(double x, long n)    // (Overflow is possible)
{
    return  n <  0  ? 1 / power(x,-n)          // Negative power
        :  n == 0   ? 1                        // Special case
        :  n == 1   ? x                        // Base case
        :  n%2 == 0 ? (x = power(x, n/2)) * x  // Even power
        :             x * power(x, n-1);       // Odd power
}
```


Helper Functions

- This is a slick implementation from the web, but not terribly efficient
- We only need to do the first two checks once
- A more efficient implementation would use a helper function

```
double power(double x, long n) { // (Overflow is possible)
    return  n <  0  ? power_recurse(1.0/x,-n)  // Negative power
        :  n == 0   ? 1                        // Special case
        : power_recurse(x,n);

}

double power_recurse(double x, long n) {
   return  n == 1  ? x                                 // Base case
       : n%2 == 0 ? (x = power_recurse(x, n/2)) * x    // Even powe
       :            x * power_recurse(x, n-1);         // Odd power
}
```


Writing Recursive Programs

- You need to make sure that you catch the base case **before** you recurse
- The recursive case can call itself, possibly many times, provided the inductive argument is closer to the base case
- That is,
  - Ensure that you use a 'smaller problem'
  - Assume that you can solve the 'smaller problem'

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - returns a value or
  - change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

Unrolling Recursion

- Recursion can frequently be replaced
- E.g. we can easily write a factorial function

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    long res = 1;
    for (int i=2; i<=n; i++)
        res *= i;
    return res;
}
```

with no function calls this will run much faster than the recursive version

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator
- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B
- E.g. the greatest common denominator of 70 and 25 is 5
- Euclid's algorithms uses the fact that
  - $\gcd(A, B) = \gcd(B, A \mod B)$
  - $\gcd(A, 0) = A$
- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{

     if (b==0) {
        return a;
     }
     return gcd(b, a%b);
}
```


Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{

     if (b==0) {
        return a;
     }
     long c = a%b;
     a = b;
     b = c;
     return gcd(a, b);
}
```

- Example of tail recursion

Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{
  while(true) {
     if (b==0) {
        return a;
     }
     long c = a%b;
     a = b;
     b = c;
  }
}
```

- Example of tail recursion

When Definitely not to Recurse

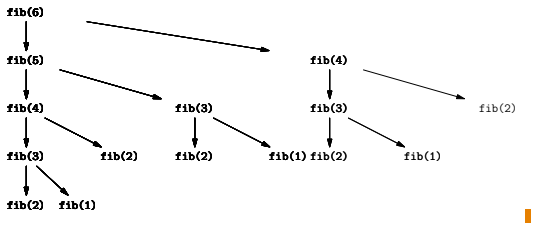
- A classic recursively defined sequence is the Fibonacci series
  - $f_n = f_{n-1} + f_{n-2}$
  - $f_1 = f_2 = 1$
- We might be tempted to write a recursive function to define the series

```
long fibonacci(long n)
{
  if (n<=2)
     return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Why shouldn't you want to do this?

Fibonacci

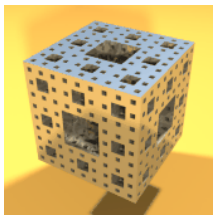
```
long fib(long n)
{
  if (n<=2)
  return 1;
  return fib(n-1) + fib(n-2);
}
```



Why Use Recursion At All?

- Both `factorial` and `gcd` could be written without using recursion
- The programs would probably run faster
- The `gcd` program would be less clear
- The cost of the additional function calls is often insignificant
- It would considerably harder to write many programs such as `power` non-recursively
- Later we will see algorithms like `quick sort` which rely on recursion

Outline



Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Time to Compute Power

- How long does it take to compute x^n?
- Remember

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- Thus

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \text{ is even} \\ T((n-1)/2) + 2 & \text{if } n \text{ is odd} \end{cases}$$
$$\leq T(\lfloor n/2 \rfloor) + 2$$

- Where $T(1) = 0$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - $b_i \in \{0, 1\}$
  - $b_m = 1$
  - m is the number of digits in the binary representation of n
  - $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - After $m-1$ 'divides' we reach 1
- Thus $T(n) \leq 2(m-1)$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \leq n < 2^{m+1}$$
$$m \leq \log_2(n) < m + 1$$

- But $T(n) \leq 2(m-1) \leq 2(\log_2(n) - 1) = \Theta(\log_2(n))$

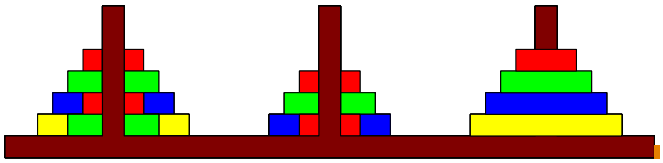
Towers of Hanoi

In an ancient city, so the legend goes, monks in a temple had to move a pile of 64 sacred disks from one location to another. The disks were fragile; only one could be carried at a time. A disk could not be placed on top of a smaller, less valuable disk. In addition, there was only one other location in the temple (besides the original and destination locations) sacred enough for a pile of disks to be placed there.

Using the intermediate location, the monks began to move disks back and forth from the original pile to the pile at the new location, always keeping the piles in order (largest on the bottom, smallest on the top). According to the legend, before the monks could make the final move to complete the new pile in the new location, the temple would turn to dust and the world would end.

A Smaller Tower of Hanoi

- Here is a smaller problem of just four disks

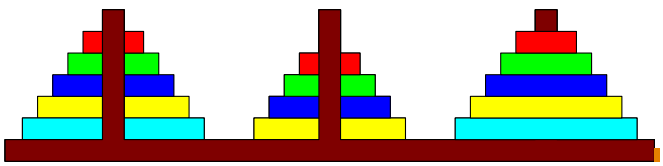


Algorithms in the Real World

- We require an algorithm to solve the towers of Hanoi
- Algorithms don't just apply to computers!
- If you try to solve the problem by hand you will discover that its quite fiddly
- There is a simple recursive solution which turns out to be optimal
- Let `move(X,Y)` denote the procedure of moving the top disk from peg `X` to peg `Y`
- Let `hanoi(n,X,Y,Z)` denote the procedure of moving the top `n` disks from peg `X` to peg `Z` using peg `Y`

Solving Towers of Hanoi

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```



Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n-1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

How Long Does It Take?

- How many moves does it take to transfer n disks?
- We can use the same procedure as before

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```

  - ⋆ $T(n) = 2\,T(n-1) + 1$
  - ⋆ $T(0) = 0$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- $T(1) = 2 \times 0 + 1 = 1$
- $T(2) = 2 \times 1 + 1 = 2 + 1 = 3$
- $T(3) = 2 \times 3 + 1 = 6 + 1 = 7$
- $T(4) = 2 \times 7 + 1 = 14 + 1 = 15$
- Looks like $T(n) = 2^n - 1$

Proof by Recursion

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case: Assume $T(n-1) = 2^{n-1} - 1$ then

$$T(n) = 2\,T(n-1) + 1$$
$$T(n) = 2 \times (2^{n-1} - 1) + 1$$
$$T(n) = 2^n - 2 + 1 = 2^n - 1$$ ✓

Time Complexity of Recursion

- The time complexity for recursion can be tricky to calculate
- The procedure is to calculate the time, $T(n)$, taken for a problem of size n in terms of the time taken for a smaller problem
- The difficulty is to solve the recursion
- Recursive programs can be very quick (e.g. $O(\log n)$ for computing integer powers)
- Recursive programs can also be very slow, (e.g. $O(2^n)$ for towers of Hanoi)
- In case your interested, if it takes 1 second to move a disk it will take almost $585\,000\,000\,000$ years to move 64 disks

Lessons

- Recursion is a powerful tool for writing algorithms
- It often provides simple algorithms to otherwise complex problems
- Recursion comes at a cost (extra function calls)
- There are times when you should avoid recursion (computing Fibonacci numbers)
- You need to be able to analyse the time complexity of recursion
- Used appropriately, recursion is fantastic!

Lesson 10: *Make Friends with Trees*

Binary trees, binary search trees, sets, tree iterators

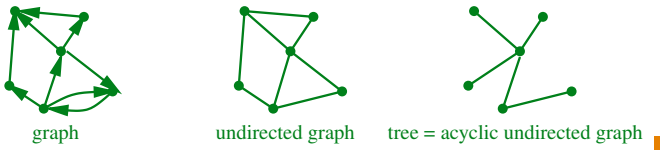
Outline

Trees

- Trees are one of the major ways of structuring data
- They are used in a vast number of data structures
  - Binary search trees
  - B-trees
  - splay trees
  - heaps
  - tries
  - suffix trees
- We shall cover most of these

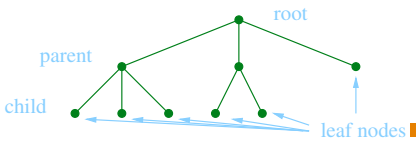
Defining Trees

- Mathematically a tree is an **acyclic undirected graph**
  - **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - **undirected**: the edges goes both ways
  - **acyclic**: there are no cycles in the graph



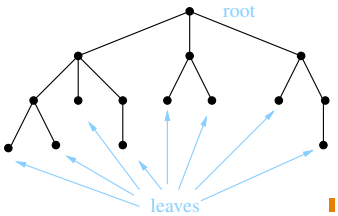
Borrowing from Nature

- We often impose an ordering on the nodes (or a direction on the edges)—known as a rooted tree
- Borrowing from nature, we recognise one node as the **root** node
- Nodes have **children** nodes living beneath them
- Each child has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes



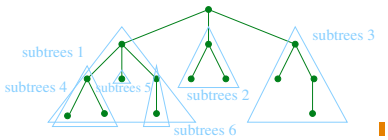
Spot the Error

- One small biological inconsistency
- Yep!, computer scientists draw there trees upside down
  - root at the top
  - leaves at the bottom



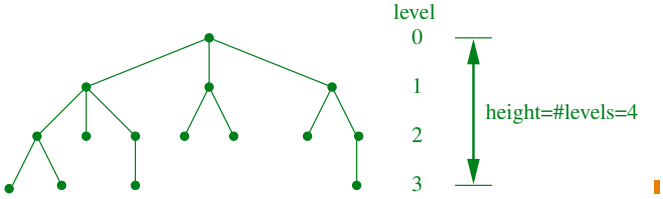
Subtrees

- We can think of the tree made up of **subtrees**



Level of Nodes

- It is useful to label different levels of the tree
- We take the **level** of a node in a tree as its distance from the root
- We take the **height** of a tree to be the number of levels



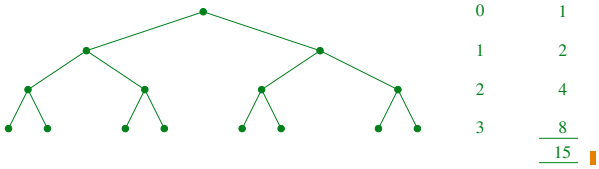
Outline

1. Trees
2. **Binary Trees**
   - Implementing Binary Trees
3. Binary Search Trees
   - Definition
   - Implementing a Set
4. Tree Iterators

Binary Trees

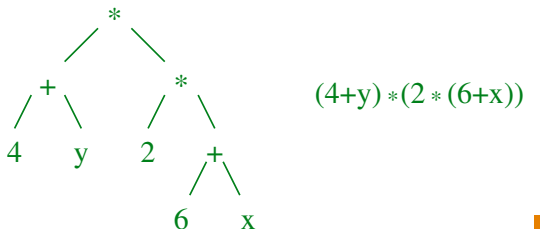
- A **binary tree** is a tree where each node can have zero, one or two children
- The total number of possible nodes at level l is 2^l
- The total number of possible nodes of a tree of height h is

$$1 + 2 + \cdots + 2^{h-1} = 2^h - 1$$



Uses of Binary Trees

- Binary trees have a huge number of applications
- For example, they are used as **expression trees** to represent formulae

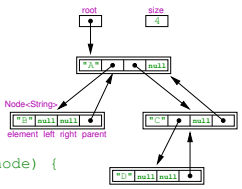


Implementation

- We wish to build a generic binary tree class with each node housing an element
- Again we use a `Node<T>` class as the building block for our data structure—in this case a node of the tree
- The `Node<T>` class will contain a pointer to left and right children
- To help navigate the tree each node will contain a pointer to its parent

C++ Code

```
template <typename T>
class binary_tree {
private:
  class Node {
  public:
    T element;
    Node* parent;
    Node* left = 0;
    Node* right = 0;

    Node(const T& value, Node* parent_node) {
      element = value;
      parent = parent_node;
    }
  };

  unsigned no_elements = 0;
  Node* root = 0;
```



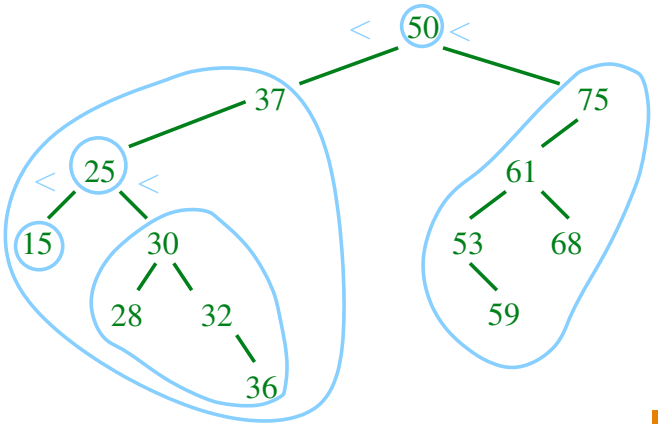
Outline

1. Trees
2. Binary Trees
   - Implementing Binary Trees
3. **Binary Search Trees**
   - Definition
   - Implementing a Set
4. Tree Iterators

Binary Search Trees

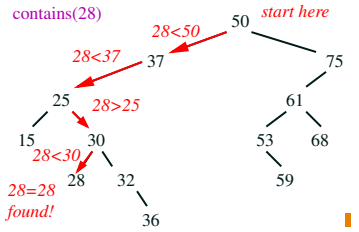
- We will concentrate on one of the most important binary trees, namely the **binary search tree**
- The binary search tree keeps the elements ordered
- We can define a binary search tree recursively
  1. Each element in the left subtree is less than the root element
  2. Each element in the right subtree is greater than the root element
  3. Both left and right subtrees are binary search trees

Example Binary Search Tree



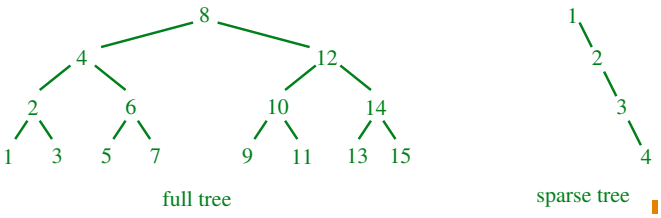
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found



Speed of Search

- The number of comparisons necessary to find an element in a binary tree depends on the level of the node in the tree
- The worst case number of comparisons is therefore the height of the tree
- This depends on the density of the tree



Implementing a Set

- A set is a fundamental **abstract data type**
- It is a collection of things with no repetition and no order
- Ironically because order doesn't matter we can order the elements

$$\{1,3,5,5,3,4\} = \{5,3,4,1\} = \{1,3,4,5\}$$

- This allows rapid search—a feature we care about
- Binary trees are one of the efficient ways of implementing a set

Fitting In

- The standard template library provides a class `std:set<T>`
- This contains many functions like
  - ⋆ Constructors
  - ⋆ `size()`
  - ⋆ `insert(T o)`
  - ⋆ `find(Object o)`
  - ⋆ `erase(Object o)`
  - ⋆ `begin()` and `end()`

Comparable

- To sort any objects they must be comparable
- In the STL the set implementation has a second template parameter: `std::set<T, Compare = less<T> >`
- by default this is defined to be `less<T>` (which is a function already defined for most common types) which you can define
- If you have a set of complex objects you will have to define `Compare`

```
bool MyCompare(MyObject left, MyObject right) {
  return something
}

mySet = set<MyObject, MyCompare>;
```


Find an Element

- One of the core operations of a binary tree is to find a node

```
iterator find(const T& element) {
 Node* current = root;
 while (current!=0) {
   if (current->element == element) {
     return iterator(current);
   }
   if (element < current->element) {
     current = current->left;
   } else {
     current = current->right;
   }
 }
 return iterator(0);
}
```

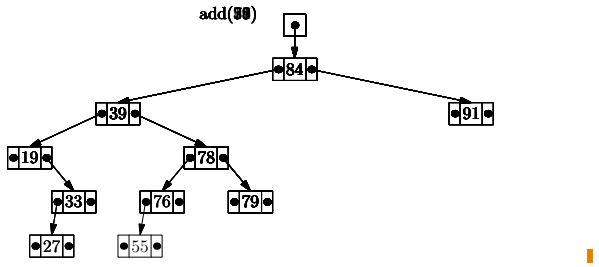

Add an Element

```
pair<iterator, bool> insert(const T& element) {
  if (no_elements==0) {
    root = new Node(element, 0);
    ++no_elements;
    return pair<iterator, bool>(iterator(root), true);
  }
  Node* parent = 0;
  Node* current = root;
  while(current != 0) {
    if (current->element == element) {
      return pair<iterator, bool>(iterator(0), false);
    }
    parent = current;
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
```



```
  current = new Node(element, parent);
  if (element < parent->element) {
    parent->left = current;
  } else {
    parent->right = current;
  }
  ++no_elements;
  return pair<iterator, bool>(iterator(current), true);
}
```

Tree in Action



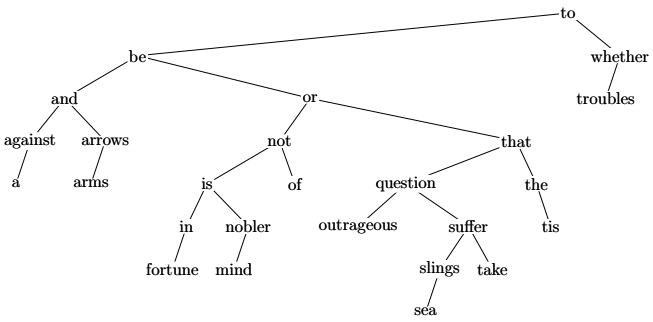
Shape of Tree

- The structure of the tree depends on the order in which we add elements to it
- Suppose we add

  To be, or not to be: that is the question:
  Whether 'tis nobler in the mind to suffer
  The slings and arrows of outrageous fortune,
  Or to take arms against a sea of troubles,

- Ignoring punctuation we get the following tree

Hamlet



Outline

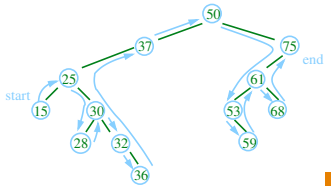
Tree Iterators

- As with most container classes it is very useful to define iterators
- `begin()` should return a "pointer" to the start of the tree
- `end()` provides a "pointer" past the end
- `operator*()` returns the element
- `opeator++()` increments the "pointer"
- `operator!=(lhs, rhs)` is used to compare iterators

```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```


Successor

- To find the successor we first start in the left most branch
- We follow two rules
  1. **If** right child exist **then** move right once and then move as far left as possible
  2. **else** go *up* to the left as far as possible and then move up right



$\{15, 25, 28, 30, 32, 36, 37, 50, 53, 59, 61, 68, 75\}$

Lessons

- Trees and particularly binary trees are one of the most important tools of a computer scientist
- Conceptually they are quite simple
- However, there are a lot of details that need to be understood
- Coding even simple trees needs great care
- As we will see things get more complicated

Lesson 11: *Keep Trees Balanced*

AVL trees, red-black trees, TreeSet, TreeMap

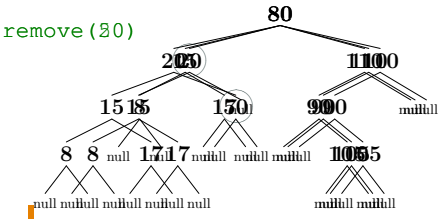
Outline

Recap

- Binary search trees are commonly used to store data because we need to only look down one branch to find any element
- We saw how to implement many methods of the binary search tree
  - `find`
  - `insert`
  - `successor` (in outline)
- One method we missed was `remove`

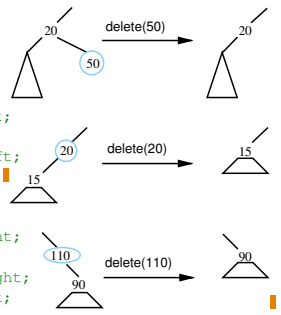
Deletion

- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)



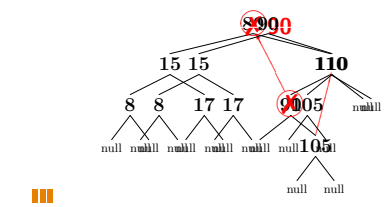
Code to remove Node n

```
if (n->left==0 && n->right==0) {
   if (n == n->parent->left)
      n->parent->left = 0;
   else
      n->parent->right = 0;
} else if (n->right==0) {
   if (n == n->parent->left)
      n->parent->left = n->left;
   else
      n->parent->right = n->left;
   n->left->parent = n->parent;
} else if (n->left==0) {
   if (n == n->parent->left)
      n->parent->left = n->right;
   else
      n->parent->right = n->right;
   n->right->parent = n->parent;
}
delete n;
```



Removing Element with Two Children

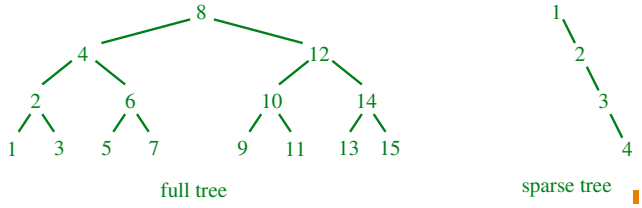
- If an element has two children then
  - replace that element by its successor
  - and then remove the successor using the above procedure



Outline

Why Balance Trees

- The number of comparisons to access an element depends on the depth of the node
- The average depth of the node depends on the shape of the tree



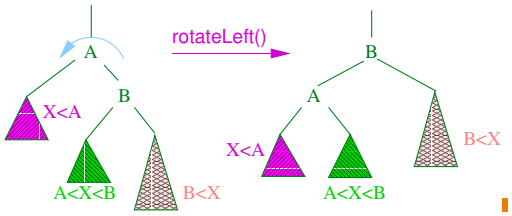
- The shape of the tree depends on the order the elements are added

Time Complexity

- In the best situation (a full tree) the number of elements in a tree is $n = \Theta(2^l)$ the depth is l so that the maximum depth is $\log_2(n)$
- It turns out for random sequences the average depth is $\Theta(\log(n))$
- In the worst case (when the tree is effectively a linked list), the average depth is $\Theta(n)$
- Unfortunately, the worst case happens when the elements are added *in order* (not a rare event)

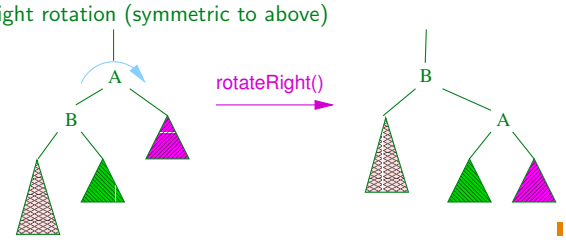
Rotations

- To avoid unbalanced trees we would like to modify the shape
- This is possible as the shape of the tree is not uniquely defined (e.g. we could make any node the root)
- We can change the shape of a tree using **rotations**
- E.g. left rotation



Types of Rotations

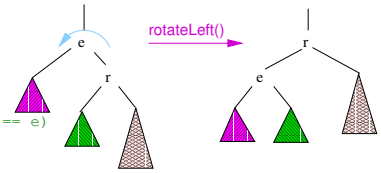
- We can get by with 4 types of rotations
  - ⋆ Left rotation (as above)
  - ⋆ Right rotation (symmetric to above)



  - ⋆ Left-right double rotation
  - ⋆ Right-left double rotation

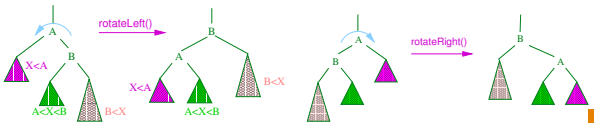
Coding Rotations

```
void rotateLeft(Node<T>* e)
{
  Node<T>* r = e->right;
  e->right = r->left;
  if (r->left != 0)
     r->left->parent = e;
  r->parent = e->parent;
  if (e->parent == 0)
     root = r;
  else if (e->parent->left == e)
     e->parent->left = r;
  else
     e->parent->right = r;
  r->left = e;
  e->parent = r;
}
```



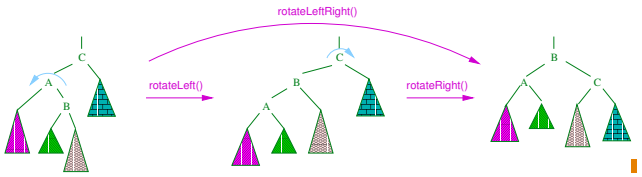
When Single Rotations Work

- Single rotations balance the tree when the unbalanced subtree is on the outside



Double Rotations

- If the unbalanced subtree is on the inside we need a double rotation




```
leftRotation(C.left);
rightRotation(C);
```


Outline

1. Deletion
2. Balancing Trees
   - Rotations
3. **AVL**
4. Red-Black Trees
   - TreeSet
   - TreeMap

Balancing Trees

- There are different strategies for using rotations for balancing trees
- The three most popular are
  - ⋆ AVL-trees
  - ⋆ Red-black trees
  - ⋆ Splay trees
- They differ in the criteria they use for doing rotations

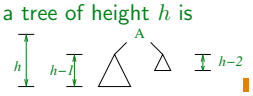
AVL Trees

- AVL-trees were invented in 1962 by two Russian mathematicians Adelson-Velski and Landis
- In AVL trees
  1. The heights of the left and right subtree differ by at most 1
  2. The left and right subtrees are AVL trees
- This guarantees that the worst case AVL tree has logarithmic depth

Minimum Number of Nodes

- Let $m(h)$ be the minimum number of nodes in a tree of height h
- This has to be made up of two subtrees: one of height $h-1$; and, in the worst case, one of height $h-2$
- Thus, the least number of nodes in a tree of height h is

$$m(h) = m(h-1) + m(h-2) + 1$$



- with $m(1) = 1$, $m(2) = 2$

Proof of Exponential Number of Nodes

- We have $m(h) = m(h-1) + m(h-2) + 1$ with $m(1) = 1$, $m(2) = 2$
- This gives us a sequence $1, 2, 4, 7, 12, \cdots$
- Compare this with Fibonacci $f(h) = f(h-1) + f(h-2)$, with $f(1) = f(2) = 1$
- This gives us a sequence $1, 1, 2, 3, 5, 8, 13, \cdots$
- It looks like $m(h) = f(h+2) - 1$
- Proof by substitution

Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$ with $m(1) = 1$, $m(2) = 2$
- We can prove by inductions, $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$, $m(2) = 2 \geq (3/2)^1 = 3/2$ ✓

$$m(h) \geq \left(\tfrac{3}{2}\right)^{h-3}\left(\tfrac{3}{2}+1+\left(\tfrac{3}{2}\right)^{3-h}\right) \geq \left(\tfrac{3}{2}\right)^{h-3}\tfrac{5}{2} = \left(\tfrac{3}{2}\right)^{h-3}\tfrac{10}{4} \geq \left(\tfrac{3}{2}\right)^{h-3}\tfrac{9}{4} = \left(\tfrac{3}{2}\right)^{h-1}$$ ✓

- Taking logs: $\log(m(h)) \geq (h-1)\log(3/2)$ or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O\left(\log(m(h))\right)$$

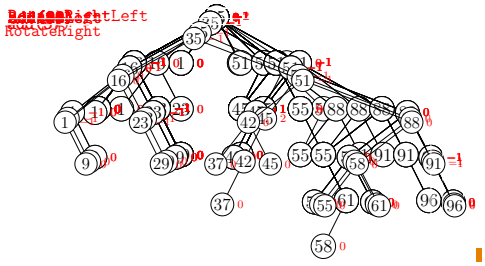
- The number of elements, n, we can store in an AVL tree is $n \geq m(h)$ thus

$$h \leq O(\log(n))$$

Implementing AVL Trees

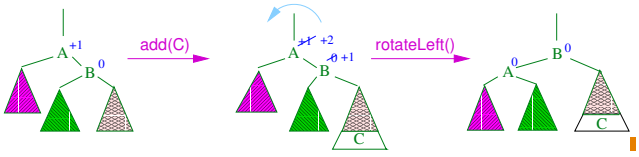
- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\texttt{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$



Balancing AVL Trees

- When `adding` an element to an AVL tree
  - ⋆ Find the location where it is to be inserted
  - ⋆ Iterate up through the parents re-adjusting the `balanceFactor`
  - ⋆ **If** the balance factor exceeds ± 1 `then` re-balance the tree and stop
  - ⋆ `else if` the balance factor goes to zero **then** stop



AVL Deletions

- AVL deletions are similar to AVL insertions
- One difference is that after performing a rotation the tree may still not satisfy the AVL criteria so higher levels need to be examined
- In the worst case $\Theta(\log(n))$ rotations may be necessary
- This may be relatively slow—but in many applications deletions are rare

AVL Tree Performance

- Insertion, deletion and search in AVL trees are, at worst, $\Theta(\log(n))$
- The height of an average AVL tree is $1.44\log_2(n)$
- The height of an average binary search tree is $2.1\log_2(n)$
- Despite being more compact insertion is slightly slower in AVL trees than binary search trees without balancing (for random input sequences)
- Search is, of course, quicker

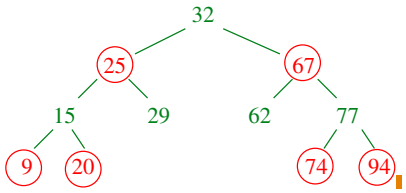
Outline

1. Deletion
2. Balancing Trees
   - Rotations
3. AVL
4. **Red-Black Trees**
   - TreeSet
   - TreeMap

Red-Black Trees

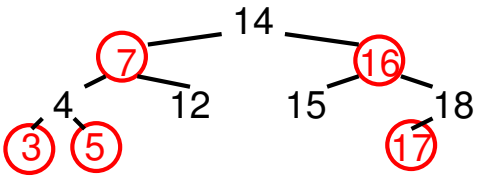
- Red-black trees are another strategy for balancing trees
- Nodes are either *red* or *black*
- Two rules are imposed

  Red Rule: the children of a red node must be black
  Black Rule: the number of black elements must be the same in all paths from the root to elements with no children or with one child



Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



Performance of Red-Black Trees

- Red-black trees are slightly more complicated to code than AVL trees
- Red-black trees tend to be slightly less compact than AVL trees
- However, insertion and deletion run slightly quicker
- Both Java Collection classes and C++ STL use red-black trees

Set

- The standard template library (STL) has a class `std::set<T>`
- It also has a `std::underordered_set<T>` class (which uses a hash table covered later)
- As well as `std::multiset<T>` that implements a multiset (i.e. a set, but with repetitions)
- Using sets you can also implement **maps**

Maps

- One major abstract data type (ADT) we have not encountered is the map class
- The map class `std:map<Key,V>` contain key-value pairs `pair<Key,V>`
  - ⋆ The first element of type `Key` is the **key**
  - ⋆ The second element of type `V` is the **value**
- Maps work as content addressable arrays

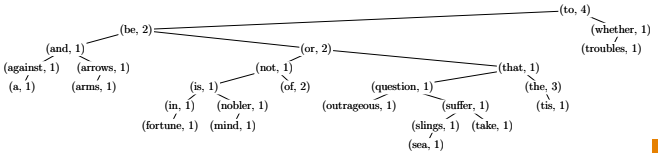
```
map<string, int> students;
student["John Smith"] = 89;
student["Terry Jones"] = 98;
cout << students["John Smith"];
```


Implementing a Map

- Maps can be implemented using a set by making each node hold a `pair<K,V>` objects

```
class pair<K,V>
{
  public:
  K first;
  V second;
}
```

- We can count words using the key for words and value to count

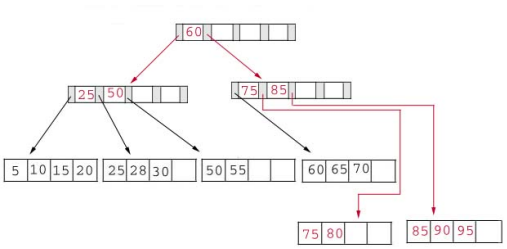


Lessons

- Binary search trees are very efficient (order $\log(n)$ insertion, deletion and search) provided they are balanced
- Balanced trees are achieved by performing rotations
- There are different strategies for deciding when to rotate including
  - ⋆ AVL trees
  - ⋆ Red-black trees
- Binary trees are used for implementing **sets** and **maps**

Further Mathematics and Algorithms

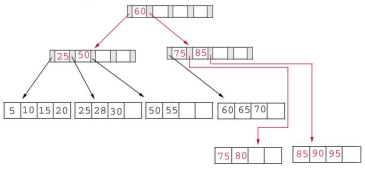
Lesson 12: *Sometimes It Pays Not to Be Binary*



B-Trees, Tries, Suffix Trees

Outline

1. B-Trees
2. Tries
3. Suffix Tree



B-Trees

- **B-trees** are balanced trees for fast search, finding successors and predecessors, insert, delete, maximum, minimum, etc.
- Not to be confused with binary trees
- They are designed to keep related data close to each other in (disk) memory to minimise retrieval time
- Important when working with large amount of data that is stored on secondary storage (e.g. disks)
- Used extensively in databases

When Big-O Doesn't Work

- An underlying assumption of Big-O is that all elementary operations take roughly the same amount of time
- This just isn't true of disk look-up
- The typical time of an elementary operation on a modern processor is 10^{-9} seconds
- But a typical hard disk might do 7 200 revolutions per minute or 120 revolutions per second
- The typical time it takes to locate a record is around 10ms or 10^7 times slower than an elementary operation

Accessing Data from Disk

- When accessing data from disk minimising the number of disk accesses is critical for good performance
- In database applications we want to store data as large sets
- Storing data in binary trees is disastrous as we typically need around $\log_2(n)$ disk accesses before we locate our data
- It is not unusual in databases for $n = 10\,000\,000$ so that $\log_2(n) \approx 24$
- Using binary trees it would often take several seconds to find a record

Multiway-Trees

- To remedy this we can use M-way trees so that the access time is

$$\log_M(n) = \frac{\log_2(n)}{\log_2(M)}$$

- In practice we might use $M \approx 200 \approx 2^8$ so we can reduce the depth of the tree by around a factor of 8
- The basic data structures for doing this is the B-tree
- There are many variants of B-tree, all trying to squeeze a bit more performances from the basic structure

B^+ Tree

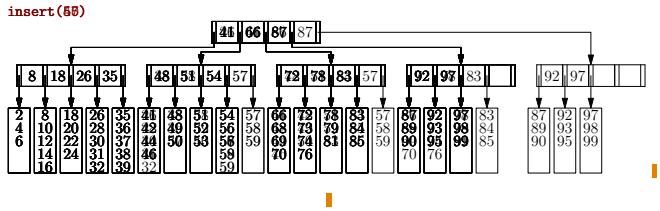
- A pretty basic implementation would obey the following rules
  1. The data items are stored at leaves
  2. The non-leaf nodes store up to M-1 keys to guide the search: key i represents the smallest key in subtree $i + 1$
  3. The root is either a leaf or has between 2 and M children
  4. All non-leaf nodes except the root have between $\lceil M/2 \rceil$ and M children
  5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data entries

Choosing M and L

- The choice of M and L depends on the block size (the information read in one go from disk)
- It also depends on the type of data that is being stored (integer, reals, strings, etc.)
- M and L might be in the hundreds or thousands
- In the examples below we consider tiny $M = L = 5$ which is unrealistic, but drawable

B-Tree Example

- $M = 5$, $L = 5$



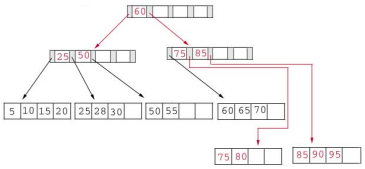
Other Changes

- If the root is full then it can be split into two and a new root created
- B-trees also have to allow the removal of records without losing its structure
- There are a number of variant strategies (e.g. neighbouring nodes can adopt a child if the current node cannot expand any more)
- The actual implementation of B-trees is tricky because there are many special cases

B-Tree Summary

- B-trees are an important data structure for databases where reducing the number of disk searches is vital
- They tend to be much more complex than the other data structures we have seen
- The problem of disk access can be improved by replacing disk memory with solid-state drives (still slow compared to memory)
- For massive databases new data structures have been developed to allow faster (although less flexible) information access (e.g. NOSQL, MongoDB, Neo4j)

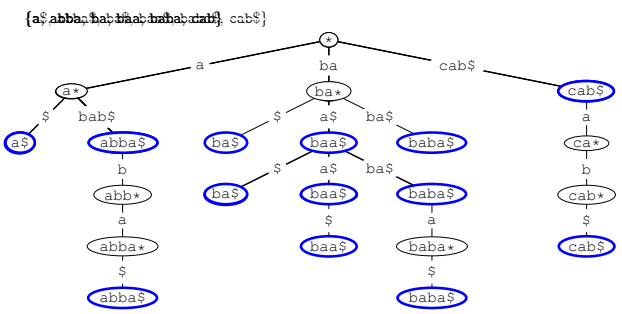
Outline



Tries

- A **Trie** (pron. 'try') or **digital tree** is a multiway tree often used for storing large sets of words
- They are trees with a possible branch for every letter of an alphabet
- Their names comes from the word *retrieval*
- Tries usually compactify the edges in the tree
- All words end with a special letter "$"

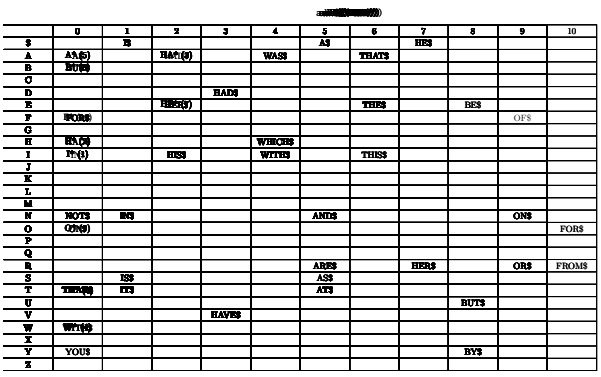
Trie



Uses of Tries

- Tries are yet another way of implementing sets
- They provide quick insertion, deletion and find
- Typically considerably quicker than binary trees and hash tables
- They are particularly good for spell checkers, completion algorithms, longest-prefix matching, hyphenation
- Each search finds the longest match between the words in the set and the query

Trie for 31 Most Common English Words



Disadvantage of Tries

- Table-based tries typically waste large amounts of memory
- Often table-based tries are used for the first few layers, while lower levels use a less memory intensive data structure
- These days memory is less of a problem so table-based tries are acceptable for some applications
- There are many implementations of tries each suited to a particular task

Binary Tries

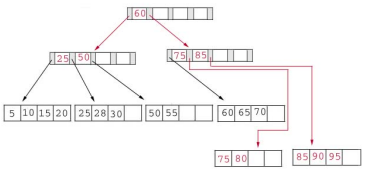
- One extreme (though not uncommon) solution to address memory issues is to build a bit-level trie so the data-structure is a binary tree
- It differs from a binary tree in that the decisions to go left or right depends on the current bit
- Although you lose the advantage of a multiway tree (of reducing the depth) it does find the longest match and it speeds up finds which fail

Why Tries?

- Tries are a classic example of a trade-off between memory and computational complexity
- Tries are slightly specialist and tend to get used in very particular applications
  - ⋆ Finding longest matches
  - ⋆ Completion, spell checking, etc.
- A basic trie is not too complicated, however, . . .
- There are many implementation which try to overcome the difficulty of wasting too much memory

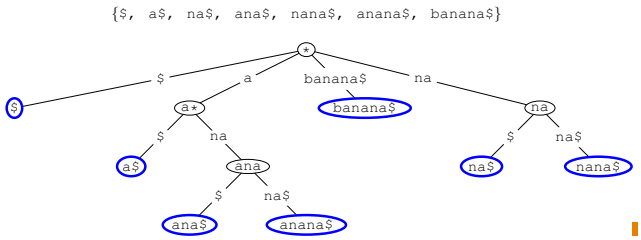
Outline

1. B-Trees
2. Tries
3. **Suffix Tree**



Suffix Tree

- Suffix tree is a trie of all suffixes of a string
- E.g. banana

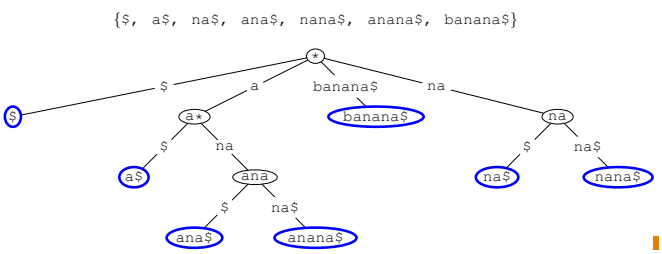


Importance of Suffix Tree

- The first linear-time algorithm for computing suffix trees was proposed by Peter Weiner in 1973, a more space efficient algorithm was proposed by Edward M. McCreight in 1976
- Esko Ukkonen in 1995 proposed a variant of McCreight's algorithm, but in a way that was much easier to understand
- It really only got implemented after this
- They are very important for string-based algorithms
- The classic application is in finding a match for a query string, Q, in a text, T

String Matching

- To find a match of a query string, Q, in a text, T, we can first construct the suffix tree of the string T we then simple look up the query, Q, using the trie



Complexity of Suffix Trees

- Using a regular trie for a suffix tree would typically use far too much memory to be useful
- However, by using pointers to the original text it is possible to build a suffix tree using $O(n)$ memory where n is the length of the text
- Furthermore (and rather incredibly) there is a linear time ($O(n)$) algorithm to construct the trie
- The algorithm is not however trivial to understand

Uses of Suffix Trees

- Suffix trees are efficient whenever it is likely that you will do multiple searches
- Exact word matching is in itself a very important application
- Suffix trees in combination with dynamic programming (which we will eventually get to) can be used to do inexact matching (finding the match with the smallest edit distance)
- Suffix trees get used in bioinformatics, advanced machine learning algorithms, . . .

Lessons

- Multiway trees can considerable speed up search over binary trees
- They are very important in some specialised applications (e.g. databases, spell-checking, completion, suffix trees)
- They are not as general purpose as binary search trees and are more complicated to implement
- But they can give the best performance—sometimes performance matters enough to make it worthwhile implementing multiway trees

Lesson 13: *Make a hash of it*

Hash tables, separate chaining, open addressing, linear/quadratic probing, double hashing

Outline

Content Addressable Memory

- Suppose we have a list of objects which we want to look up according to its contents
- This is often referred to as **associative memory** structures
- A classical example would be a telephone directory
  - We look up a name
  - We want to know the number
- What data structure should we use?

Lists and Trees

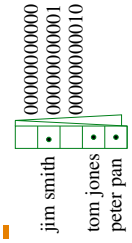
- To find an entry in a normal list takes $\Theta(n)$ operations
- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$
  - We will study binary search later
  - Maintaining an ordered list is costly ($\Theta(n)$ insertions)
- We could use a binary search tree
  - Search is $\Theta(\log(n))$
  - Insertion/deletion is $\Theta(\log(n))$

Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions
- Let us consider the content we want to search on as a **key**
- For telephone numbers the key would be the name of the person we want to phone
- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array
- That is the key is a string of, say, 100 characters so can be represented by an 800 digit binary number
- We could look up the key in a table of 2^{800} items

Hashing

- This approach is slightly wasteful of memory
- Almost all memory locations would be empty
- We can save on memory by folding up the table up onto itself



Hashing Codes

- A **hashing function** `hashCode(x)` takes an object, `x`, and returns a positive integer, the **hash code**
- To turn the hash code into an address take the modulus of the table size

```
int index = abs(hashCode(x) % tableSize);
```

- If `tableSize` $= 2^n$ we can compute this more efficiently using a mask

```
int index = abs(hashCode(x) & (tableSize -1));
```


Hashing Functions

- Hashing functions take an object and return an integer
- Hashing functions aren't magic
  - They tend to add up integers representing the parts of the object
- We want the integers to be close to random so that similar objects are mapped to different integers
- Sometimes two objects will be mapped to the same address—this is known as a **collision**
- Collision resolution is an important part of hashing

Hashing Strings

- A strings might be hashed using a function

```
unsigned long long hash(string const& s) {
  unsigned long long results = 12345;

  for (auto ch = s.begin(); ch != s.end(); ++ch) {
    results = 127*results + static_cast<unsigned char>(*ch);
  }
  return results;
}
```

- The numbers 12345 and 127 is to try to prevent clashes—there are lots of alternatives
- What we want is that strings that might be similar receive very different hash codes

DIY

- The `unordered_set<T, Hash<T> >` allows you to define your own hash function
- By default this is set to `std::hash<T>(T)`
- Not all classes have hash function defined so you will need to do this
- Care is needed to make you hash function produce near random hash codes

Outline

Collision Resolution

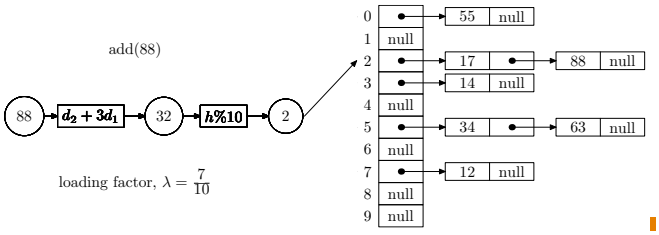
- Collisions are inevitable and must be dealt with
- There are two commonly used strategies
  - Separate chaining—make a hash table of lists
  - Open addressing—find a new position in the hash table
- Collisions add computational cost
- They occur when the hash table becomes full
- If the hash table becomes too full then it may need to be resized

Resizing a Hash Table

- Resizing a hash table is easy
  - Create a new hash table of, say, twice the size
  - Iterate through the old hash table adding each element to the new hash table
- Note that you have to recompute all the hash codes
- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance
- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

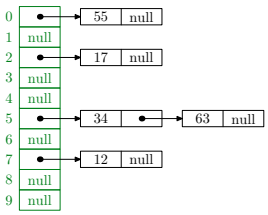


Search

- To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list
- The time complexity depends on where objects are hashed
- If the objects are evenly dispersed in the table, search (and insertion) is $\Omega(1)$
- If the objects are hashed to the same entry in the hash table then search is $O(n)$
- Provided you have a good hashing function and the hash table isn't too full you can expect $\Theta(1)$ average case performance

Iterating Over a Hash Table

- To iterate over a hash table we
  - Iterate through the array
  - At each element we iterate through the linked list
- The order of the elements appears random
- This becomes more efficient as the table becomes fuller



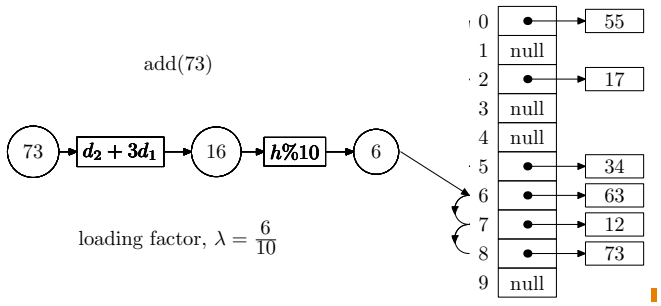
55, 17, 34, 63, 12

Outline

Open Addressing

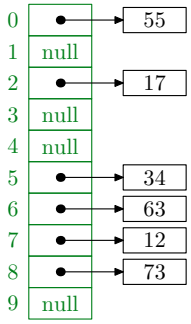
- In open addressing we have a single table of objects (without a linked-list)
- In the case of a collision a new location in the table is found
- The simplest mechanism is known as **linear probing** where we move the entry to the next available location

Linear Probing

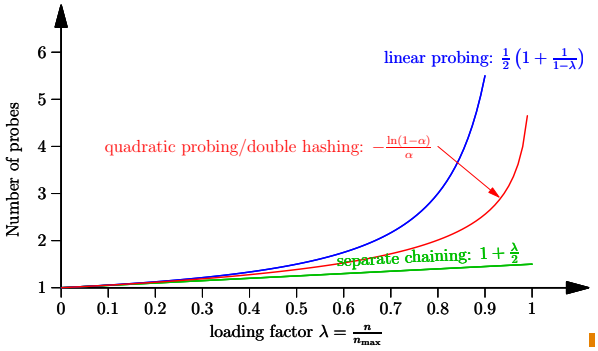


Linear Probing Pile Up

- The entries will tend to pile up or cluster—this is sometimes referred to as **primary clustering**
- Clusters become worse as the number of entries grow
- Clusters will increase the number of probes needed to find an insert location
- The proportion of full entries in the table is known as the **loading factor**



Reducing Number of Probes



- To avoid clustering we can use **quadratic probing** or **double hashing**

Quadratic Probing

- In quadratic probing we try the locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$
- That is we takes steps 1, 4, 9, 16, . . .
- Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full
- One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn't full
- However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full

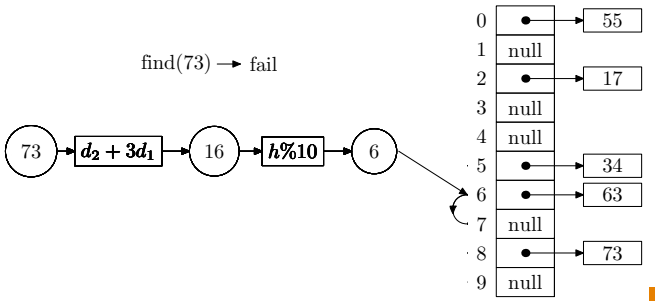
Double Hashing

- An alternative strategy is to known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$
- $h_2(x)$ is a second hash function that depends on the key
- A good choice is $h_2(x) = R - (x \mod R)$ where R is a prime smaller than the table size
- It is important that $h_2(x)$ is not a divisor of the table size
  - ⋆ Either make sure the table size is prime or
  - ⋆ Set the step size to 1 if $h_2(x)$ is a divisor of the table size

Problems with Remove

- For all open addressing hash systems removing an entry is a problem
- Remember our strategy to find an input x is
  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2
- If we remove an entry then find might reach an empty location which was previously full
- This can prevent us finding a true entry

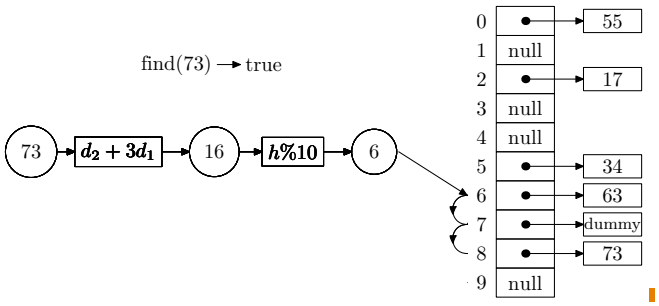
Linear Probing Example



Lazy Remove

- One easy fix is to mark the deleted table with a special entry
- A find method would consider this entry as full
- An iterator would ignore this entry
- An insert operator could insert a new entry in these special locations

Lazy Remove in Action



Outline

What Strategy to Use?

- Most libraries including the STL (and the Java Collection class) use separate chaining
- This has the advantage that its performance does not degrade badly as the number of entries increase
- This reduces the need to resize the hash table
- The C++ standard did not include a hash table until C++11 ☹—although very good hash tables existed in C++

Hash Sets and Maps

- C++ also provides an `unordered_map<Key, V>` class
- It's performance is asymptotically superior to `map`, $O(1)$ rather than $O(\log(n))$
- Hash functions can take time to compute (it is often $O(\log(n))$) so `unordered_sets` might not be faster than `sets`
- One major difference is that the iterator for `sets` return the elements in order, `undordered_set`'s iterator doesn't

Applications

- Hash tables are used everywhere
- E.g. most databases use hash tables to speed up search
- In many document applications hash tables will be being generated in the background
- Content addressability is ubiquitous to many application where hash tables are used as standard

Lessons

- Hash tables are one of the most useful tools you have available
- They aren't particularly difficult to understand, but you need to know about
  - ⋆ hashing functions
  - ⋆ collision strategies
  - ⋆ performance (i.e. when they work)

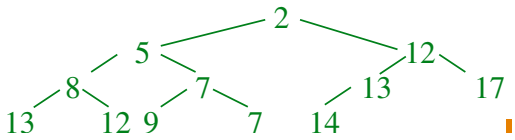
Lesson 14: *Use Heaps!*

Heaps, Priority queues, Heap Sort, Other heaps

Outline

Heaps

- A (min-)heap is (from one perspective) a binary tree
- It is a binary tree satisfying two constraints
  - ⋆ It is a **complete** tree
  - ⋆ Each child has a value 'greater than or equal to' its parent



- **complete** means that every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left

Outline

Priority Queues

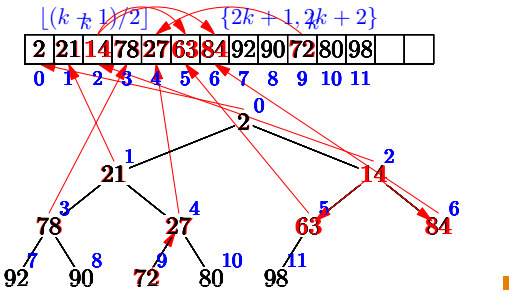
- One of the prime uses of heaps is to implement a priority queue
- A Priority Queue is a queue with priorities
- That is, we assign a priority to each element we add
- The head of the queue is the element with highest priority (smallest number)
- Used, for example, in simulating real time events
- Used to implement "greedy algorithms"

Priority Queue

- A simple Priority Queue might include
  - ⋆ `unsigned size()` returning the the number of elements
  - ⋆ `bool empty()` returns true if empty
  - ⋆ `void push(T element, int priority)` adds an element
  - ⋆ `T top()` returns head of queue
  - ⋆ `void pop()` dequeues head of queue

Array Implementation of Heaps

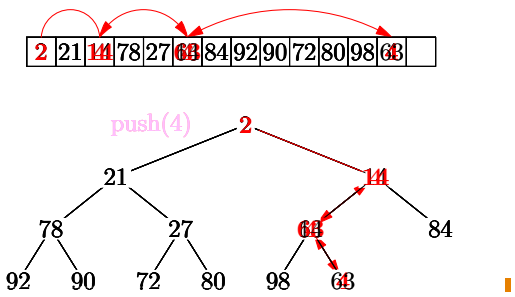
- Because the tree is complete we can implemented the heap efficiently using an array



Code for a Priority Queue

```
#include <vector>
using namespace std;

template <typename T, typename P>
class heapPQ {
private:
  vector<pair<T, P> > array;

public:

  heapPQ(unsigned capacity=11) {
    array.reserve(capacity);
  }

  unsigned size() {return array.size();}

  bool empty() {return array.empty();}

  const T& top() {return array[0].first;}
```

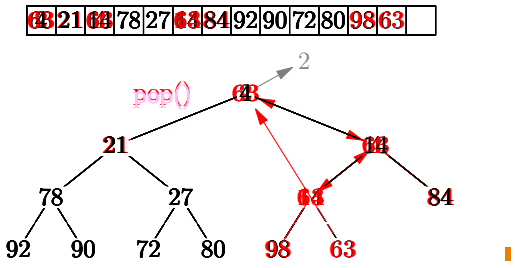
Adding an Element



Adding an Element

```
void push(T value, P priority) {
  pair<T,P> tmp(value, priority);
  array.push_back(tmp);
  unsigned child = size() - 1;

  /* Percolate Up */
  while(child!=0) {
    unsigned parent = (child-1)>>1;  // floor((child-1)/2)
    if (array[parent].second < array[child].second)
      return;
    array[child] = array[parent];
    array[parent] = tmp;
    child = parent;
  }
}
```

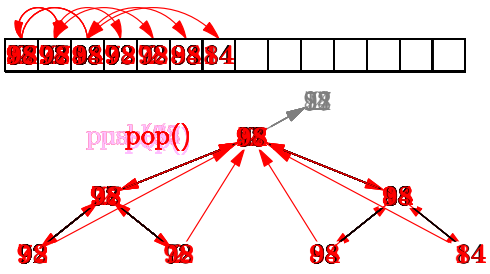

Popping the Top



Popping the Top

```
void pop() {
  unsigned parent = 0;
  pair<T, P> tmp = array.back();
  array[0] = tmp;
  array.pop_back();
  unsigned child = 1;

  /* Percolate down */
  while(child<size()) {
    if (child+1<=size() && array[child+1].second < array[child].second)
      ++child;
    if (array[child].second > array[parent].second)
      return;
    array[parent] = array[child];
    array[child] = tmp;
    parent = child;
    child = 2*parent + 1;
  }
}
```


Heaps in Action

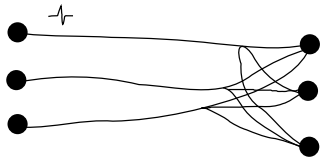


Time Complexity of Heaps

- The two important operation are `add` and `removeMin`
- These both either percolating an element up the tree or percolating an element down the tree
- The number of operations depends on the depth of the tree which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $O(\log(n))$
- Except `add` could also require resizing the array, but the amortised cost of this is low

Real Time Simulation

- A nice application of priority queues is to perform real time simulations
- I was once modelling a neural network where neuron fired an impulse which would then be received by other neurons



Synchronised Firing

- We wanted to show that if a group of neurons fired together they could make another group of neurons fire in synchrony despite the fact that it would take different times for the receiving neurons to feel the pulse (due to the different lengths of the axons)
- A famous Israeli group had "proved" this couldn't happen
- Using a priority queue we modelled the neurons
  - ⋆ When a neuron fired the receiving neurons would be put on a priority queue according to when they received the pulse
  - ⋆ If the receiving neurons received enough pulse in a short enough time they would then fire

Efficient Modelling

- Using a priority queue meant we knew when the next event would happen
- We did not have to run a clock where most of the time nothing happened
- This allowed us to perform a very large simulation efficiently
- The simulation showed that the pulse of neurons synchronised despite the "proof" that this wouldn't happen

Outline

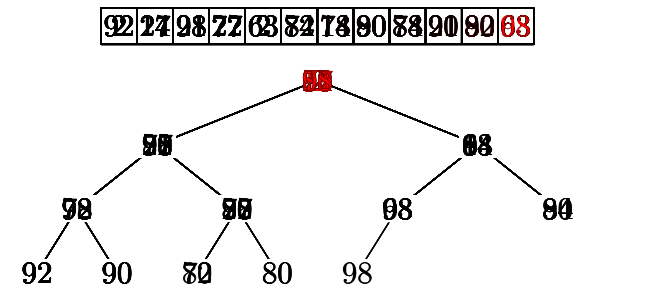
Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

```
template <typename T>
void sort(vector<T> aList)
{
    HeapPQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
      aHeap.push(element, element);

    aList.clear();
    while(aHeap.size() > 0) {
      aList.push_back(aHeap.top());
      aHeap.pop();
    }
}
```

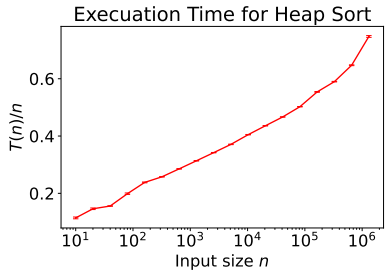
- Note that this is not an in-place sort algorithm (i.e. it uses lots of memory)

Example of Heap Sort



Complexity of Heap Sort

- As we have to add n elements and then remove n elements the time complexity is log-linear, i.e. $O(n \log(n))$



- This is actually a very efficient algorithm

Outline

Other Heaps

- Binary Heaps are so useful that other types of heaps have been developed
- The simplest enhancement is to combine a binary heap with a map which maintains a pointer to each element
- The map has to be updated every time elements are moved in the heap (fortunately only $O(\log(n))$ elements are move each time the heap is updated)
- The advantage of this heap is that the priorities of elements can be changed (involving percolating elements up or down the tree)

Merging Heaps

- One common demand on a heaps is to merge two heaps
- Unfortunately binary heaps are not efficiently merged
- There are a variety of different heaps (leftist heaps, skew heaps, binomial queues, . . .) designed to be merged
- All these heaps are real binary trees (i.e. represented by pointers rather than put in an array)
- They are slower than a binary heap because indexing is slightly slower as is creating node objects
- However, they allow merging

Other Operations

- All other operations are achieved by merging
  - Adding an element is achieved by merging the current heap with a heap of one element
  - Removing the minimum element is achieved by removing the root and merging the left and right tree
- For details see the course text (you won't be examined on the details of these heaps)

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - They are used when you need a queue with priorities, e.g. in operating systems
  - They can be used to perform pretty efficient sort
  - They are often used for implementing greedy type algorithms
  - One important application is in real time simulations
- There exists many extensions of heaps

Lesson 15: *Analyse!*

Pseudo code, binary search, insertion sort, selection sort, lower bound complexity

Outline

Algorithm Analysis

- We've covered most of the basic data structures
- The rest of the course is going to focus more on algorithms
- We will look predominantly at
  - Searching
  - Sorting
  - Graph Algorithms
- Emphasise general solution strategies

Code and Pseudo Code

- C++ code is often difficult to read—there are often programming details we don't care about
- It contains details such as throwing exception which are repetitive and often depends on who you are writing the code for
- Algorithms are not language dependent (data structures are a bit more language dependent)
- To focus on what is important we will use a stylised programming language called **pseudo code**

Pseudo Code

- There is no standard for pseudo code
- The commands are not too dissimilar to C++
- The one strange convention is that assignments use an arrow ←
- Arrays are written in bold $\boldsymbol{a}$ with elements a_i
- In pseudo-code you are free to invent any operations that can be easily interpreted

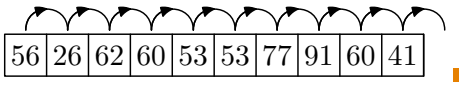
Outline

Dumb Search

```
DumbSearch(a, x)
{
  /* search array a = (a_1, ... a_n)  */
  /* for x return true */
  /* if successful else false */
  for i←1 to n
    if (a_i = x)
      return true
    endif
  endfor

  return false
}
```

```
bool search(T a[], T x)
{
  for (int i=0; i<n; i++) {
    if (a[i] == x)
      return true;
  }

  return false;
}
```

find(12) ⟶ false



Time Complexity

- Worst case:
  - The worst case for a successful search is when the element is in the last location in the array
  - This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - The best case is when the element is in the first location
  - This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - Assume every location is equally likely to hold the key

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

- For an unsuccessful search n comparison are necessary

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

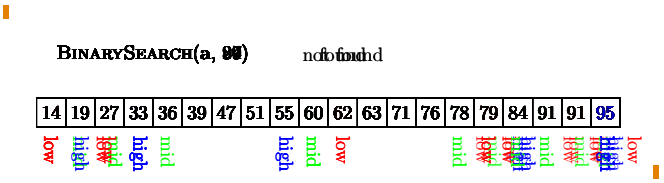
```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid −1
    else
      return true
    endif
  endwhile
  return false
}
```

- ⋆ Based on a **divide-and-conquer** strategy
- ⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

- ⋆ Based on a recursive idea

Binary Search in Action



Analysis

- We count the number of comparisons (counting each `if/else if` statement as a single comparison)
- Let $C(n)$ be the number of comparisons needed to search in an array of size n
- After one comparison we are left (in the worst case) with having to search an array not larger than $\lfloor n/2 \rfloor$, thus

$$C(n) < C(\lfloor n/2 \rfloor) + 1$$

- We've seen this relation before (lesson on Recursion)
- Easy to show $C(n) < \lfloor \log_2(n) \rfloor + 1 = O(\log(n))$

Outline

1. Algorithm Analysis
2. Search
3. **Simple Sort**
   - Insertion Sort
   - Selection Sort
4. Lower Bound

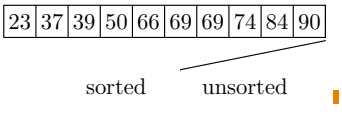
Sort Characteristics

- Sort is one of the best studied algorithms. We care about stability, space and time complexity
- A sort algorithm is said to be **stable** if it does not change the order of elements that have the same value
- Space Complexity. Sort is said to be
  - ⋆ **In-place** if the memory used is $O(1)$
- Time Complexity. In particular we are interested in
  - ⋆ Worst case
  - ⋆ Average case
  - ⋆ Best case

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n} (i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

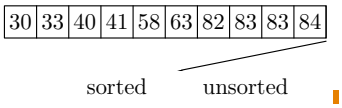
$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons
- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
 for i←1 to n-1
  min ←i
  for j←i+1 to n
   if a_j<a_min
    min←j
   end if
  end for
  swap a_i and a_min
 end for
}
```

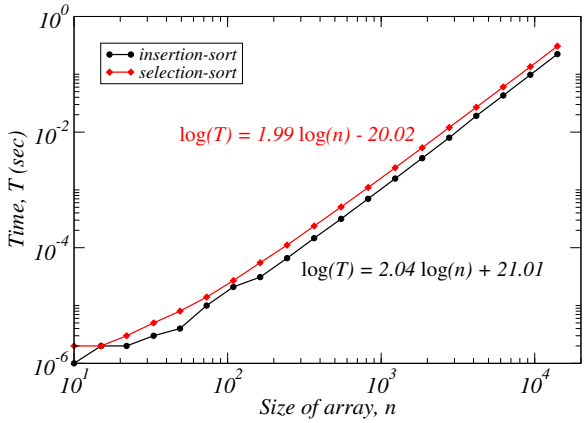


Analysis of Selection Sort

- Selection sort is in-place
- It isn't stable

- Selection sort always requires $n(n-1)/2$ comparisons so has the same worst case, but worse average case and best case complexity as insertion sort
- It only performs $n-1$ swaps—this makes it attractive (insertion sort moved more elements)

Insertion versus Selection Sort



Bubble Sort

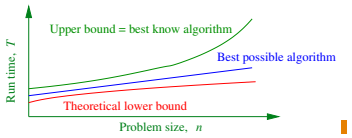
- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Outline

1. Algorithm Analysis
2. Search
3. Simple Sort
   - Insertion Sort
   - Selection Sort
4. **Lower Bound**

How Well Can You Do?

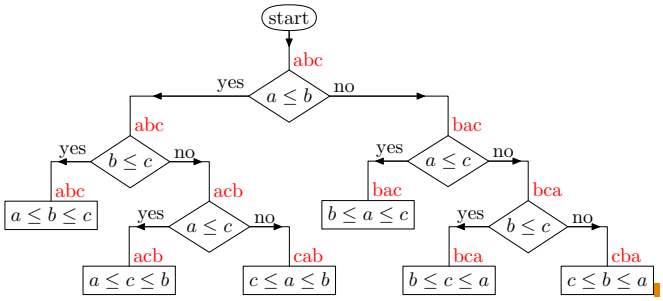
- Given a problem we would like to know what is the time complexity of the best possible program
- Usually there is no way of knowing this
- We can get an upper bound—if we know the time complexity of any algorithm that solves the problem we have an upper bound
- Lower bounds are far trickier
- A lower bound of $f(n)$ is a guarantee that we spend at least $f(n)$ operations to solve the problem



Decision Trees

- Decision trees are a way to visualise (at least, in principle) many algorithms
- They will eventually give us a lower bound on the time complexity of sort using binary decisions
- A decision tree shows the series of decisions made during an algorithm
- For sort based on binary comparisons the decision tree shows what the algorithm does after every comparison

Decision Tree for Insertion Sort



- Note there is one leaf for every possible way of sorting the list

Decision Trees and Time Complexity

- The time taken to complete the task is the depth of the tree at which we finish (i.e. the leaf nodes)
- We can thus read of the time complexity
  - ⋆ worst case time: depth of the deepest of leaf
  - ⋆ best case time: depth of the shallowest of leaf
  - ⋆ average case time: average depth of leaves
- Different sort strategies will have different decision trees
- Decision trees are usually far too large to write out ☹

Requirements of Correct Sort

- Any sort based on binary comparisons must have a leaf of the tree for every possible way of sorting the list
- The array $[a, b, c]$ must be arranged differently for all combinations

$$[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]$$

- That is they must go through a different path of the decision tree
- If not sort won't work

Minimum Number of Leaves

- There must be, at least, one leaf node of the decision tree for each possible permutation of the list
- How many permutations are there of a list of size n?
- Start with a sequence $(a_1, a_2, \ldots, a_n)$
- To create a new permutation we can choose any member of the list as the first element
- We can choose any of the remaining $n-1$ elements of the list as the second element of the permutation
- The total number of permutation is $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = n!$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

How Big is $\log_2(n!)$

- We showed in the second lecture that

$$\left(\frac{n}{2}\right)^{n/2} < n! < n^n$$

- It is not too difficult to show that asymptotically (i.e. as $n \to \infty$) that $n!$ approaches $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$—this is known as **Stirling's approximation**
- Thus

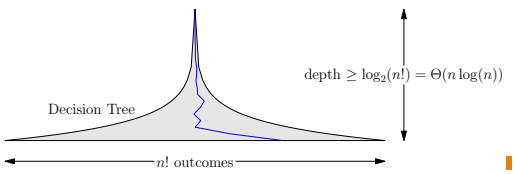
$$\begin{aligned}\log_2(n!) &\approx n\log_2(n) - n\log_2(e) + \frac{\log_2(n)}{2} + \frac{\log_2(2\pi)}{2} \\ &= \Theta(n\log_2(n))\end{aligned}$$

Complexity of Sorting

- We therefore have a lower bound on the time complexity of $\Omega(n\log(n))$
- This is true for any sort using binary comparisons
- We will see in the next lecture there exists algorithms with time complexity $O(n\log(n))$
- This means our lower bound is tight—i.e. it is the true cost of the best algorithm
- Having a lower bound we know we are not going to obtain a substantially faster algorithm

Lessons

- Analysis of algorithms is hard
- Analysis is important: without it we don't know if we have a good algorithm or whether we should try to find a more efficient one
- Lower bounds are particularly important



Lesson 16: *Sort Wisely*

Merge sort, quick sort and radix sort

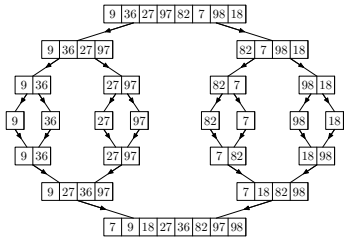
Outline

Merge Sort

- Merge sort is an example of sort performed in log-linear (i.e. $O(n\log(n))$) time complexity
- It was invented in 1945 by John von Neumann
- It is an example of a divide-and-conquer strategy
  - ⋆ That is, the problem is divided into a number of parts recursively
  - ⋆ The full solution is obtained by recombining the parts

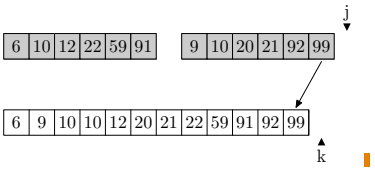
Algorithm

```
MERGESORT(a)
{
 if n > 1
   copy a[1 : ⌊n/2⌋] to b
   copy a[⌊n/2⌋ + 1 : n] to c
   MERGESORT(b)
   MERGESORT(c)
   MERGE(b, c, a)
 endif
}
```



Merge

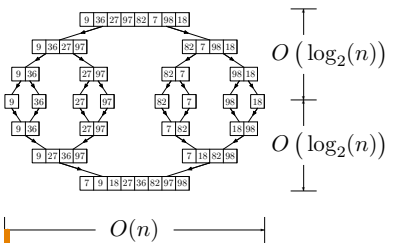
```
MERGE(b[1 : p], c[1 : q], a[1 : p + q])
{
 i←1
 j←1
 k←1
 while i ≤ p and j ≤ q do
  if b_i ≤ c_j
    a_k ←b_i
    i ←i+1
  else
    a_k ←c_j
    j ←j+1
  endif
  k ←k+1
 end
 if i=p
  copy c[j : q] to a[k : p + q]
 else
  copy c[i : q] to a[k : p + q]
}
```



Properties of Merge Sort

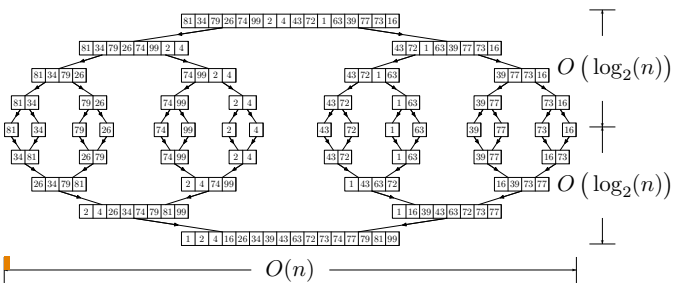
- Merge sort is stable provided we merge carefully (i.e. it preserves the order of two entries with the same value)
- Merge sort isn't in-place—we need an array of at most size n to do the merging
- Merging is quick. Given two arrays of size n the most number of comparisons we need to perform is $n-1$

Time Complexity of Merge Sort



$n = 8$

Time Complexity of Merge Sort



$n = 16$

Time Complexity

- We again measure the complexity in the number of comparisons
- From the above argument $C(n) = O(n \times \log_2(n))$
- We can be a bit more formal

$$C(n) = 2C(\lfloor n/2 \rfloor) + C_{\text{merge}}(n) \qquad \text{for } n > 1$$
$$C(0) = 1$$

- But in the worst case $C_{\text{merge}}(n) = n - 1$
- Leads to $C_{\text{worst}}(n) = n \log_2(n) - n + 1$

General Time Complexity

- In general if we have a recursion formula

$$T(n) = aT(n/b) + f(n)$$

with $a \geq 1$, $b > 1$

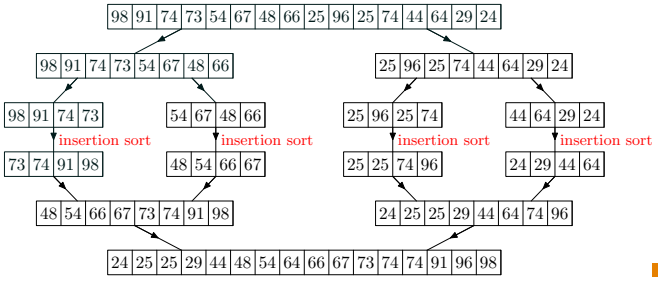
- If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{if } a < b^d \\ \Theta\left(n^d \log(n)\right) & \text{if } a = b^d \\ \Theta\left(n^{\log_d(a)}\right) & \text{if } a > b^d \end{cases}$$

- Analogous results hold for the family O and Ω

Mixing Sort

- For very short sequences it is faster to use insertion sort than to pay the overhead of function calls



Outline

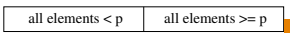
1. Merge Sort
2. **Quick Sort**
3. Radix Sort

Quicksort

- The most commonly used fast sorting algorithm is **quicksort**
- It was invented by the British computer scientist by C. A. R. Hoare in 1962
- It again uses the divide-and-conquer strategy
- It can be performed in-place, but it is **not** stable
- It works by splitting an array into two depending on whether the elements are less than or greater than a **pivot** value
- This is done recursively until the full array is sorted

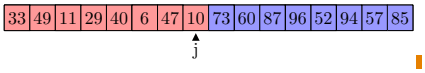
Partition

- We need to partition the array around the pivot p such that



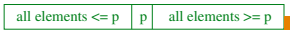

```
PARTITION(a, p, left, right)
{
 i ←left
 j ←right
 repeat {
  while a_i < p
    i++
  while a_j ≥ p
    j--
  if i ≥ j
    break
  SWAP(a_i, a_j)
 }
}
```

pivot = 52



Optimising Partitioning

- There are different ways of performing the partitioning
- We want to minimise the time taken on the inner loop
- This means we want to perform as few checks as possible
- One method of doing this is to place *sentinels* at the ends of the array
- We can also reduce work by placing the partition in its correct position



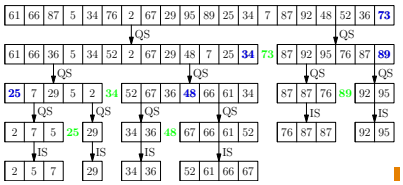
Choosing the Pivot

- There are different strategies to choosing the pivot
- Choose the first element in the array
- Choose the median of the first, middle and last element of the array
- This increases the likelihood of the pivot being close to the median of the whole array
- For large arrays (above 40) the median of 3 medians is often used

Quicksort

We recursively partition the array until each partition is small enough to sort using insertion sort

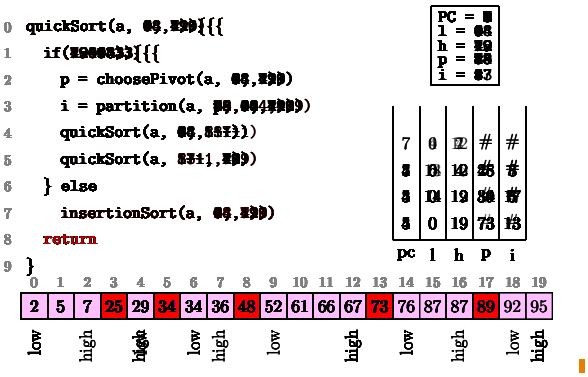
```
QuickSort(a, left, right) {
  if (right-left < threshold)
    InsertionSort(a, left, right)
  else
    pivot = ChoosePivot(a, left, right)
    part = Partition(a, pivot, left, right)
    QuickSort(a, left, part-1)
    QuickSort(a, part+1, right)
  endif
}
```



Time Complexity

- Partitioning an array of size n takes $\Theta(n)$ operations
- If we split the array in half then number of partitions we need to do is $\lceil \log_2(n) \rceil$
- This is the best case thus quicksort is $\Omega\left(n \log(n)\right)$
- If the pivot is the minimum element of the array then we have to partition $n-1$ times
- This is the worst case so quicksort is $O\left(n^2\right)$
- This worst case will happen if the array is already sorted and we choose the pivot to be the first element in the array!

QuickSort



Sort in Practice

- The STL in C++ offers three sorts
  - ⋆ `sort()` implemented using quicksort
  - ⋆ `stable_sort()` implemented using mergesort
  - ⋆ `partial_sort()` implemented using heapsort
- Java uses
  - ⋆ Quicksort to sort arrays of primitive types
  - ⋆ Mergesort to sort Collections of objects
- Quicksort is typically fastest but has worst case quadratic time complexity

Selection

- A related problem to sorting is selection
- That is we want to select the k^{th} largest element
- We could do this by first sorting the array
- A full sort is not however necessary—we can use a modified quicksort where we only continue to sort the part of the array we are interested in
- This leads to a $\Theta(n \log(n))$ algorithm which is considerably faster then sorting

Outline

1. Merge Sort
2. Quick Sort
3. **Radix Sort**

Radix Sort

- Can we get a sort algorithm to run faster than $O\left(n \log(n)\right)$?
- Our proof that this was optimal assumed we were performing binary decisions (is a_i less than a_j?)
- If we don't perform pairwise comparisons then the proof doesn't apply
- Radix sort is the classic example of a sort algorithm that doesn't use pairwise comparisons

Sorting Into Buckets

- The idea behind radix sort is to sort the elements of an array into some number of buckets
- This is done successively until the whole array is sorted
- Consider sorting integers in decimals (base 10 or radix 10)
- We can successively sort on the digits
- The sort finishes when we have got through all the digits

Radix Sort in Action

11	0	null
13	1	null
26	2	null
29	3	null
37	4	null
43	5	null
51	6	null
51	7	null
52	8	null
79	9	null

Time Complexity of Radix Sort

- We need not use base 10 we could use base r (the radix)
- If the maximum number to be sorted is N then the number of iterations of radix sort is $\log_r(N)$
- Each sort involves n operations
- Thus the total number of operations is $O\left(n\lceil\log_r(N)\rceil\right)$
- Since N does not depend on n we can write this as $O\left(n\right)$

Bucket Sort

- A closely related sort is bucket sort where we divide up the inputs into buckets based on the most significant figure
- We then sort the buckets on less significant figures
- Quicksort is a bucket sort with two buckets, but where we choose a pivot to determine which bucket to use

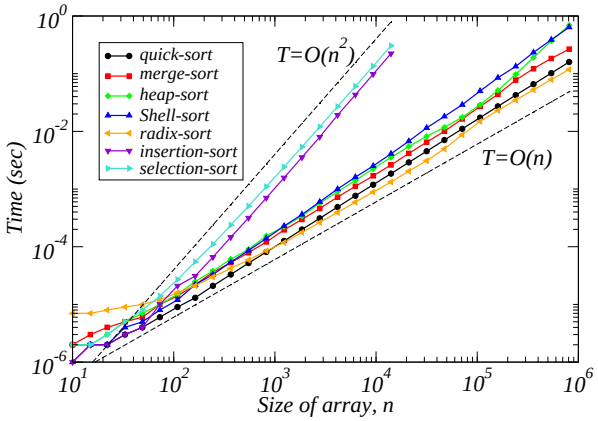
Minimum Time for Sort

- Can we do better?
- In any sort we need to examine all possible elements in the array
- If there is an element that isn't examined then we don't know where to put it
- Thus the lower bound on any sort algorithm is $\Omega(n)$

Practical Sort

- In practice, radix sort or bucket sort are rarely used
- The overhead of maintaining the buckets make them less efficient than they might appear
- Radix sort is harder to generalise to other data types than comparison based sorts
- In practice quick sort and merge sort are usually preferred
- Having said that there are some very neat implementations of radix sort

Comparison of Sort Algorithms

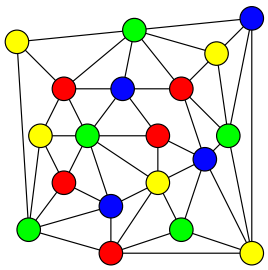


Lessons

- Sort is important—it is one of the commonest high level operations
- Merge sort and quick sort are the most commonly used sort
- There are sorts that have a better time complexity that quicksort
- In practice it is difficult to beat quicksort

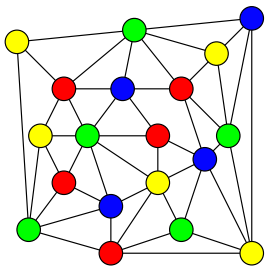
Further Mathematics and Algorithms

Lesson 17: *Think Graphically*



Graph theory, applications of graphs, graph problems

Outline

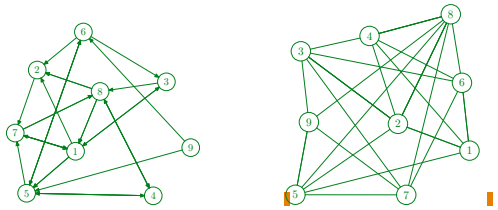


Motivation

- Many different problems can be described in terms of graphs
- This often reveals the true nature of the problem
- It unifies many apparently different problems
- As much is known about graph problems it often provides a pointer to the solution

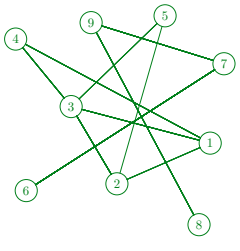
Definition of a Graph

- A graph, G, can be described by
  - A set of vertices or nodes $\mathcal{V} = \{1, 2, 3 \ldots n\}$
  - A set of edges $\mathcal{E} = \{(i, j) | \text{vertex } i \text{ is connected to vertex } j\}$
- The edges may be
  - **directed**—sometimes called a **digraph**
  - **undirected**



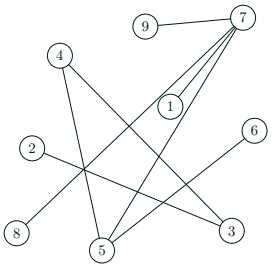
Connected and Unconnected Graphs

- A graph is **connected** if you can get from one node to any other along a series of edges
- Otherwise it is **disconnected**



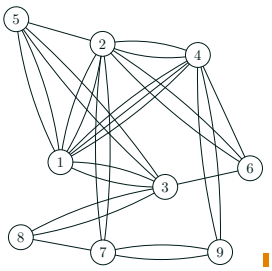
Trees

- A tree is a connected graphs with no cycles
- A tree will have $n - 1$ edges



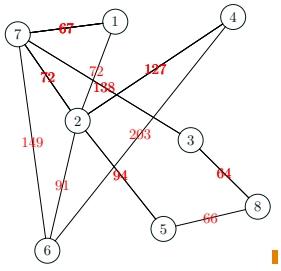
Multigraphs

- If the collection of edges is a *multiset* then we obtain a **multigraphs** where more than one edge is allowed between pairs of vertices



Weighted Graphs

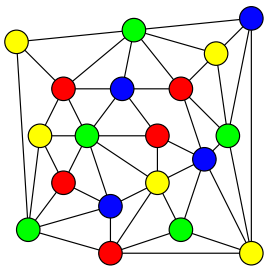
- If we assign a number to an edge we obtain a **weighted graph**



Networks

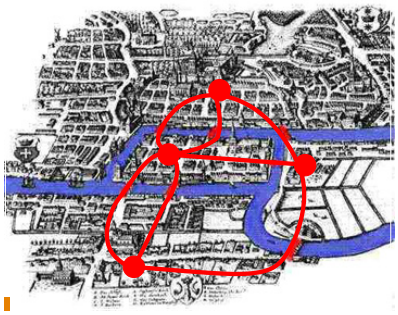
- Sometimes we add more information to the graph
- E.g. attributes to the nodes or edges
- Graphs with many attributes are often referred to as **networks**

Outline



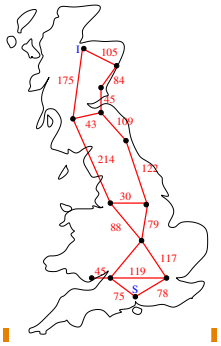
Bridges of Königsberg

Is there a tour around Königsberg going over every bridge once?



In 1736 Euler published a paper answering this question and founding graph theory

Representing Distances



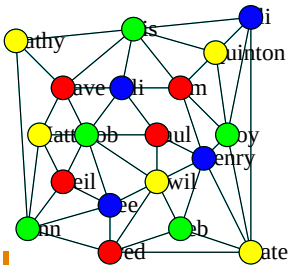
- Consider some graph
- With weights representing the distance between nodes
- What is the shortest distance between S and I?

Other Applications

- We could take the weights to represent the time taken to travel between nodes
- In a computer network the weights might represent the bandwidth
- In a representation of a transport system the weights might represent the carrying capacity of the traffic on a road
- Graphs can be used to represent other kinds of relationships
- E.g. We could create a digraph of links between web pages

Christmas Card Problem

- I have four types of Christmas cards
- Some of my friends know each other

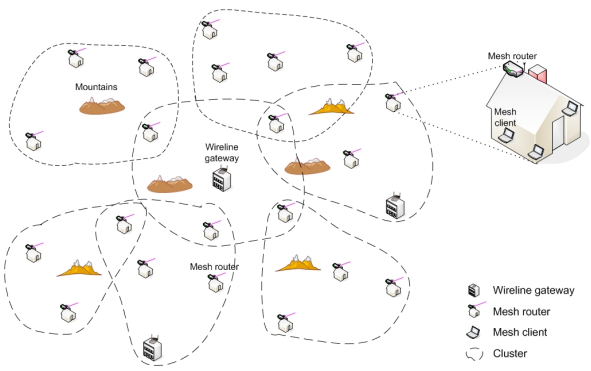


- I don't want to send friends that know each other the same card

A Real World Problem

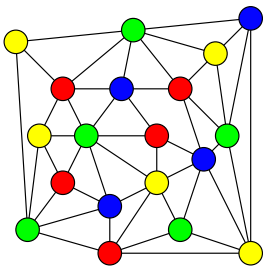
- A food company used different colour bags for each of it products
- To save money they reduced the stock of bags to 25
- They wanted to know what items to put in what bags so that as few customers as possible would have items with the same colour bags
- This can again be reduced to a graph colouring problem
  - Each node represents an item
  - The edges were weighted by the number of customers that took both items
  - The aim was to colour the nodes with 25 colours to minimise the weights where the edges shared the same colour

Frequency Assignment Problem



Outline

1. Graph Theory
2. Applications of Graphs
   - Geometric applications
   - Relational applications
3. **Implementing Graphs**
4. Graph Problems



Representations

- There is no single way to represent graphs
- The best representation depends on the graph
- Some books describe a $Graph\ ADT$—graphs are too varied for this to be very useful
- An important issue in representing a graph is how to store the edge information

Adjacency Matrices

- One representation of a graph $G = (\mathcal{V}, \mathcal{E})$ is in term of an $n \times n$ **adjacency matrix A** with elements

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in \mathcal{E} \\ 0 & \text{if } (i,j) \notin \mathcal{E} \end{cases}$$

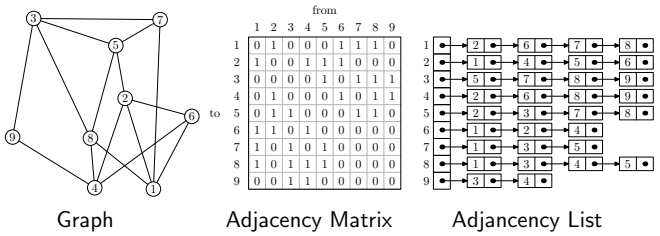
  where $n = |\mathcal{V}|$
- For undirected graphs **A** is a symmetric matrix, i.e. $\mathbf{A} = \mathbf{A}^{\mathsf{T}}$
- For weighted graphs we often store the **connectivity matrix** or **cost-adjacency matrix**, **C**, where

$$C_{ij} = \begin{cases} w_{ij} & \text{if } (i,j) \in \mathcal{E} \\ 0 & \text{if } (i,j) \notin \mathcal{E} \end{cases}$$

Adjacency Lists

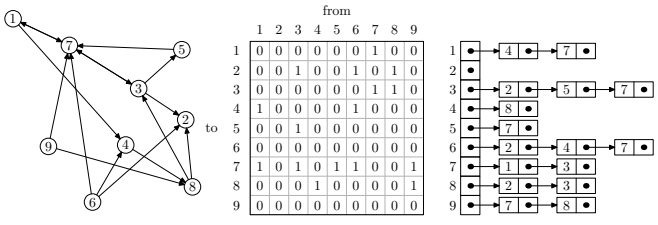
- For **dense** graphs where the number of edges is $\Theta(n^2)$ the adjacency matrix is often a useful representation
- But in **sparse** graphs where the number of edges is $\Theta(n)$ the adjacency matrix has a very large number of zeros
- A more efficient representation is in terms of the adjacency list where the set of outgoing edges is stored for each node
- In some applications it is useful to store both the adjacency matrix and the adjacency list

Representing Undirected Graphs



Graph Adjacency Matrix Adjancency List

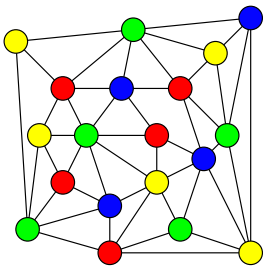
Representing Digraphs



Graph Adjacency Matrix Adjancency List

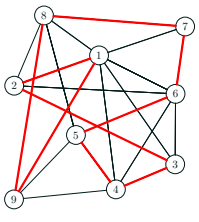
Outline

1. Graph Theory
2. Applications of Graphs
   - Geometric applications
   - Relational applications
3. Implementing Graphs
4. **Graph Problems**



Hamilton Cycle

- The Euler path problem is to find a path through a multigraph that passes through every edge once—easy to solve
- The Hamilton cycle problem is to find a cycle that goes through each vertex exactly once



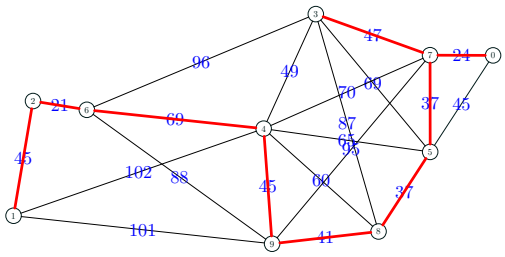
- There is no known efficient algorithm to solve this

Shortest Path and TSP

- The shortest path problem is to find a path between two nodes
- There is an efficient algorithm—see next lecture
- In the travelling salesperson problem the task is to find the shortest tour (Hamilton cycle)—we usually assume there is an edge between every pair of nodes
- There is no know efficient algorithm to solve all TSPs

Minimum Spanning Tree

- Suppose we want to construct pylons connecting a number of cities using the least amount of cable



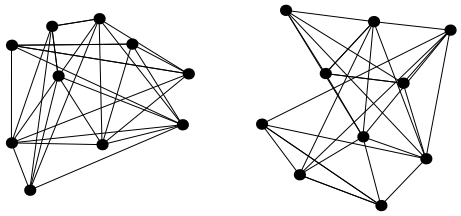
- We will study an efficient algorithm to solve this in the next but one lecture

Graph Partitioning

- The simplest version of this problem is to cut a graph into two equal halves so that you minimise the number of edges you cut
- If the edges are weighted then you want to minimise the sum of edges that are cut
- If the vertices are weighted you want to balance the sum of vertex weights in the two partitions
- An example of this problem is in dividing up a problem to run on a parallel computer
  - ⋆ Nodes are subtasks (weights on nodes are run times)
  - ⋆ Edge weights indicate communication cost
- There is no known efficient algorithm to solve this

Graph Isomorphism

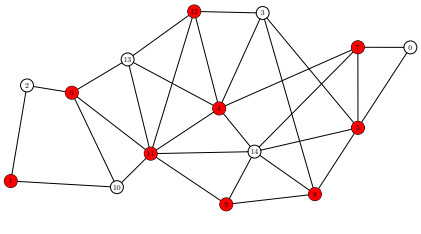
- Do two graphs have the same structure?



- There is no known efficient algorithm to solve this problem
- Theoretically it is interesting because it is not NP-complete

Vertex Cover

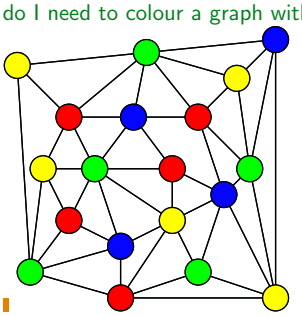
- How many guards do you need to cover all the corridors in a museum



- There is no known efficient algorithm to solve this

Graph Colouring

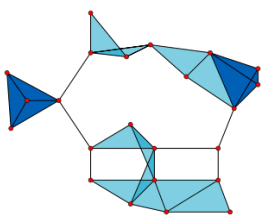
- How many colours do I need to colour a graph with no conflicts



- There is no known efficient algorithm to solve this

Other Graph Problems

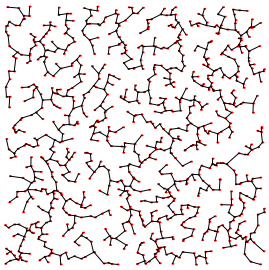
- These are only a sample of the many famous graph problems
- Others include
  - ⋆ Max-clique (hard)
  - ⋆ Maximal independent set (hard)
  - ⋆ Maximal flow problem (easy)
  - ⋆ Max-cut (hard)



Lessons

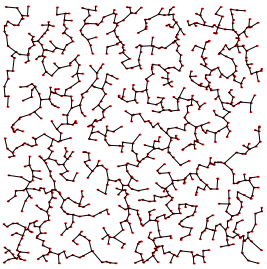
- Graphs are an important method for abstracting problems
- They appear in a huge number of disparate fields
- There are many problems for which efficient algorithms are known
- There are many problems which are believed to be hard—i.e. there aren't any efficient algorithms
- Even for hard problems there are good algorithms for finding approximated solutions

Lesson 18: *Know Your Graph Algorithms*



Weighted graph algorithms, Minimum spanning tree, Prim, Kruskal, shortest path, Dijkstra

Outline

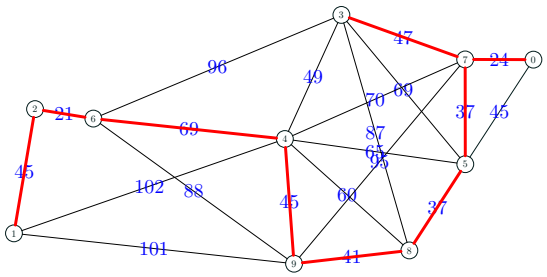


Graph Algorithms

- We consider a graph algorithm to be **efficient** if it can solve a graph problem in $O(n^a)$ time for some fixed a
- That is, an efficient algorithm runs in polynomial time
- A problem is **hard** if there is no known efficient algorithm
- This does **not** mean the best we can do is to look through all possible solutions—see later lectures
- In this lecture we are going to look at some efficient graph algorithms for weighted graphs

Minimum spanning tree

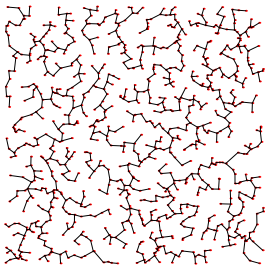
- A minimal spanning tree is the shortest tree which spans the entire graph



Greedy Strategy

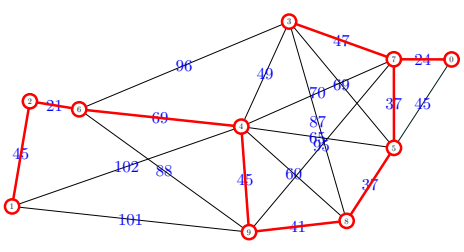
- We consider two algorithms for solving the problem
  - ⋆ Prim's algorithm (discovered 1957)
  - ⋆ Kruskal's algorithm (discovered 1956)
- Both algorithms use a **greedy strategy**
- Generally greedy strategies are not guaranteed to give globally optimal solutions
- There exists a class of problems with a **matroid** structure where greedy algorithms lead to globally optimal solutions
- Minimum spanning trees, Huffman codes and shortest path problems are matroids

Outline



Prim's Algorithm

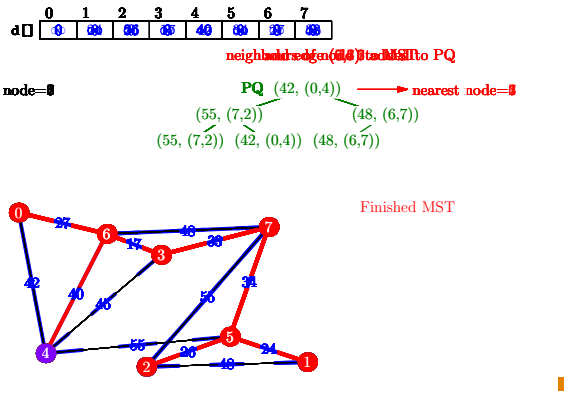
- Prim's algorithm grows a subtree greedily
- Start at an arbitrary node
- Add the shortest edge to a node not in the tree



Pseudo Code

```
PRIM(G = (V, E, w)) {
  for i←1 to |V|
    d_i ←∞            \\ Minimum 'distance' to subtree
  endfor
  E_T ←∅              \\ Set of edges in subtree
  PQ.initialise()     \\ initialise an empty priority queue
  node ← v_1          \\ where v_1 ∈ V is arbitrary
  for i←1 to |V| − 1
    d_node ←0
    for k ∈ {v ∈ V|(node, v) ∈ E} \\ k is a neighbours of node
      if ( w_node,k < d_k )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```

Prim's Algorithm in Detail



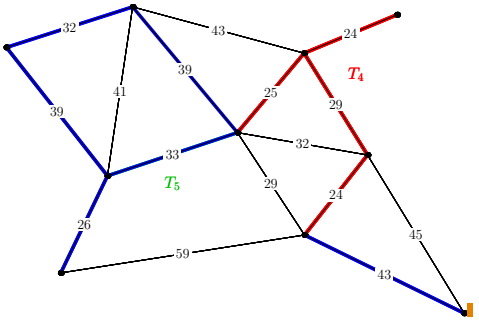
Why Does This Work?

- Clearly Prim's algorithm produces a spanning tree
  - ⋆ It is a tree because we always choose an edge to a node not in the tree
  - ⋆ It is a spanning tree because it has $|\mathcal{V}| - 1$ edges
- Why is this a minimum spanning tree?
- Once again we look for a proof by induction

Proof by induction

- We want to show that each subtree, T_i, for $i = 1, 2, \cdots, n$ is part of (a subgraph) of some minimum spanning tree
- In the base case, T_1 consists of a tree with no edges, but this has to be part of the minimum spanning tree
- To prove the inductive case we assume that T_i is part of the minimum spanning tree
- We want to prove that T_{i+1} formed by adding the shortest edge is also part of the minimum spanning tree
- We perform the proof by contradiction—we assume that this added edge isn't part of the minimum spanning tree

Contrariwise



Loop Counting

```
PRIM(G = (V, E, w)) {
  for i←0 to |V|
    d_i ←∞
  endfor
  E_T ←∅
  PQ.initialise()
  node ← v_1
  for i←1 to |V| − 1                    // loop 1  O(|V|)
    d_node ←0
    for k ∈ {v ∈ V|(node, v) ∈ E}  // inner loop O(|E|/|V|)
      if ( w_node,k < d_k )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )     // O(log(|E|))
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```

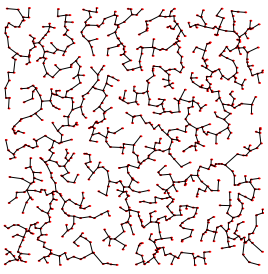

Run Time

- The worst time is

$$O(|\mathcal{V}|) \times O\left(\frac{|\mathcal{E}|}{|\mathcal{V}|}\right) \times O(\log(|\mathcal{E}|)) = O\left(|\mathcal{E}|\log(|\mathcal{E}|)\right)$$

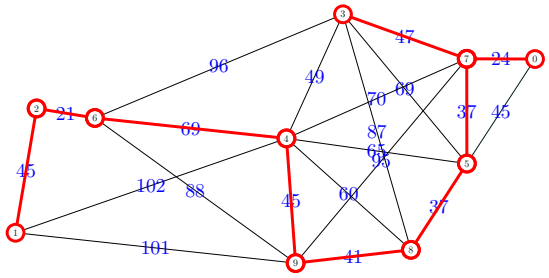
- Note that $|\mathcal{E}| < |\mathcal{V}|^2$
- Thus, $\log(|\mathcal{E}|) < 2\log(|\mathcal{V}|) = O\left(\log(|\mathcal{V}|)\right)$
- Thus the worst case time complexity is $|\mathcal{E}|\log(|\mathcal{V}|)$

Outline



Kruskal's Algorithm

- Kruskal's algorithm works by choosing the shortest edges which don't form a loop



Pseudo Code

```
KRUSKAL(G = (V, E, w))
{
  PQ.initialise()
  for edge ∈ |E|
    PQ.add( (w_edge, edge) )
  endfor

  E_T ← ∅
  noEdgesAccepted ←0

  while (noEdgesAccepted < |V| − 1)
    edge ←PQ.getMin()
    if E_T ∪ {edge} is acyclic
      E_T ←E_T ∪ {edge}
      noEdgesAccepted ←noEdgesAccepted +1
    endif
  endwhile

  return E_T
}
```

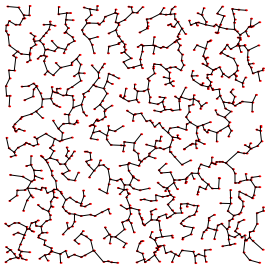

Analysis

- Kruskal's algorithm looks much simpler than Prim's
- The sorting takes most of the time, thus Prim's algorithms is $O(|\mathcal{E}|\log(|\mathcal{E}|)) = O(|\mathcal{E}|\log(|\mathcal{V}|))$
- We can sort the edges however we want—we could use quick sort rather than heap sort using a priority queue
- But we haven't specified how we determine if the added edge would produce a cycle

Cycling

- For a path to be a cycle the edge has to join two nodes representing the same subtree
- To compute this we need to quickly **find** which subtree a node has been assigned to
- Initially all nodes are assigned to a separate subtree
- When two subtrees are combined by an edge we have to perform the **union** of the two subtrees
- This is a tricky but standard operation known as **union-find**

Outline



Union-Find

- In the union-find algorithm we have a set of objects $x \in \mathcal{S}$ which are to be grouped into subsets $\mathcal{S}_1$, $\mathcal{S}_2$, ...
- Initially each object is in its individual subset (no relationships)
- We want to make the **union** of two subsets (add relationship between elements)
- We also want to **find** the subset given an element
- This is a common problem for which we will write a class `DisjointSets` to perform fast `unions` and `finds`

DisjointSets

- We want to create a class

```
class DisjointSets
{
    DisjointSets(int numElements) {/* Constructor */}
    int find(int x) {/* Find root */}
    void union(int root1, int root2) {/* Union */}

    private:
      int[] s;
}
```

- Where `find(x)` returns a unique identifier for the subset which element `x` belongs to
- The array `s` contains labelling information to implement `find(x)`

The Union-Find Dilemma

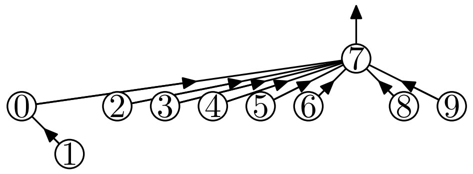
- A natural algorithm to perform finds is to maintain an array returning a subset label for each element—this makes `find` fast
- However, every time we combine two subset we have to change all the labels in this array (taking $O(n)$ operations)
- If we are unlucky the cost of performing n unions is $\Theta(n^2)$
- If we ensure that we relabel the smaller subset then the time complexity is $\Theta(n\log(n))$
- Fast *finds* seems to give slow(ish) *unions*
- What about the other way around?

Fast Union

- To achieve fast unions we can represent our disjoint sets as a forest (many disjoint trees)
- Every time we perform a union we make one of the trees point to the head of the other tree
- The cost of `find` depends on the depth of the tree
- To make unions efficient we make the shallow tree a subtree of the deeper tree

Putting it Together

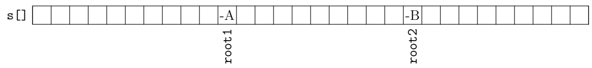
`find(6)=7`



7	0	7	7	7	7	7	-3	7	7
0	1	2	3	4	5	6	7	8	9

Smart Union

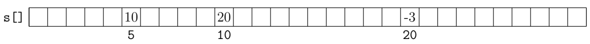
```
DisjointSets::DisjointSets(int numElements)
{
    s = new int[numElements];
    for(int i=0; i<s.length; i++)
        s[i] = -1;                  // roots are negative number
}

void DisjointSets::union(int root1, int root2)
{
    if (s[root2]<s[root1]) {        // root2 is deeper
        s[root1] = root2;           // make root2 the root
    } else {
        if (s[root1]==s[root2])
            s[root1]--;             // update height if same
        s[root2] = root1;           // make root1 new root
    }
}
```



Path Compression

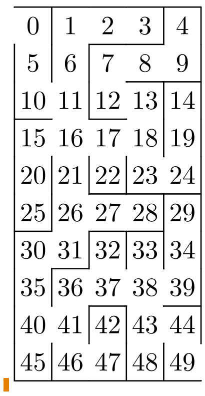
- To speed up `find` we relabel all nodes we visit during `find` by the root label

```
public int DisjointSets::find(int index)
{
    if (s[index]<0)
        return index;
    else
        return s[index] = find(s[index]);
}
```



Mazes

- Union-Find is a data structure which can occur in very different applications
- One application is building a maze
- Start from a complete lattice
- Remove a randomly chosen edge if it connects two unconnected regions
- Stop when the start and end cell are connected
- Or better after all cells are connected



Time Complexity of Union-Find

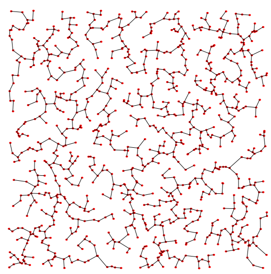
- If we perform M finds and N unions then the time complexity is $O(M \log_2^*(N))$
- Where $\log_2^*(N)$ is the number of times you need to apply the logarithm function before you get a number less than 1
- In practice $\log_2^*(N) \le 5$ for all conceivable N

  $\log_2(\log$ [illegible]

- The proof of this time complexity is rather involved

Outline

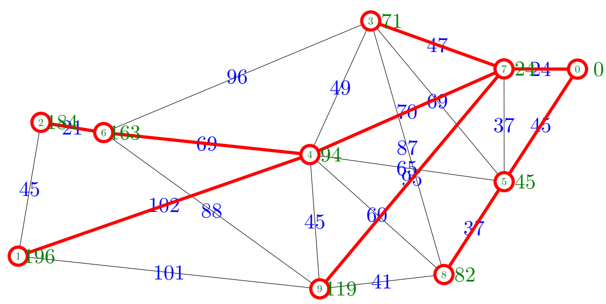
1. Minimum Spanning Tree
2. Prim's Algorithm
3. Kruskal's Algorithm
4. Union Find
5. **Shortest Path**



Shortest path

- We can efficiently compute the shortest path from one vertex to any other vertex
- This defines a spanning tree, but where the optimisation criteria is that we choose the vertex that are closest to the *source*
- To find this spanning tree we use Dijkstra's algorithm where we successively add the nearest node to the source which is connected to the subtree built so far
- This is very close to Prim's algorithm and has the same complexity

Dijkstra's Algorithm



Pseudo Code

```
DIJKSTRA(G = (V, E, w), source) {
  for i←0 to |V|
    d_i ←∞          \\ Minimum 'distance' to source
  endfor
  E_T ←∅            \\ Set of edges in subtree
  PQ.initialise()  \\ initialise an empty priority queue
  node ←source
  d_node ←0
  for i←1 to |V| − 1
    for k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,k + d_node < d_k )
        d_k ←w_node,k + d_node
        PQ.add( (d_k, (node, k)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    while next_node not in subtree
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```


Compare to Prim's Algorithm

```
PRIM(G = (V, E, w)) {
  for i←1 to |V|
    d_i ←∞          \\ Minimum 'distance' to subtree
  endfor
  E_T ←∅            \\ Set of edges in subtree
  PQ.initialise()  \\ initialise an empty priority queue
  node ← v_1        \\ where v_1 ∈ V is arbitrary
  for i←1 to |V| − 1
    d_node ←0
    for k ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,k < d_k )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```

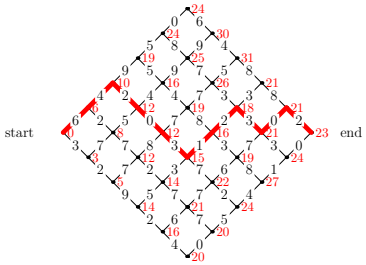

Dijkstra Details

- Dijkstra is very similar to Prim's (it differs in the distances that are used)
- It has the same time complexity
- It can be viewed as using a greedy strategy
- It can also be viewed as using the dynamic programming strategy (see lecture 22)

Lessons

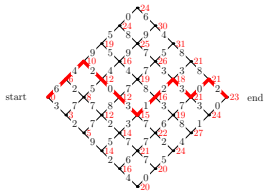
- There are many efficient (i.e. polynomial $O(n^a)$) graph algorithms
- Some of the most efficient ones are based on the Greedy strategy
- These are easily implemented using priority queues
- Minimum spanning trees are useful because they are easy to compute
- Dijkstra's algorithm is one of the classics

Lesson 19: *Dynamic Programming*



Dynamic programming, line breaking, edit distance, Dijkstra, TSP

Outline

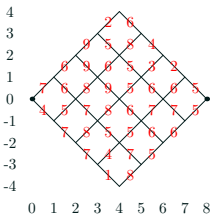


Dynamic Programming

- One of the most powerful strategies for solving optimisation problems is **dynamic programming**
  - ⋆ Build a set of optimal partial solutions
  - ⋆ Increase the size of the partial solutions until you have a full solution
  - ⋆ Each step uses the set of optimal partial solutions found in the previous step
- Developed by Richard Bellman in the early 1950's
- The name is unfortunate as it doesn't have much to do with programming

A Toy Problem

- Consider the problem of find a minimum cost path from point (0,0) to (8,0) on the lattice



- The costs of traversing each link is shown in red
- The cost of a path is the sum of weights on each link

Brute Force

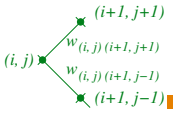
- The obvious brute force strategy is to try every path
- For a problem with n steps we require $n/2$ to be diagonally up and $n/2$ to be diagonally down
- The total number of paths is

$$\binom{n}{n/2} \approx \sqrt{\frac{2}{\pi\, n}}\, 2^n$$

- For the problem shown above with $n = 8$ there are 70 paths
- For a problem with $n = 100$ there are 1.01×10^{29} paths

Building a Solution

- We can solve this problem efficiently using dynamic programming by considering optimal paths of shorter length
- Let $c_{(i,j)}$ denote the cost of the optimal path to node (i, j)
- We denote the weights between two points on the lattice by $w_{(i,j)(i+1,j\pm1)}$



- Clearly $c_{(0,0)} = 0$

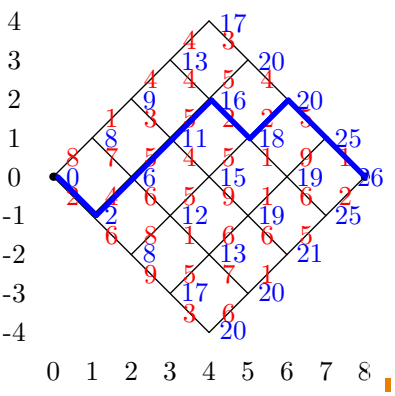
Forward Algorithm

- Suppose we know the optimal costs for all the edge in column i
- Our task is to find the optimal cost at column $i + 1$
- If we consider the sites in the lattice then the optimal cost will be

$$c_{(i+1,j)} = \min\big(c_{(i,j+1)} + w_{(i,j+1)(i+1,j)},\ c_{(i,j-1)} + w_{(i,j-1)(i+1,j)}\big)$$

- This is the defining equation in dynamic programming
- We have to treat the boundary sites specially, but this is just book-keeping

Example



Backward Algorithm

- Having found the optimal costs $c_{(i,j)}$ we can find the optimal path starting from $(n, 0)$
- At each step we have a choice of going up or down
- We choose the direction which satisfies the constraint

$$c_{(i,j)} = c_{(i-1,j\pm1)} + w_{(i-1,j\pm1)(i,j)}$$

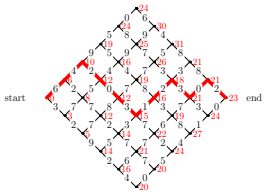
- If both directions satisfy the constraint we have more than one optimal path

Time Complexity

- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(\frac{n+1}{2})^2$ lattice point
- It took constant time to compute each cost so the total time to perform the forward algorithm was $\Theta(n^2)$
- The time complexity of the backward algorithm was $\Theta(n)$
- This compares with $\exp(\Theta(n))$ for the brute force algorithm

Outline

1. Dynamic Programming
2. **Applications**
   - Line Breaks
   - Edit Distance
   - Dijkstra's Algorithm
3. Limitation



Applications of Dynamic Programming

- Dynamic programming is used in a vast number of applications
  - ⋆ String matching algorithms
  - ⋆ Shape matching in images
  - ⋆ Dynamical time-warping in speech
  - ⋆ Hidden Markov Models in machine learning
- Unlike greedy algorithms the idea is readily extended to many different applications

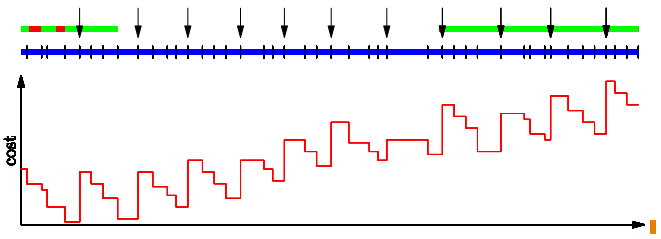
Using Dynamic Programming

- The challenge is recognising that you can use dynamic programming and representing the problem right
- Learn this from examples
- Consider writing a word processor that splits paragraphs up into lines
- You want to choose the line breaks so that the lines are all roughly the same length
- This is a global optimisation task

  minimise the total number of spaces left at the end of each line

Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



Real Word Breaking

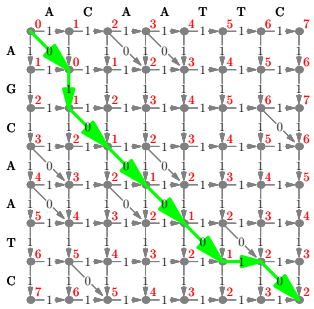
- In advanced word processing you care about hyphenation, large gaps at the end of lines, etc.
- These all affect the way you would assign costs
- Dynamic programming is used in LaTeX to produce nice line breaks
- A similar algorithm is used to produce nice page breaks

Inexact Matching

- A second example of dynamic programming is to find inexact matches
- The edit distance between two strings is the number of changes needed to move from one string to another
- The exact metric depends on the application, but might include number of substitutions, insertions and deletions
- This has many applications, e.g. in genomics to see what DNA strings (or proteins) are related

Edit Distance

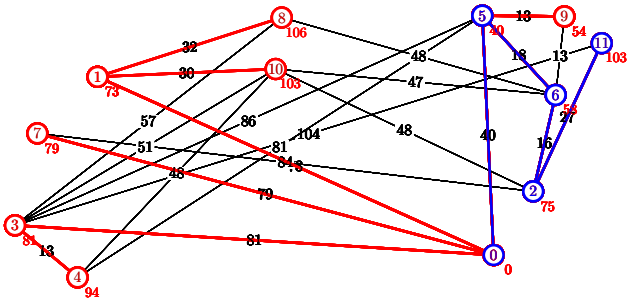
- What is the minimum edit distance between ACAATTC and AGCAATC



Dijkstra's Algorithm

- We saw early Dijkstra's algorithm for find the minimum distance between a source and destination node
- We grouped this with the greedy algorithms as we choose the next node to add to the minimum-distance spanning tree to be the closest node to the source we could access
- However, we should perhaps more rightly identify it as using a dynamic programming strategy as we are building up the cost of getting of the partial solution to reach a node
- We use the greedy strategy to ensure that we always find the shorter paths first

Going from Node 0 to Node 11



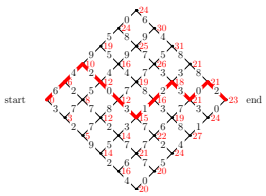
Uses of Dynamic Programming

- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text

- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman–Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the subset sum and knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted

- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content aware image resizing)
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Some approximate solution methods for the linear search problem
- Kadane's algorithm for the maximum subarray problem

Outline

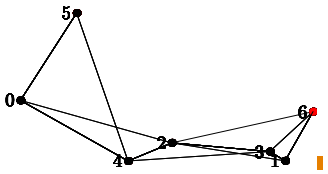


When You Can't Use It

- Not all problems can be split neatly to make dynamic programming possible
- Dynamic programming works on problems with some natural ordering
- We need this to build up a list of optimum cost of partial solutions—these have to depend on the cost of previous partial solutions
- Sometime no natural ordering exists

Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- The problem is there are $\binom{n}{k}$ subsets consisting of k cities out of a possible n
- The total number of subsets that need to be considered is

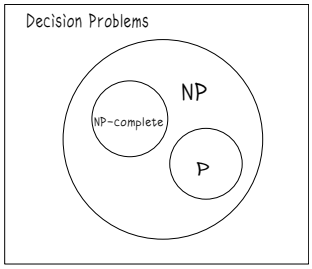
$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- The time complexity of the DP solution is $n^2\,2^n$ which is better than $n!$ and is currently the fastest known exact algorithm for TSP, but it ain't very useful in practice!

Conclusions

- Dynamic programming is one of the most powerful strategies for solving hard optimisation problems
- It works by iteratively building up costs for partial solutions using the costs of smaller partial solutions
- When it works it is great and there are hosts of practical algorithms which use DP
- However, it doesn't always work

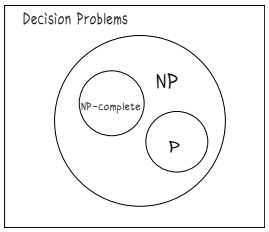
Lesson 16: *Know What's Possible*



Combinatorial optimisation, NP-completeness, polynomial reduction

Outline

1. **Motivation**
2. P, NP and NP-complete
3. Polynomial Reduction



Exponentially Large Search Spaces

- We have seen a large number of decision problems and optimisation problems involving an exponentially large search space
- For some of these we have found efficient algorithms (greedy algorithms, divide and conquer, dynamic programming, . . .)
- For other problems we have found good algorithms (backtracking, branch and bound), but they are not necessarily polynomial
- Can we say anything general about how easy they are to solve

Types of Problems

- We concentrate here on two types of problems
  - Decision Problems
  - Combinatorial Optimisation Problems
- Decision problems are problems with a true/false answer, e.g. is it possible to cross all the bridges of Königsberg once?
- We showed earlier that backtracking can be used to find a solution which answers the decision problems, e.g. Hamiltonian circuit problem
- There are many other decision problems, but the most famous is satisfiability or SAT

SAT

- Given n Boolean variables $X_i \in \{T, F\}$
- m disjunctive (or's) clauses, e.g.

$$c_1 = X_1 \vee \neg X_2 \vee X_3$$
$$c_2 = \neg X_2 \vee X_3 \vee X_5$$
$$\vdots \qquad \vdots$$
$$c_m = X_2 \vee \neg X_4 \vee \neg X_5$$

- Find an assignment, $\boldsymbol{X} \in \{T, F\}^n$ which satisfies all the clauses
- We can view this as finding an assignment that makes the formula $f(\boldsymbol{X})$ true where

$$f(\boldsymbol{X}) = c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

Decisions and Optimisation Problems

- Often we can cast a decision problem as an optimisation problem
- E.g. the MAX-SAT problem is to find an assignment of variables that satisfies the most clauses
- If we can solve the MAX-SAT optimisation problem we can solve the decision problem
- We can also cast optimisation problems as decision problems

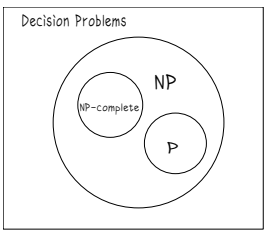
  Does there exist a TSP tour shorter than 200 miles?

Combinatorial Optimisation Problems

- In the set of discrete optimisation problems an important class are those that involve *combinatorial objects* such as permutations, binary string, etc.
- Optimisation problems involving such objects are termed **combinatorial optimisation problems**
- Classical examples of such problems include
  - Travelling Salesperson Problem (TSP)
  - Graph colouring
  - Maximum Satisfiability (MAX-SAT)
  - Many scheduling problems
  - Bin-packing
  - Quadratic integer problems

Outline

1. Motivation
2. **P, NP and NP-complete**
3. Polynomial Reduction



Polynomial Problems

- Some optimisation problems are "easy"—there are known polynomial time algorithms to solve them
  - ⋆ Minimum spanning tree
  - ⋆ Shortest path
  - ⋆ Linear assignment problem
  - ⋆ Maximum flow between any two vertices of a directed graph
  - ⋆ Linear programming
- Many apparently different problems can be mapped onto these problems

NP-Hard Problems

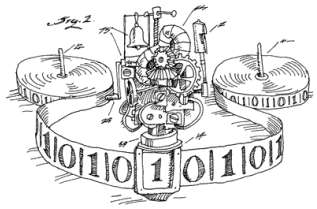
- Is it possible to solve the TSP in polynomial time?
- Answer: **maybe**
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

Decision Problems

- Decision problems are problems with a true or false answer
- E.g. does a TSP have a tour less than $2\,000$ miles?
- Algorithmic complexity theory deals with classes of decision problems that have some characteristic size, n
- E.g. n is the number of cities in a TSP
- Any decision problem that can be answered with an algorithm that runs in polynomial time (n^a) on a normal computer (Turing machine) is said to be in class P
- E.g. The decision problem "Is there a path of length less that 450 miles between Southampton and Glasgow?" is in class P

Turing Machines

- A Turing machine can be viewed as a very dumb computer which computes by having a number of states and reading and writing to a tape



- Although dumb, it can do anything that any other computer can do (although it may take polynomially more time)

Non-Deterministic Turing Machines

- A non-deterministic Turing machine is a magic machine that can guess the answer and then use a normal Turing machine to verify the answer
- We assume that it guesses right in the first go
- No one knows how to simulate a non-deterministic Turing machine in polynomial time
- We can simulate it in exponential time by trying all possible guesses

Class NP

- Any decision problem that can be answered with an algorithm that runs in polynomial time on a non-deterministic Turing machine is said to be in class NP
- A whole lot of decision problems belong to class NP
- All decision problems with polynomial algorithms are also in class NP (P⊂NP)
- *"Is the game of chess winnable by white?"* is **not** in class NP

Belonging to NP

- For a problem to belong to class NP it must
  - ⋆ be a decision problem (true of false)
  - ⋆ describable by some polynomial sized string in the length of the input n
  - ⋆ be verifiable if true in polynomial time on a normal Turing machine (e.g. a computer)
- To be verifiable it is sufficient for the decision problem to have a "**witness**" which is usually a solution that can be checked in polynomial time
- E.g. in TSP the witness would be a tour which satisfies the condition

Using Non-Deterministic Turing Machines

- A witness is sufficient for a problem to be in NP since a non-deterministic Turing machine can guess the witness in one step and then proceed to check the witness is true in polynomial time
- Thus TSP is also in NP
- As are graph-colouring, SAT, Max-Clique, Hamilton cycle and countless others

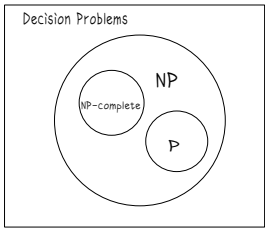
Class NP-complete

- In 1971 Cook showed that you could represent any NP problem on a non-deterministic Turing machine by a polynomially sized SAT (Satisfaction) problem
- Thus if you could solve SAT in polynomial time you can use that to simulate a non-deterministic Turing machine in polynomial time
- SAT is therefore an example of one of the hardest problems in NP since if you can solve SAT in polynomial time you can solve all problems in NP in polynomial time
- These hardest problems in NP are said to be in NP-complete
- If there existed a polynomial time algorithm for SAT then all problems in NP could be performed in polynomial time so that NP=P

Idea Behind Cook's Theorem

- Cook showed that a non-deterministic Turing machine could be encoded as a big SAT formula
- The evolution of the state and tape was represented by a big tableau ($n^k \times n^k$-table where n^k is the time it takes for the Turing machine to verify the answer)
- The structure of the clauses reflect the rules the Turing machine operates
- If the clauses are simultaneously satisfiable then there exists an input that satisfies the conditions

Outline



Polynomial Reductions

- Given two decision problems A and B we say there is a **polynomial reduction** from A to B if
  - Every instance of A can be mapped to an instance of B:
  - The truth of the instance A is the same as the corresponding instance B
- We can therefore use B to solve A
- So: $B \in P \rightarrow A \in P$
- The contrapositive of this statement is

  $A \notin P \rightarrow B \notin P$

SAT to 3-SAT

- We can reduce a clause with 4 variable to a clause with 3

  $$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \equiv (X_1 \vee \neg X_3 \vee Z) \wedge (\neg Z \vee X_6 \vee \neg X_{10})$$

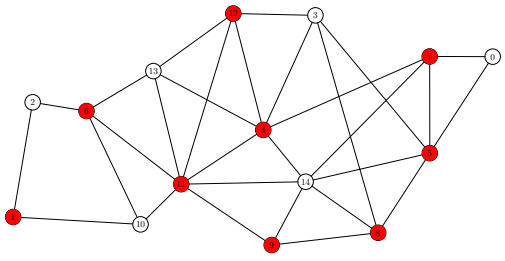
- In doing so we increase the number of variables and the number of clauses to satisfy
- We can similarly reduce a clause with more variables

  $$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \vee \neg X_{11} \vee X_{15} \equiv$$
  $$(X_1 \vee \neg X_3 \vee Z_1) \wedge (\neg Z_1 \vee X_6 \vee Z_2) \wedge (\neg Z_2 \vee \neg X_{10} \vee Z_3) \wedge (\neg Z_3 \vee \neg X_{11} \vee X_{15})$$

- Because every instance of SAT can be written as a 3-SAT problem which is only polynomially larger than the SAT problem, 3-SAT is also NP-complete

Vertex Cover

- The vertex cover problem is: "can we choose K (9) vertices of a graph such that every edge is connected to a chosen vertex?"

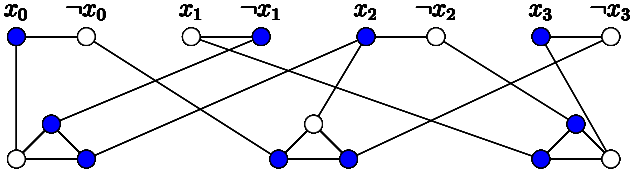


Vertex Cover is NP-complete

- Vertex cover is obviously in NP as a set of K vertices acts as a witness, i.e. it can be checked that it covers all edges
- To show vertex cover is NP-complete we show that every instance of 3-SAT is reducible to vertex cover
- The idea is to show that any 3-SAT problem with variables $\{X_1, X_2, \ldots, X_n\}$ and clauses $\{c_1, c_2, \ldots, c_m\}$ can be encoded as a vertex cover problem

3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

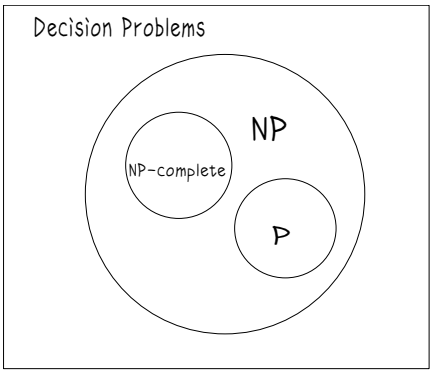


Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

Other Examples of NP-complete

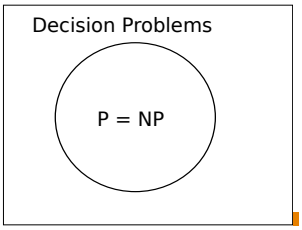
- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete
- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date
- These include
  - ⋆ TSP
  - ⋆ Graph colouring
  - ⋆ Many scheduling problems
  - ⋆ Bin-packing
  - ⋆ Quadratic integer problems

Structure of Decision Problems



$P \neq NP$?

- No one has proved that any problem in NP is not solvable in polynomial time
- If any NP-complete problem was solved in polynomial time all problems in NP would be solve and NP=P



NP-Hard

- TSP is not a decision problem—although we can make it into one—Is there a tour shorter that L?
- However, if we can find the shortest tour in polynomial time we could solve the TSP decision problem
- Thus finding the shortest tour is at least as hard as solving the decision problems
- Problems that are at least as hard as NP-complete decision problems are said to be in **NP-hard**
- Graph colouring (finding a colouring with the least number of conflicts), job scheduling, etc. are all examples of NP-hard problems

Not All Hard Problems are NP-Hard

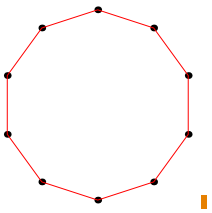
- Graph isomorphism, GI, (are two graphs identical up to a relabelling of the vertices?) has not been proved to be NP-complete—it is postulated that

$$GI \in \text{NP} \wedge GI \notin \text{P} \wedge GI \notin \text{NP-complete}$$

- Factoring is *not* believed to be NP-hard, but it is believed to be sufficiently hard that most banks use an encryption technique based on people not being able to factor large numbers easily
- For large problems polynomial algorithms can take too long

Not All NP-Hard Problem Instances are Hard

- NP-hardness is a worst case analysis
- It means there exist some instance of the problem that we don't know how to solve in polynomial time
- Many instance of the problem might be rather easy to solve
- What is the optimal TSP tour for the problem below?



Not All NP-Hard Problems are Hard

- For some problems almost all instances appear easy
- E.g. The subset-sum problem
  - ⋆ Given a set of numbers find a subset whose sums is as close as possible to some constant
  - ⋆ Subset-sum is in NP-hard but there exist a "pseudo-polynomial time" algorithm which solves almost every instance in polynomial time
- Many problems including subset-sum are known to be easy to approximate
- For other problems even finding a good approximation is known to be NP-hard

Lessons

- There exist efficient algorithms for many problems. . .
- . . . but probably not for all
- There are no known polynomial algorithm for any NP-complete problem
- These include many famous problems: TSP, graph-colouring, scheduling, . . .
- If you could find a polynomial algorithm for any of these problems then you could use it to solve all problems in NP in polynomial time