Data Structures and Algorithms
comp1009

Books

Lecture 1

a)What is a data structure

- Container for data
  - e.g sets, stacks, lists, trees, graphs
- Clean interface, e.g. push, pop, delete
- Own algorithms

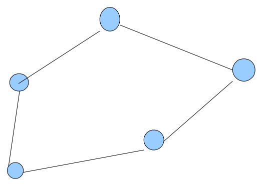
b)OO Software

- Abstraction from details of problem
- Declaration of intention
  - Intentions can be seen from the data structure
- Clean interfaces
- Hidden implementations
- Makes programs readable and maintainable
- Re-use of code
  - Don't even have to write it yourself!

Lecture 2

a)Travelling Salesman Problem

- Given a set of cities and distances (or costs between them)
- Find the shortest tour around all cities



$\frac{(n-1)!}{2}$ possible routes

- With lots of cities there are many many routes which will take years to compute all of them

b)Sorting

- Comparison between sorts
  - Insertion Sort
  - Shell Sort
  - Quick Sort
- There's a *RIGHT WAY* and a *WRONG WAY* to tackle every problem
- Only really care with large inputs

c)Estimating Run Times

- Compute the asymptotic leading functional behaivor
  - Asymptotic: For large values of n
  - Leading: ignore the less significant parts of the formula
- An algorithm that takes $4n^2+2n+199$ operations, write this as ,Θ(n^2).
- This is called the time complexity
- If it takes *x* seconds to complete an algorithim for *k* inputs, then for *n* inputs it will take $\frac{x \times g(n)}{g(k)}$ seconds to complete, where g(*x*) is the functional behaviour.

d)Big Θ, Big O and BigΩ

- Big Θ
  - An exact expression of the time complexity, a strong result
- Big O
  - An upper band on the time complexity, I.e it would take less time that this.
- BigΩ
  - A lower band on the time complexity

Lecture 3 – Arrays

a)What is

- A continuous chunk of memory
- Random Access Time Θ(1) ~ 1 time step
- Very effective use of memory

- downsides:
  - Simple arrays have fixed lengths
  - Very often don't know many many items we want to store in our array

b)Variable Length Arrays

- Solution to not knowing how many items to store

- When run out of space, double capacity and copy in the old elements

c)General Time Analysis

- See lecture slides
- Working out the number of operations needed to extend an array of size n to fit N items

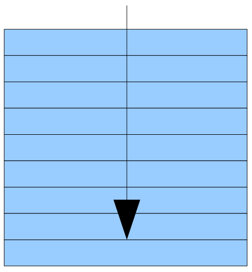
Lecture 4 – Abstract Data Types

a)Abstract Data Types

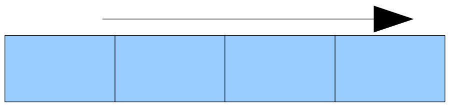
- Encapsulation
- Examples:
  - Stacks
  - Queues
  - Priority Queues
  - Sets
- Lots of possible implementations for these types
- Use of an interface makes code easier to use (and understand)

b)Stacks – LIFO

- Stacks reduce access to memory (not random access)
- Suitable though for a large number of algorithms
- Operations
  - pop() - return and remove the top element
  - push() - add an element onto the stack
  - peek() - return the top element but don't remove it



c)Queues – FIFO



- Operations
  - peek()
  - dequeue()
  - enqueue()

d)Priority Queues

- Elements inserted with a priority
- Usually the value for the priority is a lower number for a higher priority (.i.e the order in importance)

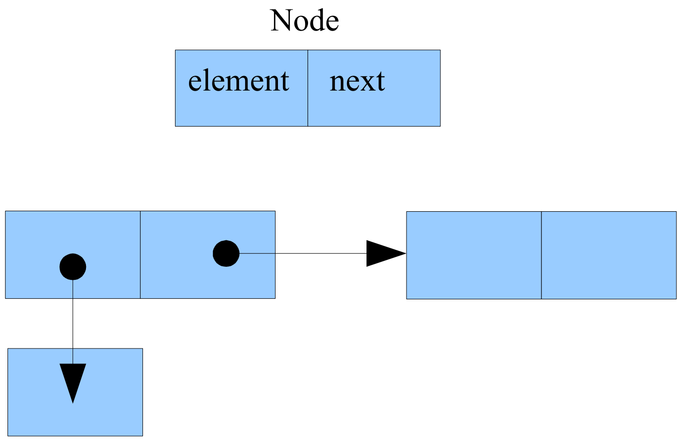
Lecture 7 – Linked Lists

a)Linked Lists

- Units of data that point to other units
- Examples:
  - Binary Tree
  - Graphs
  - Linked Lists

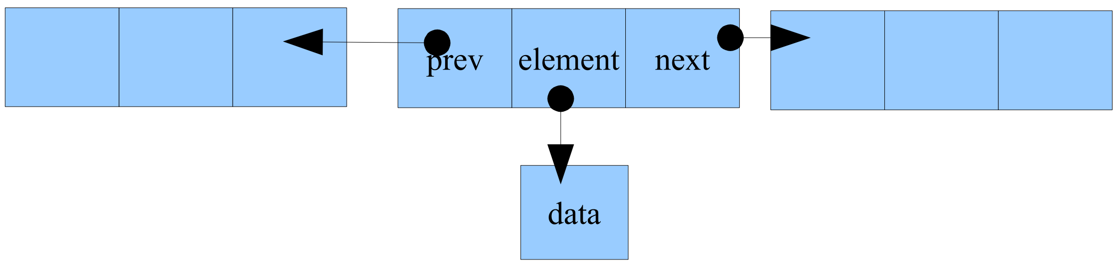
- Classic Structure in all the books but which is almost entirely useless, just good as an introduction
- NO RANDOM ACCESS

b)Singly Linked List

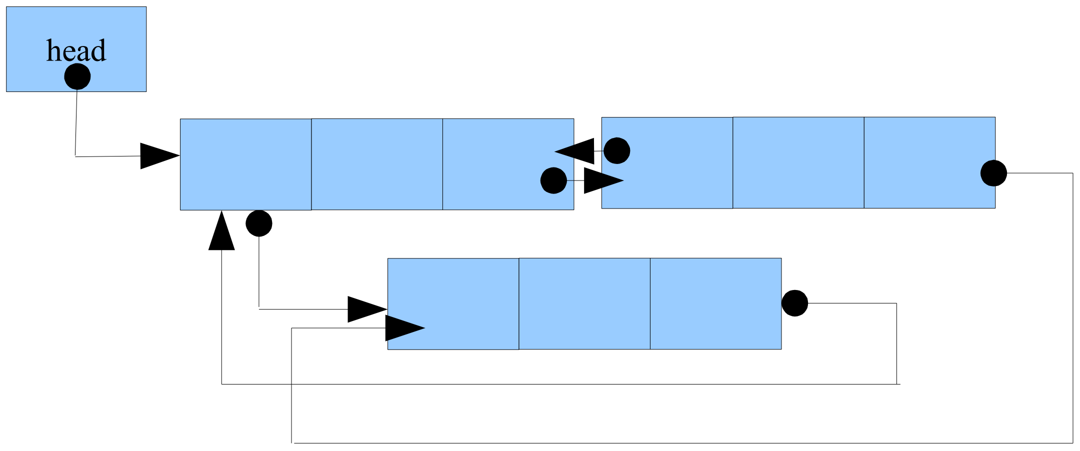


Element is data
Next is a pointer or reference to the next Node not the node itself!

d)Doubly Linked List



Java Implementation:



Lecture 8

a)Software Design Patterns

- A solution method for commonly met problems in Software Engineering
- In early 90's it was realised that many problems reoccur in computing and books made to maintain good solutions to these problems

b)Destructor

- finalise() is the de-constructor in both C++ & Java
- In C++ its called when the object goes out of scope
- In Java its called when the garbage collector tries to remove the object

c)Comparing Objects pattern

- Specify the ordering
- Also true when comparing objects to find max or min

d)Patterns

- Good practice to follow patterns
- Requires skill to make them
- and Disciplines to follow them

Lecture 9 – Recursion

a)Recursion

- Base case – where the problem is trivially solved
- Recursive clause – drives towards the base case

b)Euclid GCD

- gcd(A,B) = gcd(B,A mod B)

c)Programming

- Catch base case before the recursive clause
- Ensure each step is simpler than the last
- Assume all sub-steps will work

d)Cost of recursion

- Each function costs time and resources
- Recursion can be frequently replaced with loops
    - .i.e. recursive Fibonacci runs in exponential time but with a for loop it runs linear time

Lecture 10 – Trees

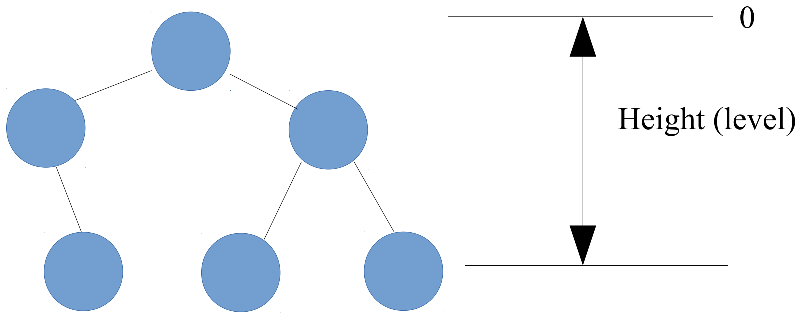
a)Trees

- Major way of structuring data
- Lots of trees but only a covering a few in class

b)What is a tree?

- Mathematically
- An acyclic undirected graph
    - Graph being a structure with nodes, vertices and edges
    - Undirected – edges grow both ways
    - Acyclic – no cycles in the graph

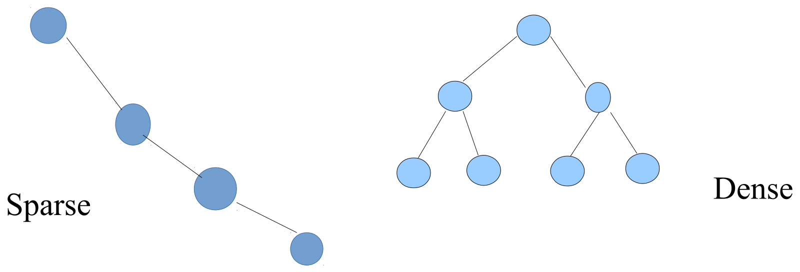
c)Binary Trees



- Each parent has up to 2 children and no more
- Not necessarily ordered nodes
- Applications?
    - Expression tree for arthritic

d)Binary Search Tree

- Ordered binary tree
- Recursively defined
- Easy to search with recursion
- Similar to sets as there is no repeat
- Dense random B.S.Ts are fast to search than sparse ones



e)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

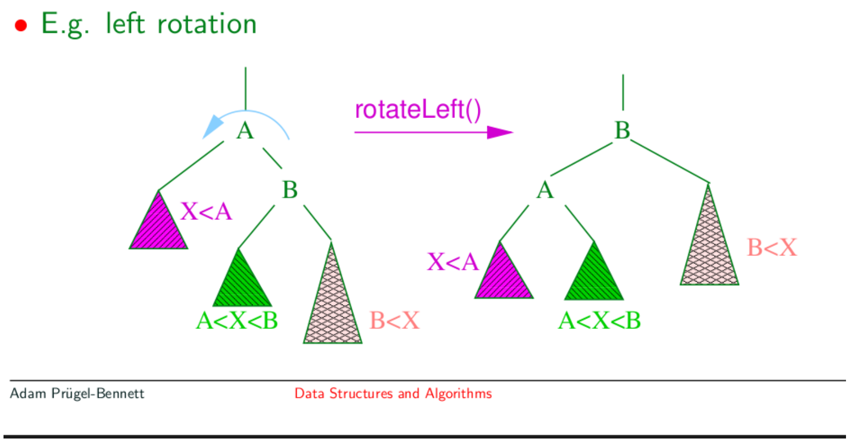
f)Deletion

- Trivial to delete a node with no children
- With one child, you connect the parent to the child of the parent of the node you're deleting
- With two children, replace with successor and delete node with successor (recursive?)

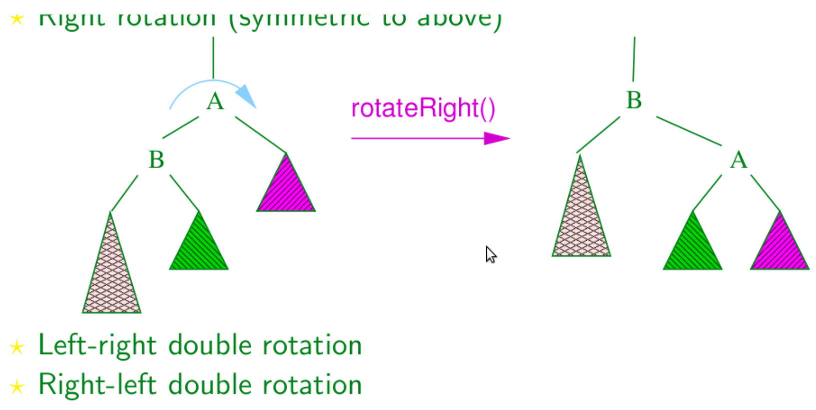
g)Balancing trees

- Important to balance trees so that searching is efficient
- Dense trees are the ideal, spare trees are too similar to linked lists
- To ensure against this, algorithms developed to balance trees out

h)Rotations



- E.g. left rotation



★ Right rotation (symmetric to above)

★ Left-right double rotation
★ Right-left double rotation

Also double rotations.... rotateLeftRight and rotateRightLeft

These operations are used to balance the tree out

Example code...

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

i)AVL

- One algorithm to balance trees...
- Every parent node has a value of either -1,0 or +1 representing its balance
- When adding an item, iterate up through the parents altering the balance appropriately,
- If any node has a balance outside the range -1..+1 then it needs rebalancing

j)Red-Black

- Slightly more complex to code than AVL
- But quicker for insertion and deletion
- Used by Java and C++

- 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
- Root is black

- 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

k)Maps

- Maps use trees where the KEY is the compared value but there is an extra element to store the data associated with this KEY called the VALUE
- Map.Entry<K,V> object in Java represents this
- Without the value it is just a set

Heaps, Priority queues

a)Heaps

- Different to memory heaps, this is a data structure
- Easy to code up for yourself (and recommended too)
- Form of priority queue
- A (min-)heap (from one perspective) is a binary tree
- Also a complete tree
- Can include items twice

b)Adding to a heap

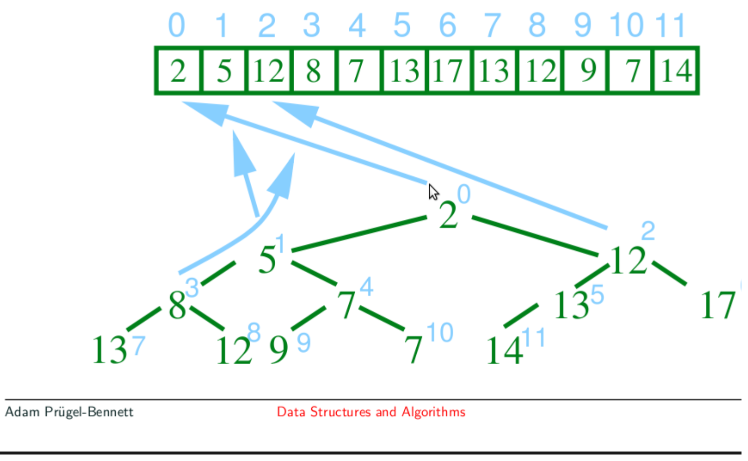
- Add element as node in next available space
- Compare with parent, if parent is bigger than element, swap them over
- Keep going up until it can't swap any more

c)Poping from a heap

- Returns item at the root of the heap
- Swaps last item to the top of the heap and then percolate down until it reaches the bottom, swapping with the lowest of the two children

d)Array Heaps

- Heaps can be implemented using simple arrays as they are complete trees



- The root of a tree is at array location 0
- The last element in the heap is at array location size()-1
- The parent of a node k is at array location (k − 1)/2
- The children of node k are at array locations 2k + 1 and 2k + 2

d)Priority Queues

- Priority queues can be made using heaps where the next item is the root of the tree
- These can be very efficient being O(log(n)) and this is the depth of the tree
- Heaps can also be used for sorting (trivial) but this is a bit slow compared to quick sort etc... but ok for partial sorts

e)Huffman Codes

- Encoding where frequently used symbols are given shorter codes
- However giving them simple codes can confuse things, a heap is needed to keep each symbol unique
- When translating a bit pattern, choose left child on a 0 and right child on a 1 until you reach a node that has a character/symbol with it

f)Huffman heap

- Order characters by number of occurrences (priority queue) and keep track of the number of occurrences of each one
- Group two lowest frequency symbols together in a sub tree, remove from list then add back in as a pseudo node until there is only 1 node left which becomes your huffmantree/heap

Anaylse!

a)Code and Pseudo Code

- Data structures are a bit language dependant due to implementation and what's available
- Algorithms are language independent
- Pseudo code is used to represent algorithms on paper

b)Pseudo Code

- No standard as such
- But....
  - $\leftarrow$ used for assignment
  - **a** used to represent arrays

c)Searching

- Dumb Search → go through array until element found
- 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
- Best case:
  - The best case is when the element is in the first location
  - This takes 1 comparison
- Average case:
  - Assume every location is equal likely to hold the key
- For an unsuccessful search n comparison are necessary

- Binary search → divide the array into two and compare middle element with item searched for to see if you go search again in the left or right half

  Item not found if lower pointer or higher pointer are out of order
- Number of comparisons = $log2(n)+1$

d)Sorting

- One of the most studied algorithms
- "Stable" if it doesn't change the order of similar items
  - .i.e. 7a 2a 7b → 2a 7a 7b

e)Insertion Sort

Work through the list of numbers starting from index 1
Store this 'number' in memory and compare with all index's before it (from index 0 which is the 'sorted end')
If this number is less than any point on that array, shift all elements up from this point and insert the original number in this point.
Do this until you've searched through every index

e)Insertion Sort time complexity
Worst case:
Numbers in inverse order, have to move each element to the front and compare with every element in the 'sorted' list
1 + 2 + 3.... n-1 → n(n+1)/2
Average case:
Numbers moved half way down the list so only have to compare to half the numbers in the 'sorted' list
n(n+1)/4
Best case:
Numbers already sorted
n-1 comparisons needed

f)Selection Sort
Go through the list comparing every item
Swap 1st element with smallest item, 2nd with second smallest etc...until you reach the end of the list

Always takes n(n-1)/2 comparisons but has less movement making it more attractive than insertion sort

g)Decisions trees
- These trees help map out different permutations of a result, .i.e. ABC could be sorted in 6 different ways: abc, acb, bac, bca, cab, cba.
- For sorting, the number of ways of sorting n objects is nPn or n!
- To have a binary tree with all these permutations on the leaves, its depth (d) must satisfy...
  - $2^d \geqslant n!$
  - d >= log2(n!)
  - n! ~ $\sqrt{2\pi \cdot n \cdot (\frac{n}{e})^n}$
  - The lowest time complexity works out as... ϴ= $n\,log2(n)$

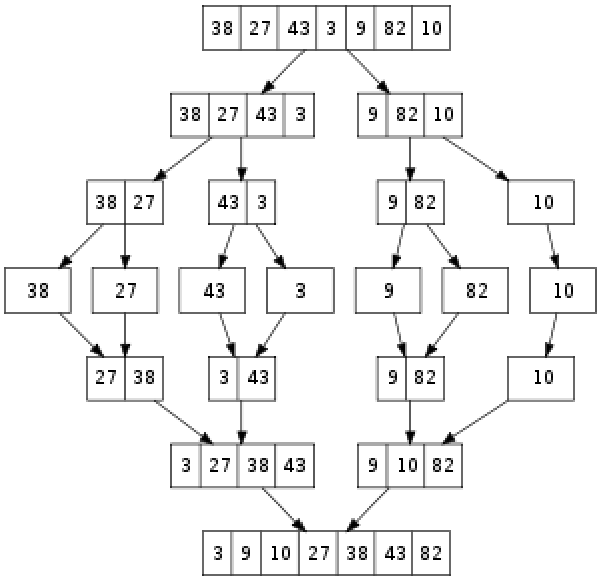
Advanced Sorting

a)Merge Sort

- *O*(nlog(n)) time complexity (nearly optimal)
- 1945 by John von Neumann

- If list is of size 0 or 1 then its already sorted
- Otherwise split into two rough parts and sort them using merge sort as two unordered lists
- Once ordered, merge back together

- Usually mixed with insertion sort so that when the array is small enough insertion sort is used instead of splitting further
- Stable!



b)Quick Sort

- Similar to Merge Sort but before splitting, select a pivot value (something like the median of the first, middle and end numbers from the list) and numbers bigger than this pivot go into one split and others go the other way. Repeat quick sort recursively on each half.
- Again insertion sort is used when the list is small enough as its quicker than endlessly splitting.
- Unstable :(

c)Usage

- Mergesort → objects
- Quicksort → numbers/primitives

d)Radix

- A form of bucket sort
- Numbers placed into 'buckets' (essentially an array of linked lists)
- The bucket is based on the number .i.e. 4 would go into the bucket at index 4.
- This is done on the least significant part of the number then worked up to the most significant, taking the numbers out of the 'buckets' in 'bucket' order
- Unfortunately number only

e)Other

- Heapsort
- Bucketsort
- Shell

Graphs

a)Graphs

- Many problems can be described and solved as a graph (even though they might not appear as one)
- Good for abstraction
- Can give a different view on a problem
- Unifies apparently different problems

b)Graph Theory

- Graphs can be directed (digraph)
- Can have connected or unconnected points
- Can have edges with weights or without
- Trees – Graph with no cycles
- Multi graph – graph where nodes can have more than one edge to another node
- Network – Mixture of different graph bits in one

c)Implementing

Can be implemented as a big matrix, where 1 means a connection and 0 means no connection

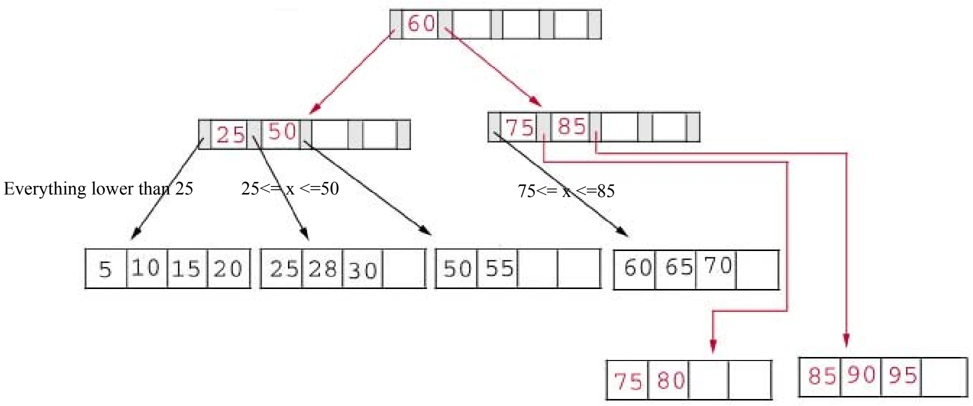
$$\begin{array}{ccccc}
 & a & b & c & d \\
a & 1 & 0 & 1 & 0 \\
b & 0 & 1 & 1 & 0 \\
c & 1 & 1 & 1 & 1 \\
d & 0 & 0 & 1 & 1
\end{array}$$

Or as a form of linked list for each node
Or as a node with a set of other nodes

General Knowledge

a)B-Trees

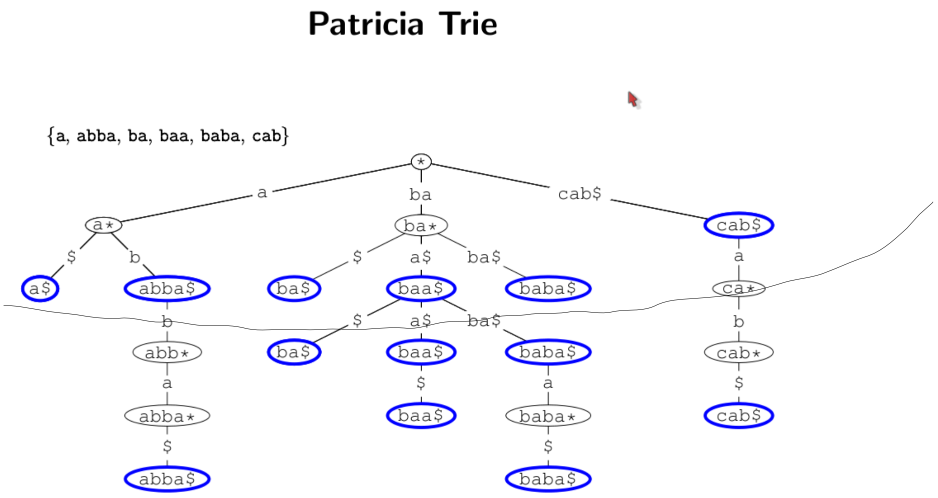
- Multi-way tree
- 'B' For balanced
- Used a lot in databases
- Useful is a lot of data is stored on disk
- Very wide but not very deep so fast actions
- Designed to keep data as close together as possible



- Operations can be hard especially when a level is full

b)Tries

- 'Digital Trees'
- Also Multi-way
- Often used for storing large sets of words
- Try to have a branch for every letter of an alphabet
- Patricia Trees compress non-branching sub trees into single nodes
- Often used in spell checking, predictive text



c)String Algorithms

- Very important
- Applications from text-editors to biological sequencing
- Brute Force can be simple but very expensive
- Speed ups through pre-processing and using suffix trees

d)Geometry

- Applications to machine learning and graphics
- Can be poor but research has found many good algorithms