Style (switch, for loops)

Monday, 23 July 2018 4:02 PM

COMP2521 Style

- Layout: consistent indentation is required as with COMP1511
- Brackets:
  - Can be omitted if control structure owns a single statement

```
if (true) do something;
```

  - Must be put after function header

```
int main (void)
{
    exit(EXIT_SUCCES);
}
```

- Can use all C control structure:
  - `if, switch, while, for, break, continue`
  - Put function start bracket on line after function header

```
if (true) {
    do something;
}
```

- Can use assignment statements in expressions, but should continue to avoid other kinds of side-effects
- Can use conditional expressions, but use `x = c ? e1 : e2` with care
- Functions may have multiple `return` statements, but use sparingly and primarily for error handling

`switch` Statements

switch encapsulates a common selections:

```
if (v == C1) {
   S1;
} else if (v == C2) {
   S2;
}
...
else if (v == Cn) {
   Sn;
}
else {
   Sn+1;
}
```

The multi-way if becomes:

```
switch (v) {
    case C1:
        S1; break;
    case C2;
        S2; break;
    ...
    case Cn:
        Sn; break;
    default:
        Sn+1;
    }
```

Note: `break` is critical; if it is not present, we will keep moving to the next cases

for loops

for encapsulates a common loop pattern:

```
initial;
while (condition) {
   do something;
   increment;
}
```

as

```
for (initial; condition; increment)
   do something;
```

Conditional expressions

```
x = c ? e1 : e2
```

Is equivalent to

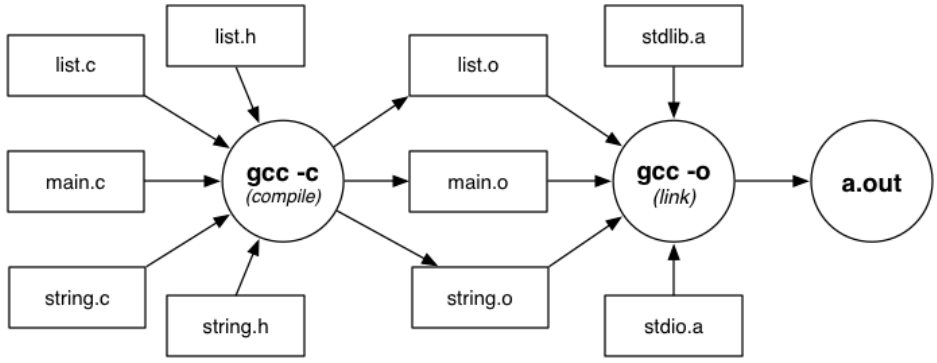
```
if (x == c) {
     e1;
else {
     e2;
}
```

Compilation and Makefiles

Thursday, 26 July 2018 9:13 AM

Compilers are programs that convert program source code to an executable form. The "executable" might be machine code or bytecode.

The GNU C compilers (**gcc**) applies source-to-source transformation (pre-processing), then compiles ***source code*** to produce ***object files***. Then links object files and ***libraries*** to produce ***executables***.



Compilation/linking with **gcc**:

```
gcc -c Stack.c
produces Stack.o, from Stack.c and Stack.h
gcc -c bracket.c
produces bracket.o, from bracket.c and Stack.h
gcc -o rbt bracket.o Stack.o
links bracket.o, Stack.o and libraries
producing executable program called rbt
```

Note that **stdio,** and **assert** are included implicitly.
gcc is a multi-purpose tool. It compiles (**-c**), links, makes executables (**-o**)

make/Makefiles

The compilation process is complex for large systems. It is ideal to compile what has changed since the last compile, but it is more practical to recompile everything to be sure.

The `make` command assists by allowing programmers to document ***dependencies*** in code. Based on these dependences we can have minimal re-compilation because make compiles only what is required.

make is driven by dependencies given in a Makefile. A dependency specifies

```
target : source1 source2 …
    commands to build target from sources
```

Rule: the `target` is rebuilt if any of the sources have changed

Example:

```
game : main.o graphics.o world.o
    gcc -o game main.o graphics.o world.o
main.o : main.c graphics.h world.h
    gcc -Wall -Werror -c main.c
graphics.o : graphics.c world.h
    gcc -Wall -Werror -c graphics.c
world.o : world.c
    gcc -Wall -Werror -c world.c
```

Things to note:

- A **`target`** (game, main.o, …) is on a newline
  - followed by a **`:`**

    - Then followed by the files that the target is depended on
- The **action** (gcc …) is always on a newline and **MUST** be indented with <TAB>

If make arguments are targets, it builds just those targets.

```
prompt$ make world.o
gcc -Wall -Werror -c world.c
```

If there are no arguments, it builds the first target in the Makefile

```
prompt$ make
gcc -Wall -Werror -c main.c
gcc -Wall -Werror -c graphics.c
gcc -Wall -Werror -c world.c
gcc -o game main.o graphics.o world.o
```

Analysis of Algorithms

Thursday, 26 July 2018 8:37 PM

Running Time

An algorithm is a step-by-step procedure for solving a problem in a finite amount of time. Most algorithms map input to output. Running time typically grows with input size, so the *average time* is often difficult to determine. Because of this we focus on the *worst case* running time. It is easier to analyse and crucial to many applications; finance, robotics, games,…

Empirical Analysis

1. Write program that implements an algorithm
2. Run program with inputs of varying size and composition
3. Measure the actual running time
4. Plot the results

Empirical Analysis has some limitations:

- It requires __ to implement the algorithm, which may be difficult
- The results may not be indicative of running time on other inputs
- Same hardware and operating systems must be used to compare two algorithms

Theoretical Analysis

Theoretical Analysis uses high-level description of algorithm instead of implementation ("pseudocode"). It characterises running time as a function of input size, n. It takes into account all possible inputs and allows us to evaluate the speed of an algorithm independent of the hardware/software environment.

Pseudocode

Pseudocode is more structured than English prose, but is less detailed than a program.
It is the preferred notation for describing algorithms, however, it hides program design issues.

Example: Find maximal element in an array

```
arrayMax(A):
|  Input  array A of n integers
|  Output maximum element of A
|
|  currentMax=A[0]
|  for all i=1..n-1 do
|  |  if A[i]>currentMax then
|  |     currentMax=A[i]
|  |  end if
|  end for
|  return currentMax
```

Control flow

- **if** … **then** … [**else**] … **end if**
- **while .. do** … **end while**
  repeat … **until**
  for [all][each] .. do … **end for**

Function declaration

- f(arguments):
  Input …
  Output …
  …

Expressions

- = assignment

- = equality testing
- n^2 superscripts and other mathematical formatting allowed
- swap A[i] and A[j] verbal descriptions of *simple* operations allowed

The Abstract RAM Model

RAM stands for Random Access Machine.

A CPU (central processing unit) has a potentially unbounded bank of memory cells. Each of which can hold an arbitrary number, or character. The memory cells are numbered, and accessing any one of them takes CPU time.

Primitive Operations

Primitive operations are basic computations performed by an algorithm. These operations are identifiable in pseudocode and are largely independent of the programming language. The exact definition of these operations are not important and they are assumed to take a constant amount of time in the RAM model.

Some examples of primitive operations include:

- Evaluating an expression
- Indexing into an array
- Calling/returning from a function.

Counting Primitive Operations

By inspecting the pseudocode we can determine the maximum number of primitive operations executed by an algorithm as a function of the input size.

Example:

```
arrayMax(A):
|  Input  array A of n integers
|  Output maximum element of A
|
|  currentMax=A[0]                1
|  for all i=1..n-1 do            n+(n-1)| n(checking), n-1 (assignment)
|  |  if A[i]>currentMax then     2(n-1) | retrieve index, then compare
|  |     currentMax=A[i]          n-1
|  |  end if
|  end for
|  return currentMax              1
                                   -------
                          Total    5n-2
```

Estimating Run Times

Define:

- a as the time taken by the fastest primitive operation
- b as the time taken by the slowest primitive operation

Let T(n) be the worst-case time of `arrayMax`. Then

$$a \cdot (5n - 2) \leq T(n) \leq b \cdot (5n - 2)$$

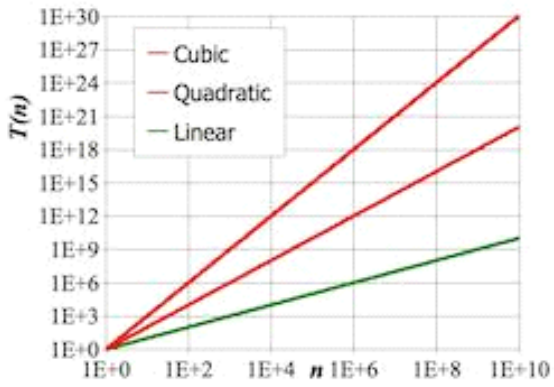
Hence, the running time $T(n)$ is bound by two linear functions

Seven commly encountered functions for algorithm analysis are:

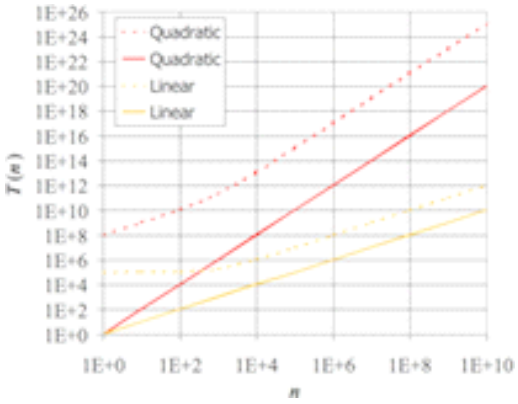
- Constant ≅ 1
- Logarithmic ≅ $\log n$
- Linear ≅ n
- N-Log-N ≅ $n \log n$
- Quadratic ≅ n^2
- Cubic ≅ n^3
- Exponential ≅ 2^n

- Quadratic ≅ n^2
- Cubic ≅ n^3
- Exponential ≅ 2^n

In a log-log chart, the slope of the line corresponds to the growth rate of the function.



The growth rate is not affected by constant factors or lower-order terms



Changing the hardware/software environment affects T(n) by a constant factor, but does not alter the growth rate of T(n). Hence the linear growth rate of the running time T(n) is an intrinsic property of algorithm `arrayMax`.

Big-Oh

Notation

Given functions f(n) and g(n), we say that

f(n) is O(g(n))

if there are positive constants c and n_0 such that

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

Big-Oh and Rate of Growth

Big-Oh notation gives an upper bound on the growth rate of a function. ***f(n) is O(g(n))*** means the growth rate of f(n) is no more than the growth rate of g(n). We use big-Oh to rank functions according to their rate of growth.

	f(n) is O(g(n))	g(n) is O(f(n))
g(n) grows faster	yes	no
f(n) grows faster	no	yes
same order of growth	yes	yes

Big-Oh Rules

1. If f(n) is a polynomial of degree d ⇒ f(n) is $O(n^d)$
   a. lower-order terms are ignored
   b. constant factors are ignored
2. Use the smallest possible class of functions
   a. say "2n is O(n)" instead of "2n is $O(n^2)$"
3. Use the simplest expression of the class
   a. say "3n + 5 is O(n)" instead of "3n + 5 is O(3n)"

Asymptotic Analysis of Algorithms

Asymptotic analysis of algorithms determines running time in big-Oh notation. It finds the worst-case number of primitive operations as a function of input size and express this function using big-Oh notation

Example:

- algorithm `arrayMax` executes at most 5n – 2 primitive operations
  ⇒ algorithm arrayMax *"runs in O(n) time"*

Constant factors and lower-order terms eventually dropped, so you can disregard them when counting primitive operations.

Example: Binary Search
The following recursive algorithm searches for a value in a **sorted** array:

```
search(v,a,lo,hi):
|  Input  value v
|         array a[lo..hi] of values
|  Output true if v in a[lo..hi]
|         false otherwise
|
|  mid=(lo+hi)/2
|  if lo>hi then return false
|  if a[mid]=v then
|     return true
|  else if a[mid]<v then
|     return search(v,a,mid+1,hi)
|  else
|     return search(v,a,lo,mid-1)
|  end if
```

Cost analysis:

- C_i = #calls to `search()` for array of length i
- for best case, $C_n = 1$
- for a[i..j], j<i (length=0)
  - $C_0 = 0$
- for a[i..j], i≤j (length=n)
  - $C_n = 1 + C_{n/2} \Rightarrow C_n = \log_2 n$

Thus, binary search is $O(\log_2 n)$ or simply $O(\log n)$.

Math Needed for Complexity Analysis

- Summations
- Logarithms
  - $\log_b (xy) = \log_b x + \log_b y$
  - $\log_b (x/y) = \log_b x - \log_b y$
  - $\log_b x^a = a \log_b x$
  - $\log_b a = \log_x a / \log_x b$
- Exponentials
  - $a^{(b+c)} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $a^b / a^c = a^{(b-c)}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c \cdot \log_a b}$
- Proof techniques
- Summation (addition of sequences of numbers)
- Basic probability (for average case analysis, randomised algorithms)

Relatives of Big-Oh

big-Omega

- f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$

big-Theta

- f(n) is Θ(g(n)) if there are constants c',c'' > 0 and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \quad \forall n \geq n_0$

- f(n) is O(g(n)) if f(n) is asymptotically *less than or equal* to g(n)
- f(n) is Ω(g(n)) if f(n) is asymptotically *greater than or equal* to g(n)
- f(n) is Θ(g(n)) if f(n) is asymptotically *equal* to g(n)

Complexity Classes

In Computer Science some problems have ***polynomial*** worst-case performance (e.g. n^2) and some have *exponential* worst-case performance (e.g. 2^n).

Problems have classes:

- *P* = problems for which an algorithm can compute answer in polynomial time
- *NP* = includes problems for which no *P* algorithm is known

Beware: NP stands for "Nondeterministic, Polynomial time (on a theoretical *Turing Machine*)".

Computer Science jargon for difficulty:

- tractable … have a polynomial-time algorithm (useful in practice)
- intractable … no tractable algorithm is known (feasible only for small *n*)
- non-computable … no algorithm can exist

Computational complexity theory deals with different degrees of intractability.

Generate and Test Algorithms

In scenarios where it is simple to test where a given state is a solution, it is easy to generate new states (preferably likely solutions), then ***generate and test*** strategy can be used.
It is necessary that states are generated systematically so that we are guaranteed to find a solution, or know that none exists. Some randomised algorithms do not require this (this will be covered more later in the course).

Simple example: checking whether an integer *n* is prime

- Generate/test all possible factors of *n*
- If none of them pass the test, then *n* is prime

Generation is straightforward:

- Produce a sequence of all numbers from 2 to *n-1*

Testing is also straightforward:

- Check whether a number divided *n* exactly

Function for primality checking:

```
isPrime(n):
|  Input  natural number n
|  Output true if n prime, false otherwise
|
|  for all i=2..n-1 do
|  |  if n mod i = 0 then
|  |     return false
|  |  end if
|  end for
|  return true
```

Complexity of `isPrime` is O(n).
Can be optimised by checking only numbers between 2 and $\lfloor\sqrt{n}\rfloor \Rightarrow O(\sqrt{n})$

Abstract Data Types

Monday, 30 July 2018 10:02 PM

Program Properties

We want our programs to be:

- **Correct**: produce required results for valid inputs
- **Reliable**: behave sensibly for invalid inputs/errors
- **Efficient**: give results quickly (even for large inputs)
- **Maintainable**: code is clear and well-structured

Guaranteeing correctness requires:

- A formal **statement of requirements** (pre/post-conditions)
- A formal **proof** that the program meets these requirements

Testing increases confidence that a program may be correct.
Trade-off: efficiency vs. clarity

Abstract Data Types

A **data type** is a set of ***values*** (atomic or structured values). It is a collection of ***operations*** are performed on those values.
An **abstract data type** is an approach to implementing data types. It separates ***interface*** from ***implementation***. The users of ADTs see only the interface, while builders of the ADT provide an implementation. Example: `FILE *`. We don't know what it looks like, but we don't need to know.

DTs, ADOs, ADTs, GADTs

We want to distinguish ...

- DT = (non-abstract) *d*ata *t*ype (e.g. C strings)
- ADO = *a*bstract *d*ata *o*bject
- ADT = *a*bstract *d*ata *t*ype (e.g. C files)
- GADT = *g*eneric (polymorphic) *a*bstract *d*ata *t*ype

ADTs ⇒ can have multiple instances (e.g. Set a, b, c;)
GADTs ⇒ multiple instances/types (e.g. Set<int> a; Set<char> b;)

Interface/Implementation

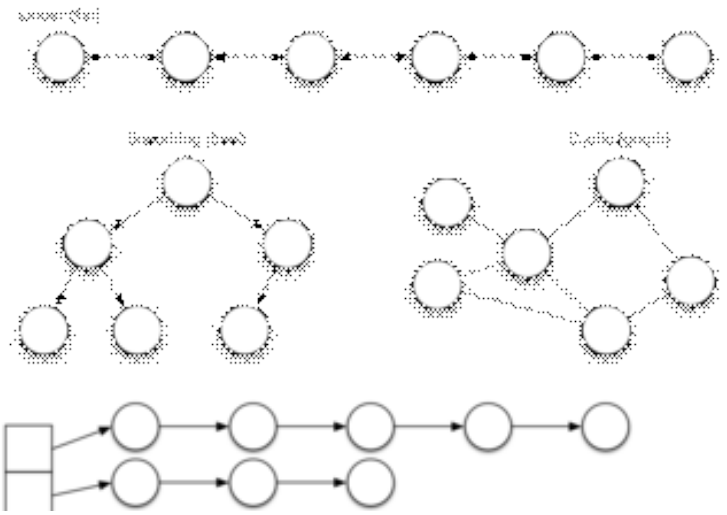
ADT ***interface*** provides a user-view of the data structure (e.g FILE *). It also provides function signatures (prototypes) for all operations, the semantics of operations (via documentation), and a contract between ADT and its clients.
ADT ***implementation*** gives a concrete definition of the data structures and the definition of functions for all operations.

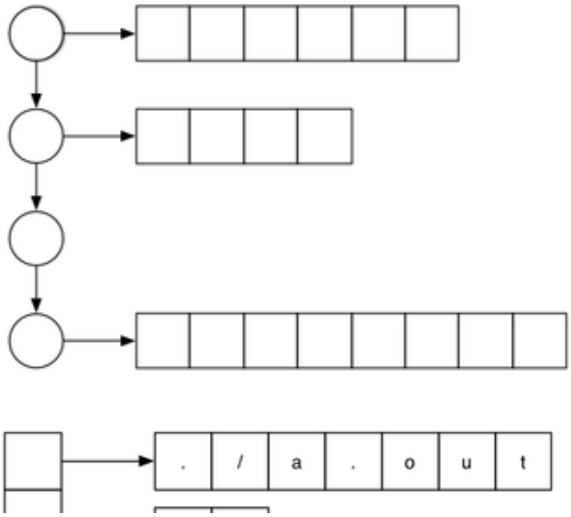
Collections

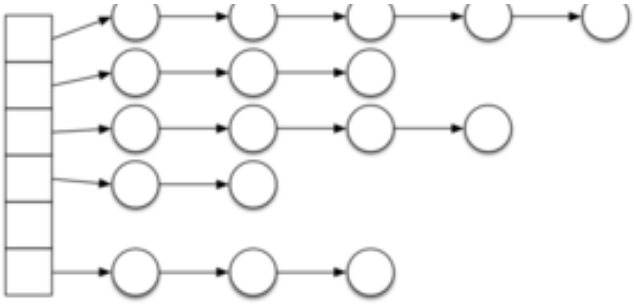
Many of the ADTs we deal with consist of a ***collection* of *items***, where each item may be a simple type or an ADT, and the items often have a ***key*** (to identify them). Collections may be categorised by:

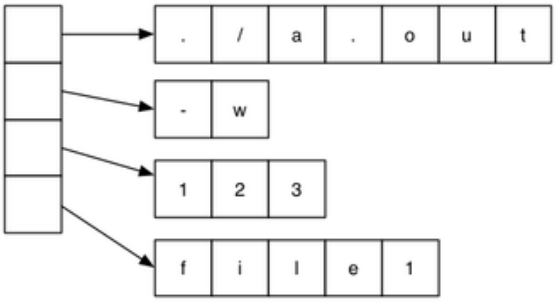
- Structure:
  - Linear (list), branching (tree), cyclic (graph)
- Usage:
  - Set, matrix, stack, queue, search-tree, dictionary

Collection structures:









Typical operations on collections include:

- ***create*** an empty collection
- ***insert*** one item into the collection
- ***remove*** one item from the collection
- ***find*** an item in the collection
- ***check*** properties of the collection (size,empty?)
- ***drop*** the entire collection
- ***display*** the collection

Example ADT: Sets of Integers

Set ADT

A set data type is a collection of unique integer values.
Some *book-keeping* operations are:

- `set newSet()` - create a new empty set
- `void dropSet(Set)` - free memory used by a set
- `void show(Set)` - display as {1, 2, 3, …}

Some assignment operations:

- `void readSet(FILE *, Set)` - read and insert set values
- `Set SetCopy(Set)` - make a copy of a set

Some data-type operations:

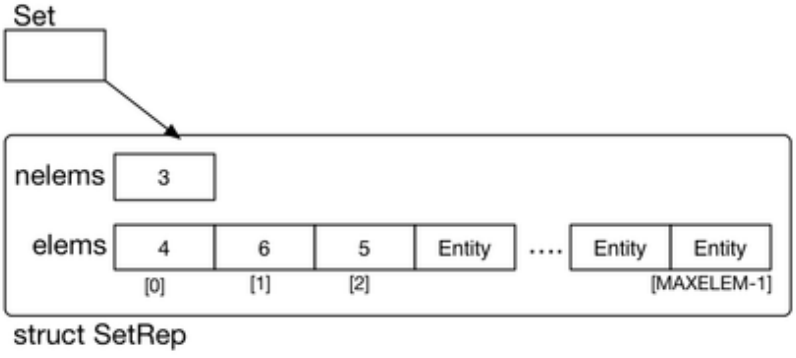
- `void SetInsert(Set,int)` - add number into set
- `void SetDelete(Set,int)` - remove number from set
- `int SetMember(Set,int)` - set membership test
- `Set SetUnion(Set,Set)` - union
- `Set SetIntersect(Set,Set)` - intersection
- `int SetCard(Set)` - cardinality (#elements)

Note: union and intersection return a newly-created Set

Pre- and post-conditions

Each Set operation has well-defined semantics.
These semantics should be expressed in detail via statements of what conditions need to hold at start of function and what will hold at end of function (assuming it is successful). These conditions *could* be implemented as `assert()`s in functions, dut only during the development/testing phase as `assert()` does not provide useful error-handling.

Sets as an Unsorted Array



Costs for set operations on unsorted array:

card: read from struct;	constant cost	*O(1)*
member: scan list from start;	linear cost	*O(n)*
insert: duplicate check, add at end;	linear cost	*O(n)*
delete: find, copy last into gap;	linear cost	*O(n)*
union: copy s1, insert each item from s2;	quadratic cost	*O(nm)*

intersect: scan for each item in s1;	quadratic cost	*O(nm)*

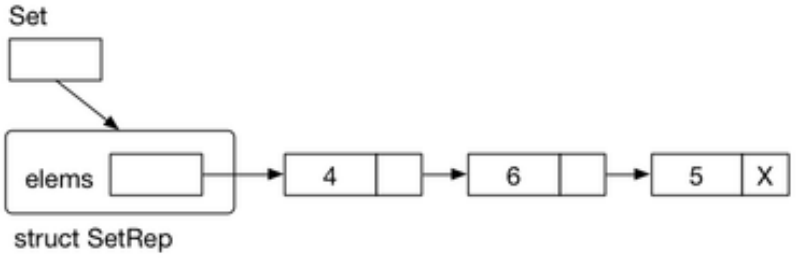
Assuming: s1 has *n* items, s2 has *m* items

Sets as a Sorted Array

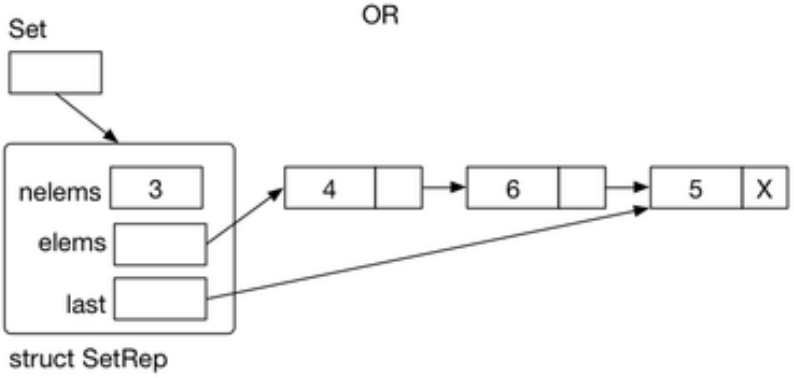
Costs for set operations on sorted array:

card: read from struct;	*O(1)*
member: binary search;	*O(log n)*
insert: find, shift up, insert;	*O(n)*
delete: find, shift down;	*O(n)*
union: merge = scan s1, scan s2;	*O(n)* (technically *O(n+m)*)
intersect: merge = scan s1, scan s2;	*O(n)* (technically *O(n+m)*)

Sets as a Linked List



OR



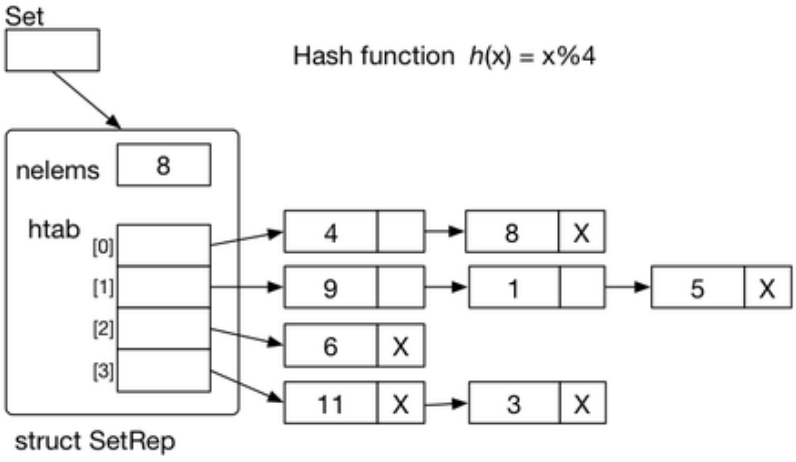
Costs for set operations on linked list:

insert: duplicate check, insert at head;	*O(n)*
delete: find, unlink;	*O(n)*
member: linear search;	*O(n)*
card: lookup;	*O(1)*
union: copy s1, insert each item from s2;	*O(nm)*
intersect: scan for each item in s1;	*O(nm)*

Assume *n* = size of *s1*, *m* = size of *s2*

If we don't have `nelems`, *card* becomes *O(n)*

Sets as Hash Tables



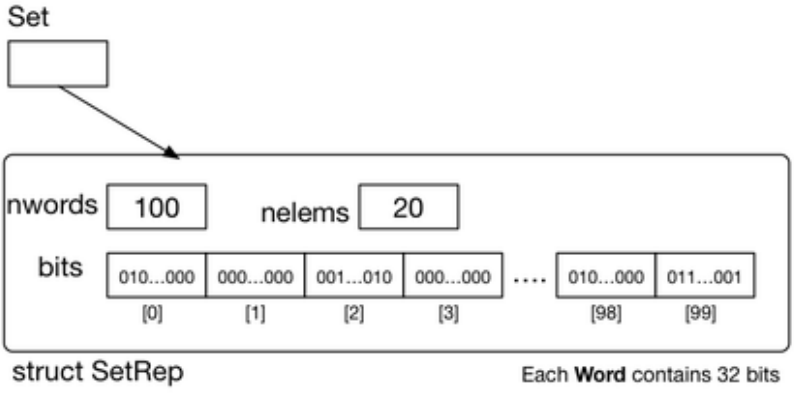
Hash function: *h*(value) = (value % TABSIZE).
Operations *insert*, *delete*, *member*

- all start by *h*(v), and then manipulate selected list

Cost for set operations via hash table:

- same *O(n)* behaviour as for single linked list
- if sorted: insert/delete = *O(n)*, ∈ = *O(n)*, ∪/∩ = *O(n+m)*
- if unsorted: insert = *O(1)*, delete = *O(n)*, ∈ = *O(n)*, ∪/∩ = *O(n.m)*
- BUT because hash table has *H* entries ...
  - has *H* lists for *n* elements (cf. 1 list for *n* elements)
  - each hash table list is, on average, *H* times shorter than single list
  - constants do not appear in *O(n)* analyses, but can be significant

Sets as Bit-strings



Restrict possible values that can be stored in the Set

- typically restricted to 0..N-1, (where N%32 == 0)
- represent each value by position in large array of bits
- insertion means set a bit to 1 (bit|1)
- deletion means set a bit to 0 (bit&0)
- bit position for value i is easy to compute

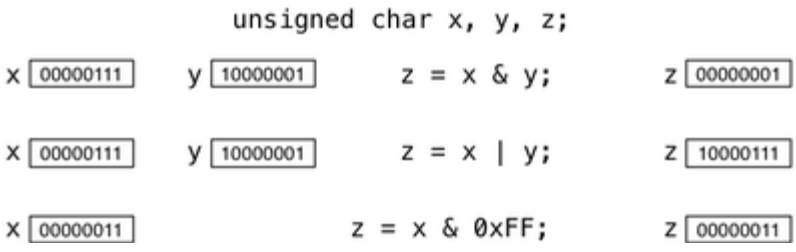
Representation of bit-list in C:

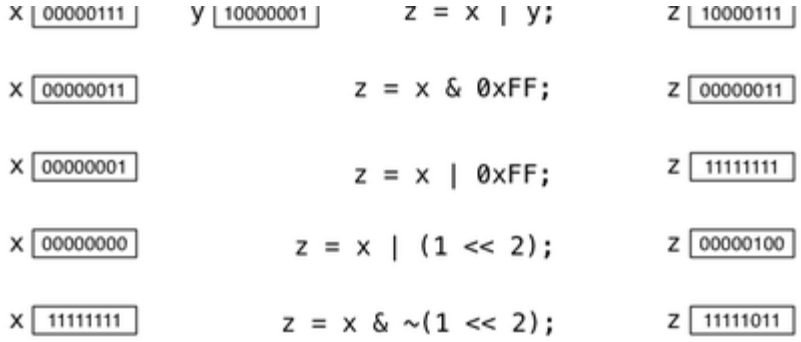
```
#define NWORDS ???
unsigned int bits[NWORDS];
```

For most common C implementations, gives us #bits = 32*NWORDS.
This gives us a set which can hold values in range 0..(#bits-1)

Setting and unsetting bits

We can set and unset bits by using the operators & and |

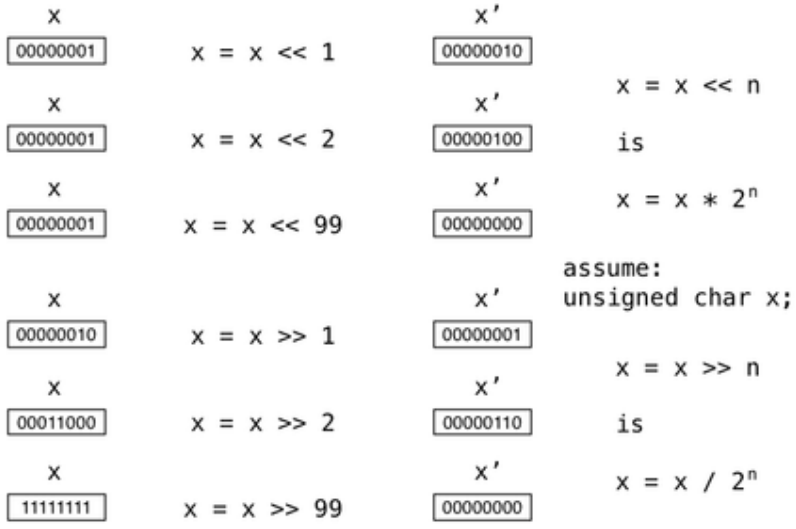




Note: `0xFF = 1111 1111`

The last two switch on/off bit 2

Find powers of 2 by bit-shifting (don't use `pow(..)` from `math.h`)



<u>Performance of Set Implementations</u>

Performance comparison:

Data Structure	insert	delete	member	∪, ∩	storage
unsorted array	*O(n)*	*O(n)*	*O(n)*	*O(n.m)*	*O(N)*
sorted array	*O(n)*	*O(n)*	*O(log_2n)*	*O(n+m)*	*O(N)*
unsorted linked list	*O(n)*	*O(n)*	*O(n)*	*O(n.m)*	*O(n)*
sorted linked list	*O(n)*	*O(n)*	*O(n)*	*O(n+m)*	*O(n)*
hash table (lists)	*O(n)*	*O(n)*	*O(n)*	*O(n+m)*	*O(n+H)*
bit-maps	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(N)*

n,m = #elems, *N* = max #elems, *H* = size of hash table

Notes on performance differences ...

- Assume *O(1)* card, because we keep a count of #elems
- For sorted array, insert cost is: cost of find + cost to interpolate
- Cost = *log_2n* tests + (avg)*n/2* moves ⇒ *O(n)*
- Two *O(n)* costs: linked list = *k.n*, hash table = *j.n* but j≪k (lists are shorter)
- If lists are sorted, use merge for ∪ and ∩ so *O(n+m)*
- Bit-map version restricts range of possible elements to *0..N-1*

If something in the ADT changes, the implementation will probably change.
e.g. If you change type from int to char, all the data types need to be changes as well.
And if you want to compare two variables:

- For int you would use '=='
- For char you would use 'strcmp()'

Generic ADTs in C

Monday, 6 August 2018 12:46 PM

Function Pointers

C can pass functions by passing a pointer to them. Function pointers are references to memory addresses of functions. They are pointer values and can be passed and assigned.
Function pointer variables/parameters are declared as:
`typeOfReturnValue (*fp)(typeOfArguments)`
e.g. `int (*fp)(int)`
In the example above, `fp` points to a function that returns an `int` and has one argument of type `int`.

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

int timesTwo(int x) {return x*2;}

int (*fp)(int);     <- declares the function pointer, it doesn't have a value yet

fp = &square;       //fp points to the square function

int n = (*fp)(10); //call the square function with input 10

fp = timesTwo;      //works without the &
                   //fp points to the timesTwo function

n = (*fp)(2);       //call the timesTwo function with input 2

n = fp(2);          //can also use normal function call
                   //notation
```

Higher-order Functions

Functions that get other functions as arguments, or return functions as a result are high-order functions.
Example: the function traverse() takes a list and a function pointer (fp) as an argument and applies the function to all nodes in the list.

```
void traverse (list ls, void (*fp) (list) ){
   list curr = ls;
   while(curr != NULL){
      // call function for the node
      fp(curr);
      curr = curr->next;
   }
}
```

Second argument is **fp**,
Pointer to a function like,
void functionName(list n)

The functions given don't have to traverse through the list anymore, instead they just perform the operations they need to do on the list as if working with a single node.

Example:

```
void printNode(list n){
    if(n != NULL){
        printf("%d->",n->data);
    }
}
```

```
void printGrade(list n){
    if(n != NULL){
        if(n->data >= 50){
            printf("Pass");
        }
        else {
            printf("Fail");
        }
    }
}
```

```
void traverse (list ls, void (*fp) (list));

//The second argument must have matching prototype

traverse(myList,   printNode);
traverse(myList,   printGrade);
```

Generic Types in C

Polymorphism refers to the ability of the same code to perform the same action on different types of data.
There are two primary types of polymorphism:

- ***Parametric polymorphism***: the code taks the type as a parameter, exither explicitly (as in C++ and Java) or implicitly (as in C)
- ***Subtype polymorphism***: subtype polymorphism is associated with inheritance hierarchies (as in Python)

Polymorphism in C

C provides a pointer to void (e.g. `void *p`), the programmer can then create generic data types by declaring values to be of type "`void *`". For example:

```
struct Node {
    void *value;
    struct Node *next;
}
```

The programmer can pass in type-specific functions (e.g. comparator functions) that take `void *`'s as parameters and that ***downcast*** the `void *`'s to the appropriate type before manipulating the data.

Example:

```c
#include <stdio.h>
#include <string.h>

// generic min function
void *min(void *element1, void *element2, int (*compare)(void *, void *)) {
    if (compare(element1, element2) < 0)
        return element1;
    else
        return element2;
}

// stringCompare downcasts its void * arguments to char * and then passes
// them to strcmp for comparison
int stringCompare(void *item1, void *item2) {
    return strcmp((char *)item1, (char *)item2);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
    printf("usage: min string1 string2\n");
    return 1;
    }

    // call min to compare the two string arguments and downcast the return
    // value to a char *
    char *minString = (char *)min(argv[1], argv[2], stringCompare);

    printf("min = %s\n", minString);
    return 0;
}
```

There is generic `min` function that computes and returns the minimum of two elements. The sample program compares two strings.

Advantages of generic types in C:

- One copy of code works with multiple objects
- The approach supports both generic data structures and generic algorithms

Disadvantages of generic types in C:

- Downcasting can be dangerous because run-time type checks are not performed in C
- Code will often look cluttered

Sorting

Monday, 6 August 2018 12:19 PM

Sorting involves arranging a collection of items in order based on some property of the item and using an ordering relation on that property.
Sorting occurs in many data contexts; arrays, linked lists (internal, in-memory), files (external, on-disk).

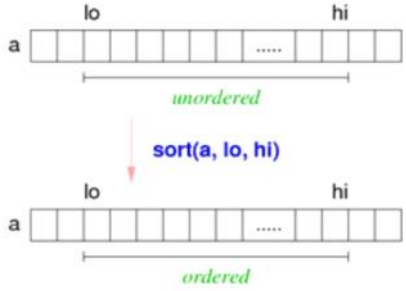
Different contexts generally require different approaches. Why is sorting useful?

- It speeds up subsequent searching
- Arranges data in a human-useful way (e.g. list of students in a tute ordered by family name or id)
- Provides intermediate step for advanced algorithms (e.g. duplicate detection/removal, many DBMS operations)

For now we will focus on sorting arrays of `Items` in ascending order.

The Sorting Problem

Arrange items in array slice `a[lo..hi]` into sorted order.



For `Item a[N]`, frequently `(lo == 0)` and `(hi == N-1)`

More formally:
The pre-conditions are:

- `lo, hi` are valid indexes, i.e. `0 ≤ lo < hi ≤ N-1`
- `a[lo..hi]` contains defined values of type Item

The post-conditions are:

- `a'[lo..hi]` contains same set (bag) of values
- for each `i` in `lo..hi-1, a'[i] ≤ a'[i+1]`

We sort arrays of `Items`, which could be ***simple values***, e.g. int, char, float, or ***structured values***, e.g. struct
Each Item contains a ***key***, which could be a ***simple value***, or a ***collection of values***.
The order of key values determines the order of the sort.
Duplicate key values are not precluded.

Properties of sorting algorithms: ***stable***, and ***adaptive***.
Stable sort:

- let `x = a[i], y = a[j], key(x) == key(y)`
- "precedes" means occurs earlier in the array (smaller index)
- *if x precedes y in a, then x precedes y in a'*

Adaptive:

- behaviour/performance of algorithm affected by data values
- i.e. best/average/worst case performance differs

In analysing sorting algorithms:

- *N* = number of items = `hi-lo+1`
- *C* = number of comparisons between items
- *S* = number of times items are swapped

Our aim is to minimise *C* and *S*.

Some cases to consider for initial order of items:

- Random order: `Items` in `a[lo..hi]` have no ordering
- Sorted order: `a'[lo] ≤ a'[lo+1] ≤ ... ≤ a'[hi]`
- Reverse order: `a'[lo] ≥ a'[lo+1] ≥ ... ≥ a'[hi]`

Comparison of Sorting Algorithms

A variety of sorting algorithms exists. Most are in-memory algorithms, but some also work with files. There are two major classes of algorithms: $O(n^2)$, and O(nlogn). $O(n^2)$ algorithms are acceptable if *n* is small (in the hundreds).

Implementing Sorting

A concrete framework for implementing sorting:

```
typedef int Item;
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define swap(A,B) {Item t; t = A; A = B; B = t;}
```

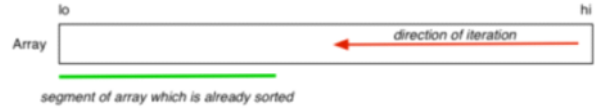
```
#define swil(A,B) {if (less(A,B)) swap(A,B);}
void sort(Item a[], int lo, int hi);
int isSorted(Item a[], int lo, int hi);
```

Sorts on Linux

- The `sort` command - sorts a file of text, and understands the fields in a line. It can sort alphabetically, numerically, in reverse and random order.
- The `qsort()` function - `qsort(void *a, int n, int size, int (*cmp)())`. It sorts any kind of array (n objects, each of size bytes) and requires the user to supply a comparison function (e.g. `strcmp()`). It sorts the list of items using the order given by `cmp()`

Sorting: Elementary Algorithms

To describe simple sorting, we use diagrams like this:



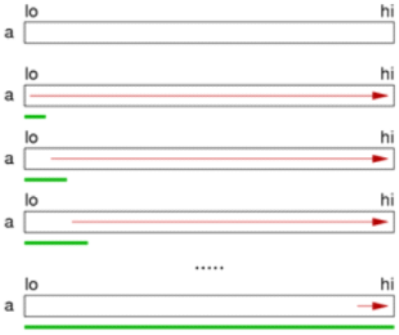
In these algorithms, a segment of the array is already sorted and each iteration makes more the array sorted.

Selection Sort

Selection sort is a ***simple***, ***non-adaptive*** method.
It finds the smallest element, then puts it in the first array slot. Then it finds the next smallest element, and puts it into the second array slot, and repeats this until all the elements are in the correct position.

"*Put in the x^{th} array slot*" is accomplished by swapping the value in the x^{th} with the x^{th} smallest value.

Each iteration improved the *sortedness* by one element.
Here is a diagrammatical state of the array after each iteration:




```
void selectionSort(int a[], int lo, int hi)
{
   int i, j, min;
   for (i = lo; i < hi-1; i++) {
      min = i;
      for (j = i+1; j <= hi; j++) {
         if (less(a[j],a[min])) min = j;
      }
      swap(a[i], a[min]);
   }
}
```

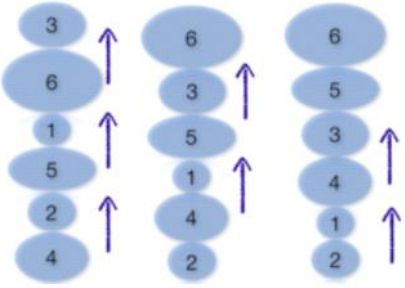
Cost analysis of selection sort (where n = `hi-lo+1`):

- On first pass, *n-1* comparisons, 1 swap
- On second pass, *n-2* comparisons, 1 swap
- ...
- On last pass, *1* comparison, 1 swap
- C = $(n-1)+(n-2)+...+1 = n*(n-1)/2 = (n^2-n)/2 \Rightarrow O(n^2)$
- S = *n-1*

The cost of selection sort is the same, regardless of the sortedness of original array.

Bubble Sort

Bubble sort is a simple ***stable*** and ***adaptive*** method.
It makes multiple passes from *N* to *i* (*i=0..N-1*), and on each pass, it swaps any out-of-order adjacent pairs. The elements move until they meet a smaller element and eventually the smallest element moves to i^{th} position. This repeats until all elements have moved to appropriate position. It stops, if there are no swaps during one pass (already sorted).



The state of the array after each iteration:

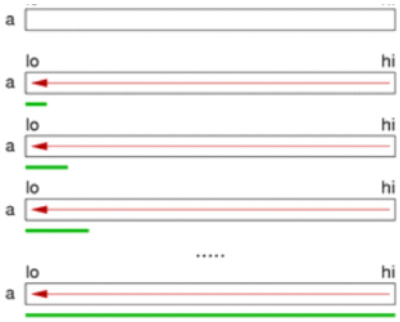
Sorting Example (courtesy Sedgewick):

```
S O R T E X A M P L E
A S O R T E X E M P L
A E S O R T E X L M P
A E E S O R T L X M P
A E E L S O R T M X P
A E E L M S O R T P X
A E E L M O S P R T X
A E E L M O P S R T X
A E E L M O P R S T X
...
A E E L M O P R S T X
```

```
void bubbleSort(int a[], int lo, int hi)
{
   int i, j, nswaps;
   for (i = lo; i < hi; i++) {
      nswaps = 0;
      for (j = hi; j > i; j--) {
         if (less(a[j], a[j-1])) {
            swap(a[j], a[j-1]);
            nswaps++;
         }
      }
      if (nswaps == 0) break;
   }
}
```

Cost analysis of bubble sort (where n = `hi-lo+1`):

- cost for i^{th} iteration:




```
    }
}
```

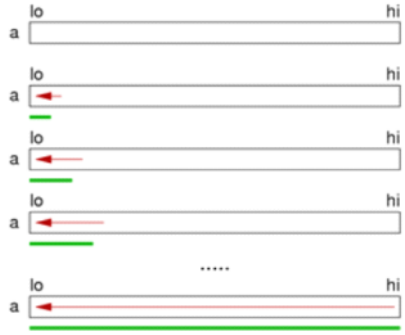
Cost analysis of bubble sort (where n = hi-lo+1):

- cost for i^{th} iteration:
  - *n-i* comparisons, ?? swaps
  - *S* depends on "sortedness", best=0, worst=*n-i*
- how many iterations? depends on data *orderedness*
  - best case: 1 iteration, worst case: *n-1* iterations
- $Cost_{best} = n$ (data already sorted)
- $Cost_{worst} = n-1 + ... + 1$ (reverse sorted)
- Complexity is thus $O(n^2)$

Insertion Sort

Insertion sort is a simple ***adaptive*** method.
It takes the first element and treats it a sorted array (of length 1).
Then it takes the next element and inserts it into the sorted part of array so that order is preserved. This increases the length of sorted part by one. We repeat this until whole array is sorted.

The state of the array after each iteration:



A simple implementation of insertion sort:

```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = lo+1; i <= hi; i++) {
      val = a[i];
      for (j = i; j > lo; j--) {
         if (!less(val,a[j-1])) break;
         a[j] = a[j-1];
      }
      a[j] = val;
   }
}
```

A better implementation of insertion sort:

```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = hi; i > lo; i--)
      swil(a[i-1], a[i]);
   for (i = lo+2; i <= hi; i++) {
      val = a[i];
      for (j = i; less(val,a[j-1]); j--) {
         a[j] = a[j-1];
      }
      a[j] = val;
   }
}
```

The use of sentinel values reduces the tests needed.

How the above algorithm works:

```
S O R T E X A M P L E
A S O R T E X E M P L
A S O R T E X E M P L
A O S R T E X E M P L
A O R S T E X E M P L
A O R S T E X E M P L
A E O R S T X E M P L
A E O R S T X E M P L
A E E O R S T X M P L
A E E M O R S T X P L
A E E M O P R S T X L
A E E L M O P R S T X
```

Complexity analysis of insertion sort

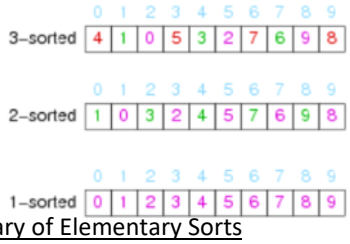
- cost for inserting element into sorted list of length *i*
  - *C*=??, depends on "sortedness", best=1, worst=*i*
  - *S*=??, don't swap, just shift, but do *C-1* shifts
- always have *N* iterations
- $Cost_{best} = 1 + 1 + ... + 1$ (already sorted)
- $Cost_{worst} = 1 + 2 + ... + N = N*(N+1)/2$ (reverse sorted)
- Complexity is thus $O(N^2)$

Shellsort

Insertion sort is based on exchanges that only improve adjacent items. It is already improved above by using moves rather than swaps. These l*ong distance* moves may be more efficient.

Shellsort is an improvement of insertion sort.
An array is *h*-sorted if by taking every *h*'th element we get a sorted array. An *h*-sorted array is made up of *n/h* interleaved sorted arrays. Shellsort makes the *h*-sort array for progressively smaller, ending with 1-sorted.

Example *h*-sorted arrays:




```
void shellSort(int a[], int lo, int hi)
{
   int hvals[8] = {701, 301, 132, 57, 23, 10, 4, 1};
   int g, h, start, i, j, val;
   for (g = 0; g < 8; g++) {
      h = hvals[g];
      start = lo + h;
      for (i = start; i < hi; i++) {
         val = a[i];
         for (j = i; j >= start && less(val,a[j-h]); j -= h)
            move(a, j, j-h);
         a[j] = val;
      }
   }
}
```

Effective sequences of *h* values have been determined empirically.
E.g. $h_{i+i} = 3h_i+1$ ... 1093, 364, 121, 40, 13, 4, 1
The efficiency of Shellsort depends on the sequence of *h* values. Surprisingly, Shellsort has not yet been fully analysed. The above sequence has been shown to be $O(n^{3/2})$, while others have found sequences which are $O(n^{4/3})$

Summary of Elementary Sorts

Comparison of sorting algorithms (online)

	No.	Of	compares	No.	Of	Swaps	No.	Of	Moves

	min	avg	max	min	avg	max	min	avg	max
Selection sort	n^2	n^2	n^2	n	n	n	.	.	.
Bubble sort	n	n^2	n^2	0	n^2	n^2	.	.	.
Insertion sort	n	n^2	n^2	.	.	.	n	n^2	n^2
Shell sort	n	$n^{4/3}$	$n^{4/3}$	.	.	.	1	$n^{4/3}$	$n^{4/3}$

Which is best?
It depends on:

- the cost of compare vs swap vs move for Items
- the likelihood of average vs worst case

Sorting Linked Lists

Selection sort on linked lists

- L = original list, S = sorted list (initially empty)
- find largest value V in L; unlink it
- link V node at front of S

Bubble sort on linked lists

- traverse list: if current > next, swap node values
- repeat until no swaps required in one traversal

Selection sort on linked lists

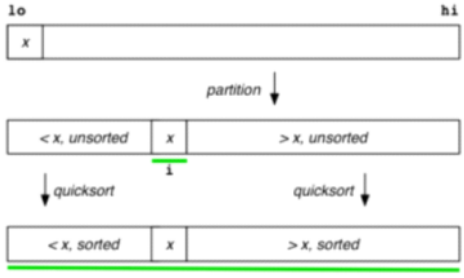
- L = original list, S = sorted list (initially empty)
- scan list L from start to finish
- insert each item into S in order

Sorting: Better O(nlogn) Algorithms

Quicksort

Quicksort chooses an item to be a ***pivot***, then re-arranges (partitions) the array so that all the elements to the left of the pivot are smaller than the pivot, and all the elements on the left of the pivot are greater than the pivot. It recursively sorts each of the partitions.

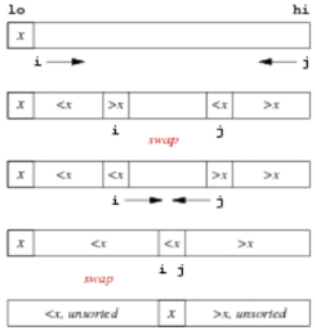
The phases of quicksort:



An elegant recursive solution:

```
void quicksort(Item a[], int lo, int hi)
{
   int i;
   if (hi <= lo) return;
   i = partition(a, lo, hi);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

The partitioning phase:



Partition implementation:

```
int partition(Item a[], int lo, int hi)
{
   Item v = a[lo];
   int  i = lo+1, j = hi;
   for (;;) {
```

Quicksort Improvements

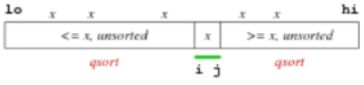
Our choice of pivot can have a significant effect our the algorithm. Always choosing the largest/smallest value gives us our worst case performance, so we try to find the *intermediate* value by ***median-of-three***.

```
void medianOfThree(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   if (less(a[hi],a[mid])) swap(a, mid, hi);
   if (less(a[mid],a[lo])) swap(a, lo, mid);


   swap(a, mid, lo+1); swap(a, lo, mid);
}
void quicksort(Item a[], int lo, int hi)
{
   int i;
   if (hi-lo < Threshhold) { ... return; }
   medianOfThree(a, lo, hi);
   i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

Another source of inefficiency is pushing recursion down to very small partitions. There is little benefit from partitioning when the size is less than 5.
Solution: handle small partition differently. For example, switch to insertion sort on small partition, or don't sort yet; use post-quicksort insertion sort.

```
void quicksort(Item a[], int lo, int hi)
{
   int i;
   if (hi-lo < Threshhold) {
      insertionSort(a, lo, hi);
      return;
   }
   medianOfThree(a, lo, hi);
   i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

If the array contains many duplicate keys, standard partitioning does not exploit this.



We can improve performance via three-way partitioning.

```
{
   Item v = a[lo];
   int  i = lo+1, j = hi;
   for (;;) {
      while (less(a[i],v) && i < j) i++;
      while (less(v,a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
   }
   j = less(a[i],v) ? i : i-1;
   swap(a,lo,j);
   return j;
}
```

Quicksort Performance

Best case: *O(nlogn)* comparisons

The choice of pivot gives two equal-sized partitions, and the same thing happens at every recursive level. Each *level* requires approximately *n* comparisons, halving at each level -> $log_2 n$ levels

Worst case: $O(n^2)$ comparisons

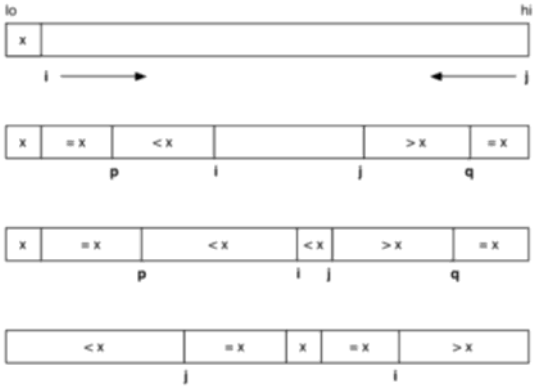
We always choose lowest/highest value for our pivot, so our partitions are size *1* and *n-1*. Each "level" requires approximately *n* comparisons, partitioning to *1* and *n-1* -> *n* levels

Quicksort can be implemented using an explicit stack:

```
void quicksortStack (Item a[], int lo, int hi)
{
   int i;  Stack s = newStack();
   StackPush(s,hi); StackPush(s,lo);
   while (!StackEmpty(s)) {
      lo = StackPop(s);
      hi = StackPop(s);
      if (hi > lo) {
         i = partition (a,lo,hi);
         StackPush(s,hi); StackPush(s,i+1);
         StackPush(s,i-1); StackPush(s,lo);
      }
   }
}
```

We can improve performance via three-way partitioning.

This is the Bently/McIlroy approach to three-way partitioning:

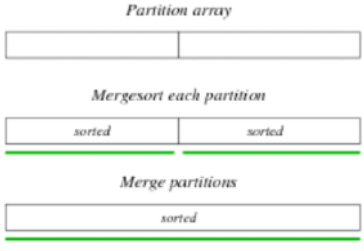


Mergesort

The basic idea for ***mergesort*** is that you split it array into two equal-sized partitions and recursively sort each of the partitions. Then you merge the two partitions into a new sorted array and copy it back to the original array.

The basic idea behind ***merging*** is that you copy elements from the inputs one at a time, and give preference to the smaller of the two. When one is exhausted, you can copy the rest of the other.

The phases of mergesort look like this:



An example of a ***mergesort*** function:

```
typedef int Item

int nums[10] =
{32,45,17,22,94,78,64,25,55,42};
mergesort(nums, 0, 9);

void mergesort(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
```

Mergesort Performance

Best case: *O(nlogn)* comparisons

We split the array into equal-sized partitions, and the same happens at every recursive level. Each "*level*" requires $\leq$ *N* comparisons halving at each level $\Rightarrow log_2 N$ levels.

Worst case: *O(nlogn)* comparisons

The partitions are exactly interleaved and we need to compare all the way to end of partitions.

A disadvantage over quicksort is that mergesort needs extra storage *O(n)*.

Non-recursive Mergesort

A non-recursive mergesort does not require a stack, instead the partition boundaries can be computed iteratively.

Bottom-up mergesort:

On each pass, the array contains sorted *runs* of length *m*.

At the start, we treat as *N* sorted runs of length 1.

The **first** pass merges adjacent elements into runs of length 2

The **second** pass merges adjacent 2-runs into runs of length 4

We continue until a single sorted run of length *N*

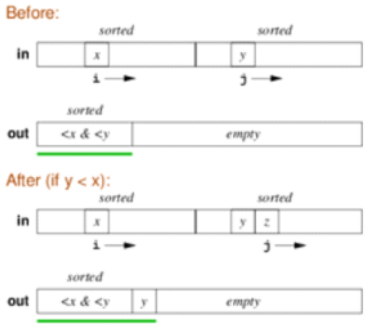
This approach is used for sorting disk files.

Bottom-up mergesort for arrays:

```
#define min(A,B) ((A < B) ? A : B)
void mergesort(Item a[], int lo, int hi)
{
   int i, m;
   int end;
   for (m = 1; m <= lo-hi; m = 2*m) {
      for (i = lo; i <= hi-m; i += 2*m) {
         end = min(i+2*m-1, hi);
         merge(a, i, i+m-1, end);
      }
   }
```

```
   int mid = (lo+hi)/2;
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
   merge(a, lo, mid, hi);
}
```

One step in the merging process looks like this:



Implementation of ***merge***:

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

   i = lo; j = mid+1; k = 0;
   // scan both segments, copying to tmp
   while (i <= mid && j <= hi) {
     if (less(a[i],a[j]))
        tmp[k++] = a[i++];
     else
        tmp[k++] = a[j++];
   }
   // copy items from unfinished segment
   while (i <= mid) tmp[k++] = a[i++];
   while (j <= hi) tmp[k++] = a[j++];

   // copy tmp back to main array
   for (i = lo, k = 0; i <= hi; i++, k++)
      a[i] = tmp[k];
   free(tmp);
}
```

```
         end = min(i+2*m-1, hi);
         merge(a, i, i+m-1, end);
      }
   }
}
```

Mergesort Variation

The previous methods require a temporary array. This provides a useful abstraction (`merge(a,lo,mid,hi)`) but requires `malloc()`/`free()` for each merge and copying data to/from temporary array.

An alternative approach would be to pass two arrays around; a ***source*** and a ***destination***. We switch array roles at each recursive call but this still requires one `malloc()`/`free()`.

Merge sort with source/destination arrays:

```
void mergeSort(Item a[], int lo, int hi)
{
   int i;
   Item *aux = malloc((hi+1)*sizeof(Item));
   for (i = lo; i <= hi; i++) aux[i] = a[i];
   doMergeSort(a, aux, lo, hi);
   free(aux);
}
void doMergeSort(Item a[], Item b[], int lo, int hi)
{
   if (lo >= hi) return;
   int mid = (lo+hi)/2;
   doMergeSort(b, a, lo, mid);
   doMergeSort(b, a, mid+1, hi);
   merge(b+lo, mid-lo+1, b+mid+1, hi-mid, a+lo);
}
// merge arrays a[] and b[] into c[]
// aN = size of a[], bN = size of b[]
void merge(Item a[], int aN, Item b[], int bN, Item c[])
{
   int i; // index into a[]
   int j; // index into b[]
   int k; // index into c[]
   for (i = j = k = 0; k < aN+bN; k++) {
      if (i == aN)
         c[k] = b[j++];
      else if (j == bN)
         c[k] = a[i++];
      else if (less(a[i],b[j]))
         c[k] = a[i++];
      else
         c[k] = b[j++];
   }
}
```

Summary of Sort Methods

We sort a collection of *N* items in ascending order.

Elementary sorts: $O(n^2)$ comparisons

- selection sort, insertion sort, bubble sort

Advanced sorts: *O(nlogn)* comparisons

- quicksort, merge sort, heap sort (priority queue)

Most are intended for use in-memory (random access data structure).

Merge sort adapts well for use as disk-based sort.

Other properties of sort algorithms: stability, adaptive

	Stability	**Adaptive**
Selection sort:	Stability depends on implementation	Not adaptive
Bubble sort:	Is stable if items don't move past same-key items	Adaptive if it terminates when no swaps
Insertion sort:	Stability depends on implementation of insertion	Adaptive if it stops scan when position is found
Quicksort:	Easy to make stable on lists; difficult on arrays	Can be adaptive depending on implementation
Merge sort:	Is stable if merge operation is stable	Can be made adaptive (but above version is not)

HeapSort

Will be discussing later in the course, after discussing a "heap" tree structure.

Sorting Lower Bound

Many popular sorting algorithms "compare" pairs of keys (objects) to sort an input sequence.

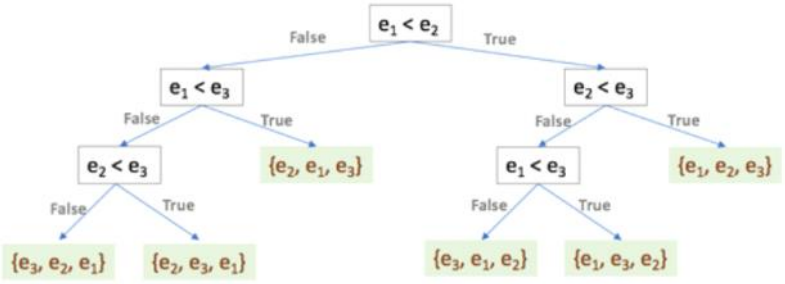
For example: selection-sort, insertion-sort, bubble-sort, merge-sort, quick-sort, etc.

Lower Bound: Any comparison-based sorting algorithm **must take *Ω (nlogn)*** time to sort *n* elements in the worst case.

Given n elements (no duplicates),

- there are *n!* possible permutation sequences
- one of these possible sequences is a sorted sequence
- each comparison reduces number of possible sequences to be considered

Decision Tree for input with three elements $\{e_1, e_2, e_3\}$



For a given input,

- the algorithm follows a path from the root to a leaf
- requires one comparison at each level
- there are *n!* leaves for *n* elements (e.g. *3!* leaves for 3 elements)
- height of such tree is at least *log2(n!)*, so number of comparisons required is at least *log2(n!)*

$log_2(n!) = log_2(1) + log_2(2) + ... + log_2(n/2) + ... + log_2(n-1) + log_2(n)$
$log_2(n!) >= log_2(n/2) + ... + log_2(n-1) + log_2(n)$
$log_2(n!) >= (n/2)log_2(n/2)$
$log_2(n!) = \Omega(n \log_2 n)$

Therefore, for any comparison-based sorting algorithm, the lower bound is $\Omega(n \log_2 n)$.

Non-Comparative Sorting

Radix Sort

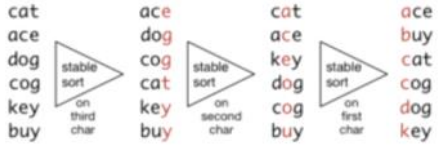
Radix sort is a ***non-comparative*** sorting algorithm.

The basic idea is that we represent the key as a ***tuple*** (k_1, k_2, ..., k_m). For example, represent 372 as (3, 7, 2) and represent sydney as (s, y, d, n, e, y). There are finite and normally few possible values of k_i. For example the numeric *0 to 9* and alpha-numeric *0 to 9* and *a to z*.

The sorting algorithm involves a ***stable*** sort on k_m, then a ***stable*** sort on $k_{(m-1)}$, and we continue this until k_1.

The time complexity of a stable sort like bucket/pigeonhole sort run in time O(*n*). The radix sort runs in time O(*mn*), where *m* is the number of sub-keys (k_i in the above tuple) and *n* is the number of items we are sorting.
Radix sort performs better (for sufficiently large n) than the best comparison-based sorting algorithms.

An example of radix sort:



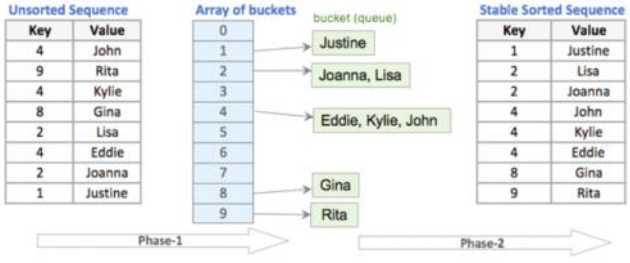
Bucket/Pigeonhole Sort

The basic idea behind bucket/pigeonhole sort is that we have a finite and normally as few possible values of *keys*. For example, the numeric *0 to 9*, the weekdays *Monday to Sunday*, and the months *January to December*. Each key value maps to an index into the array of buckets/pigeonholes. There is one bucket/pigeonhole per key value.

The sorting algorithm has two phases.
Phase 1: we move each entry from the input sequence to the corresponding bucket (say queue) in the array of buckets.
Phase 2: we move the entries of each bucket in the required order to the end of the output sequence.

The time complexity of bucket sort is O(*n*), assuming the number of buckets (keys) is not large.

An example of bucket/pigeonhole sort:



External Sorting

External Mergesort

Previous sorts all assume there is an efficient random access to `Items`, which suggests that data is in arrays in memory. This limits sortable data to what fits in memory.

When data is in disk files, random access is inefficient (files are sequential access), but the max data size is far less constrained. Because mergesort make multiple sequential passes, it adapts well as a sorting approach for files.

The basic idea behind external mergesort is that we

- have two files, A and B, which alternate as input/output
- scan input, sorting adjacent pairs, write to output
- scan input, merging pairs to sorted runs of length 4
- scan input, merging pairs to sorted runs of length 8
- repeat until entire file is sorted

The number of iteration needed is double the scan length until it is

A rough file merging algorithm:

```
fileMerge(inFile, outFile, runLength, N)
{
   inf1 = open inFile for reading
   inf2 = open inFile for reading
   outf = open outFile for writing
   in1 = 0; in2 = runLength
   while (in1 < N) {
      seek to position in1 in inf1
      end1 = in1+runLength
      it1 = getItem(inf1)
      seek to position in2 in inf2
      end2 = in2+runLength
      it2 = getItem(inf2)
      while (in1 < end1 && in2 < end2) {
         if (less(it1,it2)) {
            write it1 to outf
            it1 = getItem(inf1); in1++
         }
```

- scan input, merging pairs to sorted runs of length 4
- scan input, merging pairs to sorted runs of length 8
- repeat until entire file is sorted

The number of iteration needed is double the scan length until it is greater than the file size (≥ N `Items`).

The state of the output file after each iteration:

Naïve external mergesort algorithm in invalid C:

```
Input: stdin, containing N Items
Output: stream of Items on stdout
copy stdin file to A
runLength = 1
iter = 0
while (runLength < N) {
   if (iter % 2 == 0)
      inFile = A, outFile = B
   else
      inFile = B, outFile = A
   fileMerge(inFile, outFile, runLength, N)
   iter++;
   runLength *= 2;
}
copy outfile to stdout
```

```
         if (less(it1,it2)) {
            write it1 to outf
            it1 = getItem(inf1); in1++
         }
         else {
            write it2 to outf
            it2 = getItem(inf2); in2++
         }
      }
      while (in1 < end1) {
         write it1 to outf
         it1 = getItem(inf1); in1++
      }
      while (in2 < end2) {
         write it1 to outf
         it1 = getItem(inf1); in1++
      }
      in1 += runLength; in2 += runLength;
   }

}
```

External Mergesort Cost Analysis

Critical operation in external mergesort are:
Reading an Item, and writing an Item

- each pass = *N* reads + *N* writes
- #iterations = $\lceil \log_2 N \rceil$ (at most $\log_2 N$)
- Cost = $2N \lceil \log_2 N \rceil = O(N \log N)$

This works on any file that can be stored three times in the file system.
This approach is used by the Unix/Linux sort command.

While this is an *O(NlogN)* algorithm, each iteration reads and write the entire data file.
We can reduce iterations via an initial in-memory sorting pass:

- reading data in chunks (e.g. 512 Items at a time)
- sort each chunk in memory using e.g. QuickSort
- write each chunk out after sorting

Mergesort algorithm then starts with chunk-sorted file.
If chunks are size 2^k, need *k* less iterations.

Software Development Process

Thursday, 16 August 2018 8:54 PM

A reminder of how software development runs:

- Specification - via requirements analysis
- Design - data structures, algorithms
- Implementation - C code
- Testing, debugging - code analysis
- User testing - if has a user interface
- Performance tuning (if required)

We typically iterate over the implementation/testing phases.

In each stage there are tools available to assist us:

Specification	English, formal languages)
Design	Your brain ... and CS knowledge
Implementation	Editors, IDEs, compilers
Testing	Testing frameworks, (e.g. `Check`)
Debugging	Debuggers (e.g. `gdb`)
User testing	Usability testing methods
Performance tuning	Profilers (e.g. `prof`)

For testing, `assert()` and (even) `printf()` are also useful.

Testing

Testing is a systematic process of determining whether a program has mistakes (bugs) in it and handles bad inputs "reasonably".

Testing requires:

- The program specification, or detailed requirements
- An executable version of the program
- Sample input data and corresponding output data (or a tool that applies validation rules to the output)

Testing happens at different stages in development:

- ***Unit tests*** on behaviour of components
- ***System tests*** on overall input/output behaviour
- ***System integration tests*** on interaction of components
- ***User acceptance tests*** to find out what they don't like

A useful approach for unit testing is to

- start from lowest level functions (those, which don't call other functions)
- test each function as it is completed (and "tick it off")
- use only tested functions in testing higher-level functions

Testing alone cannot establish that program is *correct*. This is because, to show that a program is correct, we need to show that ***for all possible inputs, it produces the correct output***.
Even small programs have to many possible inputs. We can only feasibly test a small subset of possible inputs.

Instead of showing our program is correct, testing *increases our confidence* that our program is OK.
Well-chosen tests can *significantly* increase our confidence.

Testing Strategy

Testing only increases our confidence that the program works.
Unfortunately, we tend to make the **big** assumption:

"*It works for my test data, so it'll work for all data*"

The only way to be absolutely sure that this is true is to feed in every possible valid input value, and if each input produces the expected output, it's correct. This is called ***exhaustive testing***.
Exhaustive testing is not possible in practice. e.g. We can't test all int arrays of size 100.

A more realistic way of testing would be to:

- determine classes/partitions of input data set
- choose representative input values from each class
- determine expected output for each input
- execute program using all representative inputs

If, for each input, the program gives the expected output, then we have increased our confidence that the program works OK, but we have ***not*** demonstrated that the program is correct.

Developing Test Cases

Examples:

Kind of Problem	**Partitions of input values for testing**
Numeric	+value, -value, zero
Text	0, 1, 2, many text elements; lines of zero and huge length
List	0, 1, 2, many list items
Ordering	ordered, reverse ordered, random order, duplicates
Sorting	same as for ordering

Years of (*painful*) experience have yielded some common bugs:

- iterating one time ***too many*** or one time ***too few*** through a loop
- using ***< rather than <=***, or *vice versa*
- forgetting to handle the ***null/empty/zero case***
- ***assuming*** that other parts of the program supply ***valid data***
- ***assuming*** that ***library functions*** like malloc (see later) ***always succeed***
- etc. etc. etc.

Choose test cases to ensure that all such bugs will be exercised.
This requires you to understand the "***limit points***" of your program.

Making Your Program Fail

Some techniques that you can use to exercise potential bugs:

- make array sizes tiny (#define SIZE 2)
- write a special malloc that fails at random
- initialize data structures with a value other than zero
- test all possible combinations of parameter settings
- supply empty input (empty file, no argv values)
- test on different compilers/machines/operating systems

Summary: Testing Strategies

"Big Bang" approach - you write the entire program, then you design and run some test cases. This is generally a bad idea.
"As You Go" Testing - you write a small piece of code, then you test it. Then you integrate it with other tested pieces and test again. Repeat this iteratively until the entire program has been constructed.
Regression Testing - re-run all testing after any changes to the system

Debugging

Debugging is the process of removing errors from software.

It is required when the observed output ! expected output.

It is typically not due to the software being full of errors, rather it is usually due to a small incorrect fragment of code. A *bug* is a code fragment that does not satisfy its specification.
Consequences of bugs:

- compiler gives syntax/semantic error (if you're very lucky)
- program halts with run-time error (if you're lucky**)
- program never halts (not lucky, but at least you know)
- program completes, but gives incorrect results (if you're unlucky)

** but if the runtime error is due to pointer mismanagement, you're very unlucky.

Debugging has three aspects:

- ***Finding*** the code that's causing the problem
- ***Understanding*** why it's causing the problem
- ***Modifying*** the code to eliminate the problem

Generally, understanding a bug is (usually) easy once you find it. Fixing a bug is (usually) easy one you find/understand it. To fix your bug, re-examine the spec, and modify the code to satisfy the spec.

The easiest bugs to find are the ones the compiler tells you about.
The most difficult bugs to find are the one that are not reproducible (random), and those involving pointers/dynamically-allocated memory.

Assumptions are what makes debugging difficult. It is likely that an onlooker will find the bug quicker than you.

Debugging Strategies

The following will assist you in the task of finding bugs:

- make the bug reproducible
- search for patterns in the failure
- divide and conquer (isolate the buggy region)
- write self-checking code (e.g. assert)
- write a log file (execution trace)
- draw a picture (esp. for pointer bugs)

Debuggers are tools to assist in finding bugs.
They typically provide facilities to control the execution of a program (step-by-step execution, breakpoints), and view the intermediate state of the program (values stored in data structures, control stack).
Examples include:

- `gdb`, `lldb` - command-line debugger, useful for quick checks
- `ddd` - visual debugger, can display dynamic data
- `valgrind` - execution "harness" for pointer bugs

The Debugging Process

Debugging requires a detailed understanding of ***p*rogram state**.

The ***state*** of a program comprises of:

- the *location* where it's current executing
- the *names/values* of all active *variables on the stack*
- the *names/values* of all *data in the global/heap regions*

A simple example, considering just local vars in a function:
"at this point in the program, `x==3`, `y==7`, and `z==-2`"
Note: for any realistic program, the state will be rather large.

The *real* difficulty of debugging is *locating* the bug. Since a bug is "code with unintended action",

you need to know:

- in detail, what the code ***should*** do
- in detail, what the code ***actually*** does

In any non-trivial program, the sheer amount of *detail* is a problem. The trick to effective debugging is narrowing the ***focus of attention***. That is, employ a search strategy that enables you to zoom in on the bug.

When you run a buggy program, initially everything is ok.
At some point, the buggy statement is executed the program state changes from valid to incorrect, but the program won't necessarily crash at this point.
The goal is to find the point at which the state becomes "corrupted". Initially you know broadly where the problem is. Then you move to the function where the problem occurs. Then you find the precise statement with the bug.
Typically, you need to determine which variables "got the wrong value".

Locating the Bug

A simple search strategy for debugging is as follows:

1. initially the whole program is "suspect"
2. put a print statement in the middle of the suspect part of the program to display values of variables at that point
3. if the values are correct, then the bug must be in the second part of the execution
4. if the values are incorrect, the bug is in the first half
5. now restrict your attention to just the relevant half of the program (it becomes the new "suspect part of the program")
6. repeat the above steps

At each stage you have eliminated half of the program from suspicion.
In not too many steps, you will have identified a specific buggy statement.
The problems:

- which variables to print? all? what if too many?
- how to work out where the "half-way execution point" is?

Side note: this approach won't necessarily find an existing bug ...
E.g. `x:int { y = x+x; } y==x`^{2}, when initially `x==2`

A slightly smarter strategy, relying on the typical structure of programs:

- display all major data structures just after they have been initialised
  - typically requires to implement a print function for each data structure (e.g. 2d array)
- display major data structures at "strategic points" during the program's execution
- display major data structures at the end of program execution

How to determine strategic points? E.g.

- after the first, second, middle iterations of the main program loop
- after the keystroke that causes an interactive program to crash
- after the point where the last output from the program occurred

Examining Program State

A vital tool for debugging is a mechanism to display state.
One method: diagnostic printf statements of "suspect" variables.
Problems with this approach:

- it changes the program (so you're not debugging quite the same thing)
- you may guess wrong about what the suspect variables are
- generally too much is printed (e.g. printing to trace execution of a loop)

An alternative for obtaining access to program state:

- a tool that allows you to stop program execution at a certain point
- and then allows you to inspect the state (preferably selectively)

This is precisely what *debuggers* such as **gdb** provide.
Debuggers also allow you to inspect the state of a program that has crashed due to a run-time

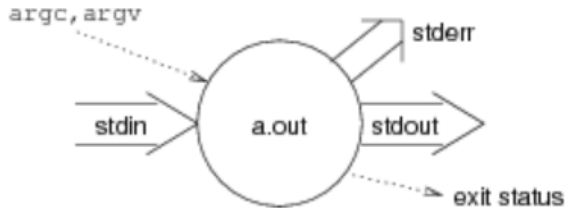
error.
Often, this takes you straight to the point where the bug occurred.

C Program Execution

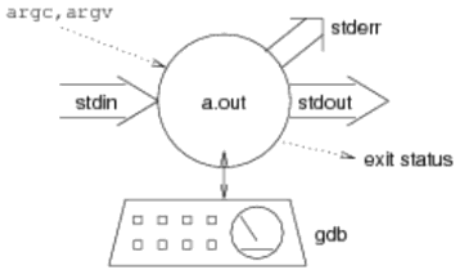
Under Unix, a C program executes either:

a. to completion, producing *(correct?)* results
b. until the program detects an error and calls `exit()`
c. until a run-time error halts it (e.g. Segmentation violation)

Normal C execution environment:



C execution environment with a debugger:



Debuggers

A debugger gives control of program execution:

- normal execution (`run`, `cont`)
- stop at a certain point (`break`)
- one statement at a time (`step`, `next`)
- examine program state (`print`)

The gdb command is a command-line-based debugger for C, C++ ...
There are GUI front-ends available (e.g. `xxgdb`, `ddd`, etc.) which provide facilities such as graphical display of data structures.

For gdb, programs must be compiled with the `gcc -g` flag.
gdb takes two command line arguments:

```
$ gdb executable core
```

E.g.

```
$ gdb  a.out  core
$ gdb  myprog
```

The core argument is optional.

gdb Sessions

gdb is like a shell to control and monitor an executing C program.
Example session:

```
$ gcc -g -Wall -Werror -o prog prog.c
$ gdb prog
Copyright (C) 2014 Free Software Foundation, Inc...
(gdb) break 9
Breakpoint 1 at 0x100f03: file prog.c, line 9.
(gdb) run
/Users/comp1921 Starting program: ..../prog
Breakpoint 1, main (argc=1, argv=0x7ffbc8) at prog.c:9
9               for (i = 1; i <= 3; i++ {
(gdb) next
10              sum += a[i];
(gdb) print sum
$1 = 0
(gdb) print a[i]
$2 = 4
(gdb) print i
$3 = 1
(gdb) print a@1
$4 = {{7, 4, 3}}
```

```
(gdb) cont
...
```

Basic gdb Commands

`quit`	Quits from gdb
`help [CMD]`	Online help. Give information about CMD command

`run ARGS`	Run the program.
	The ARGS are whatever you normally use. e.g.
	`$ xyz < data`
	is achieved by:
	`(gdb) run < data`

gdb Status Commands

`where`	Find which function the program was executing when it crashed
`list [LINE]`	Display 5 lines on either size of the current statement
`print EXPR`	Display expression values EXPR may use (current values of) variables The special expression `a@1` shows all of the array a

gdb Execution Commands

`break [PROC|LINE]`	Set breakpoint. On entry to procedure PROC (or reaching line LINE), execution stops and control is returned to gdb.
`next`	Single step (***over procedures***). Execute the next statement. If the statement is a procedure call, execute the entire procedure body.
`step`	Single step (***into procedures***). Execute the next statement. If the statement is a procedure call, go to the first statement in the procedure body.

For more details see gdb's online help.

Using a Debugger

The most common time to invoke a debugger is after a run-time error.
If this produces a `core` file, start gdb. It typically shows you the line of code causing the crash.

- Use `where` to find out who called the current function.
- Pay attention to parameter values in the stack
- Use `list` to see the current function code
- Display values of local variables

Note: that the program may crash well after the bug.

Once you find that the value of a given variable (e.g. x) is wrong, the next step is to determine ***why*** it is wrong.
There are two possibilities:

- the statement that assigned a value to x is wrong
- the values of other variables used by that statement are wrong

Example:

```
if (c > 0) { x = a+b; }
```

If we know that

- x is wrong after this statement
- the condition and the expression correctly implement their specs

Then we need to find out where a, b and c were set.

Laws of Debugging

Courtesy of Zoltan Somogyi, Melbourne University

"Before you can fix it, you must be able to break it (consistently)."
(non-reproducible bugs ... Heisenbugs ... are extremely difficult to deal with)

"If you can't find a bug where you're looking, you're looking in the wrong place. "
(taking a break and resuming the debugging task later is generally a good idea)

"It takes two people to find a subtle bug, but only one of them needs to know the program. "
(the second person simply asks questions to challenge the debugger's assumptions)
(In fact, sometimes the second person doesn't have to do or say anything! The process of explaining the problem is often enough to trigger a Eureka event.)

Possibly Untrue Assumptions

Debugging can be extremely frustrating when you make assumptions about the problem which turn out to be wrong.
Some things to be wary of:

- the executable comes from the source code you're reading
- the problem *must* be in this source file
- the problem *cannot* be in this source file
- code that calls a function never provides unexpected arguments
  (e.g. "this pointer will never be NULL; why would anyone pass a NULL?")
- library functions never return an error status
  (e.g. "malloc will always give the memory I ask for")

Performance Tuning

Why do we care about performance?
Because good performance means ***less*** hardware, and ***happy*** users,
while bad performance means ***more*** hardware, and ***unhappy*** users.

Generally, performance is equivalent to execution time; ***performance = execution time***
Other measures include memory/disk space, network traffic, disk I/O

Execution time can be measured in two ways:

- ***CPU*** - the time your program spends in the processor
- ***Elapsed*** - the wall-clock tie between start and finish

In the past, performance was a significant problem. Much programming effort was spent on efficiency *'tricks'.*
Unfortunately, there is a trade-off between execution ***efficiency*** achieved by tweaking code and the ***understandability*** of the code.

Development Strategy

A pragmatic approach to efficiency:
First, make the program simple, clear, robust and ***correct***. Then, worry about efficiency (if it's a problem at all).

Points to note:

- good design is always critical - (at design time, make sensible choice of data structures, algorithms)
- can handle efficiency at system level - (e.g. buy a bigger machine, use compiler optimisation, ...)

Strategy for developing efficient programs:

1. Design the program well
2. Implement the program well
3. Test the program well
4. Only after you're sure it's working, ***measure*** performance
5. If (and only if) performance is inadequate, *find* the ***hot spots***

6. *Tune* the code to fix these
7. Repeat measure-analyse-tune cycle until performance ok

Performance Analysis

Complexity/estimates give us some idea of performance in advance.
Often, however, assumptions made in estimating performance are ***invalid***, as we tend to overlook some frequent and/or expensive operation(s).

The best way to evaluate performance is to: ***measure program execution.***

Performance analysis can be:

- ***coarse-grained*** - an overview of ***performance characteristics***
- ***fine-grained*** - a detailed ***description of performance***

Coarse-grained performance analysis devise a range of ***representative*** inputs and measures the execution time of the program on each input. This can conveniently be combined with testing (but we only care about timing if correct result produced).

The Unix time command provides a suitable mechanism

```
$ time ./myProg < LargeInput > /dev/null
real  0m5.064s
user  0m4.113s
sys   0m0.802s
```

Decades of empirical study of program execution have shown that the `90/10` rule generally holds (or 80/20 rule or ...): ``*90% of the execution time is spent in 10% of the code*''

This implies that most of the code has little impact on the overall performance and small regions of the code are bottlenecks (aka ***hot-spots***)
To significantly improve performance, we make the bottlenecks faster.

Profiles

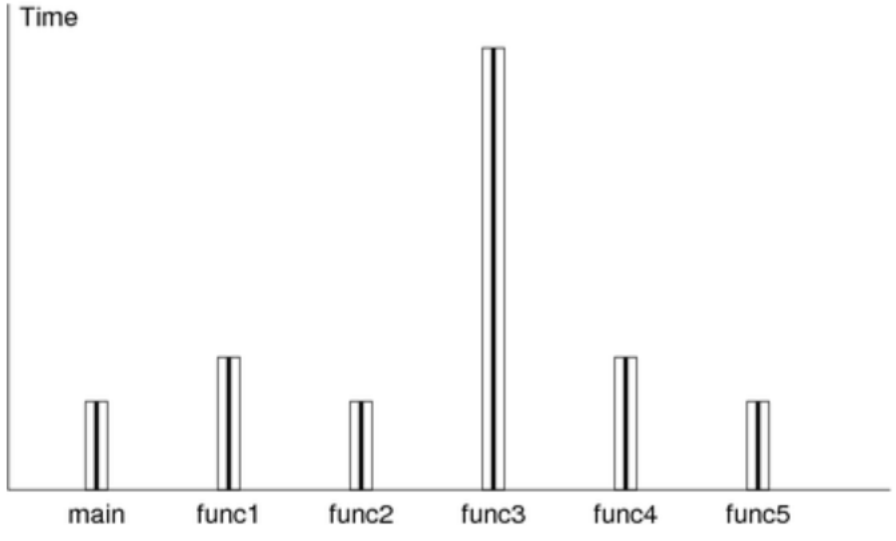
We need a method for locating ***hot spots*** (the expensive 10% of code).

An ***execution profile*** for a program is the total cost of performing each *code block* for one execution of the program. The cost may be measured via:

- a count of the number of times the block is executed
- the total execution time spent within that block

Profiles are typically collected at function level (i.e. code block = function).

A profile shows how much time is spent in each code block.



Software tools can generate profiles of program execution.

gprof: A Profiler

The gprof command displays execution profiles.
It must compile program with the `gcc -pg` flag. Executing program creates an extra gmon.out file. gprof reads gmon.out and prints profile on stdout.

Example of use:

```
$ gcc -pg -o xyz xyz.c
$ xyz < data > /dev/null
$ gprof xyz | less
```

For further usage details, man gprof.

The gprof command works at the function level.
It gives a table (***flat profile***) containing:

- number of times each function was called
- % of total execution time spent in the function
- average execution time per call to that function
- execution time for this function and its children

They are arranged in order from most expensive function down.
It also gives a ***call graph***, a list for each function:

- which functions called this function
- which functions were called by this function

Profile Example

Consider the following program that

- searches for words in text containing a given substring
- displays each such word once (in alphabetical order)

```
int main(int argc, char*argv[])
{
    char  word[MAXWORD];
    List  matches;
    char  *substring;
    FILE  *input;
matches = NULL;
    while (getWord(input, word) != NULL) {
        if (contains(word,substring)
                && !member(matches,word))
            matches = insert(matches,word);
    }
    printWords(matches);
    return 0;
}
```

Flat profile for this program (xwords et data3):

```
  %  cumulative     self              self    total
 time   seconds  seconds  calls  us/call  us/call  name
 75.00     0.03     0.03  30212     0.99     0.99  getWord
 25.00     0.04     0.01  30211     0.33     0.33  contains
  0.00     0.04     0.00    489     0.00     0.00  member
  0.00     0.04     0.00    267     0.00     0.00  insert
  0.00     0.04     0.00      1     0.00 40000.00  main
  0.00     0.04     0.00      1     0.00     0.00  printWords
```

Note: wc data3 → 7439 30211 188259.

Call graph for the same execution (xwords et data3):

```
index %time    self  children    called     name
                0.00    0.04       1/1           _start [2]
[1]    100.0    0.00    0.04       1         main [1]
                0.03    0.00   30212/30212       getWord [3]
                0.01    0.00   30211/30211       contains [4]
                0.00    0.00     489/489         member [5]
                0.00    0.00     267/267         insert [6]
                0.00    0.00       1/1           printWords [7]
-----------------------------------------------
[2]    100.0    0.00    0.04                 _start [2]
                0.00    0.04       1/1           main [1]
-----------------------------------------------
                0.03    0.00   30212/30212       main [1]
[3]     75.0    0.03    0.00   30212         getWord [3]
-----------------------------------------------
                0.01    0.00   30211/30211       main [1]
[4]     25.0    0.01    0.00   30211         contains [4]
-----------------------------------------------
                0.00    0.00     489/489         main [1]
[5]      0.0    0.00    0.00     489         member [5]
-----------------------------------------------
                0.00    0.00     267/267         main [1]
[6]      0.0    0.00    0.00     267         insert [6]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[7]      0.0    0.00    0.00       1         printWords [7]
```

Graph Data Structures

Monday, 20 August 2018 4:29 PM

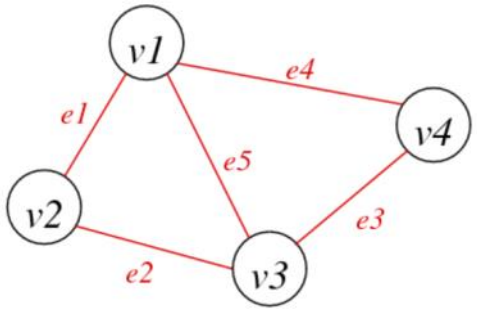
Graphs

Many applications require a collection of items (i.e. a set) and relationships/connections between these items.

Example:

- Maps: items are cities, connections are roads
- Web: items are pages, connections are hyperlinks

A graph G = (V, E), where V is a set of vertices and is a set of edges (a subset of V×V)
Example:



$V = \{v1, v2, v3, v4\}$

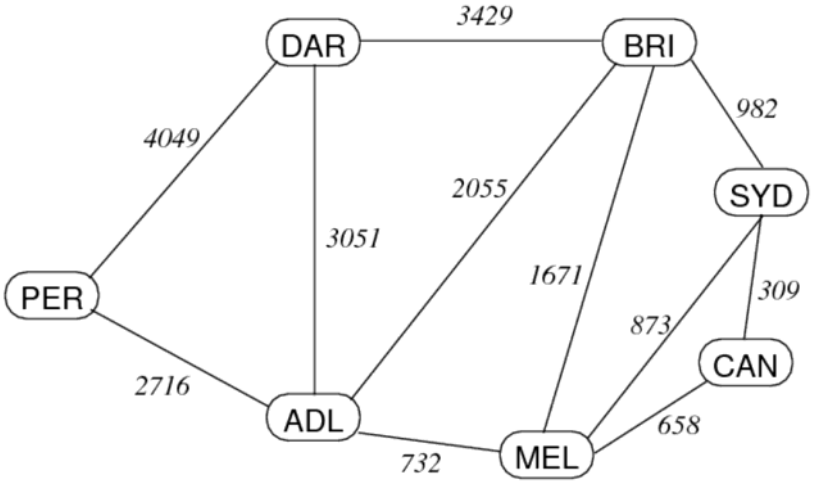
$E = \{e1, e2, e3, e4, e5\}$

A real example: Australian road distances

Distance	Adelaide	Brisbane	Canberra	Darwin	Melbourne	Perth	Sydney
Adelaide	-	2055	1390	3051	732	2716	1605
Brisbane	2055	-	1291	3429	1671	4771	982
Canberra	1390	1291	-	4441	658	4106	309
Darwin	3051	3429	4441	-	3783	4049	4411
Melbourne	732	1671	658	3783	-	3448	873
Perth	2716	4771	4106	4049	3448	-	3972
Sydney	1605	982	309	4411	873	3972	-

Note: vertices are the cities; edges are distance between cities; symmetric

An alternative representation of the above:



Some questions we might ask about a graph include:

- Is there a way to get from item A to item B?
- What is the best way to get from A to B?
- Which items are connected?

Graph algorithms are generally more complex than tree/list ones.

- There is no implicit order of functions.
- Graphs may contain cycles
- Concrete representation is less obvious
- Algorithm complexity depends on connection complexity

Properties of Graphs

Terminology: $|V|$ and $|E|$ (cardinality) normally written just as V and E.

A graph with V vertices has at most $\frac{V(V-1)}{2}$ edges. Why?

There are V vertices. The most connections each vertex can have is with every other vertex, which is V-1. But this means that the edges are counted twice, so divide by 2.

Alternatively:

There are V vertices. V_1 can be connected with V-1 vertices, V_2 can be connected with V-2 vertices, and so on. Eventually we get the arithmetic sum:

$$(V-1)+(V-2)+\cdots+1+0=\frac{V}{2}(V-1)$$

The ratio E:V can vary considerably.

- If E is closer to V^2, the graph is ***dense***
- If E is closer to V, the graph is ***sparse***

Knowing whether a graph is sparse or dense is important. It may:

- affect the choice of data structures to represent graphs
- affect the choice of algorithms to process a graph

Graph Terminology

For an edge *e* that connects vertices *v* and *w*.

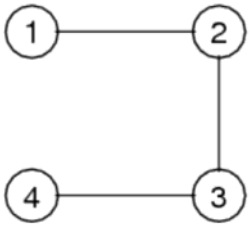
- *v* and *w* are ***adjacent***
- *e* is ***incident*** on both *v* and *w*

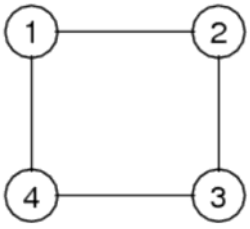
The ***degree*** of a vertex *v* is the no. of edges incident with *v*.

A ***path*** is a sequence of vertices and edges, where there are **NO** repeated edges. Each vertex has an edge to its predecessor

A ***cycle/circuit*** is a path, where the last vertex in the path is the same as the first vertex.

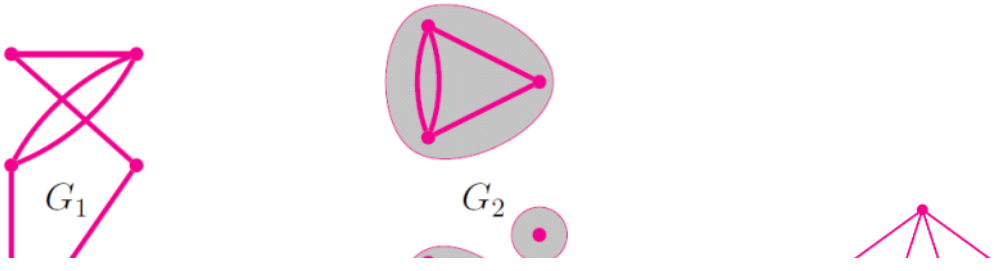
The ***length*** of a path, or cycle is the no. of edges involved.

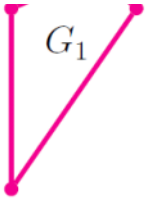


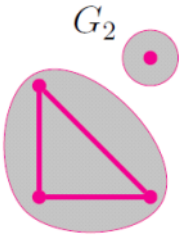


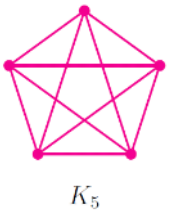
Path: 1–2, 2–3, 3–4 *Cycle: 1–2, 2–3, 3–4, 4–1*

A ***connected graph*** is a graph, where there is a path from each vertex to every other vertex. If the graph is not connected, it has ≥ 2 ***connected components***. Example:









G_1 is connected, while G_2 has 3 connected components.

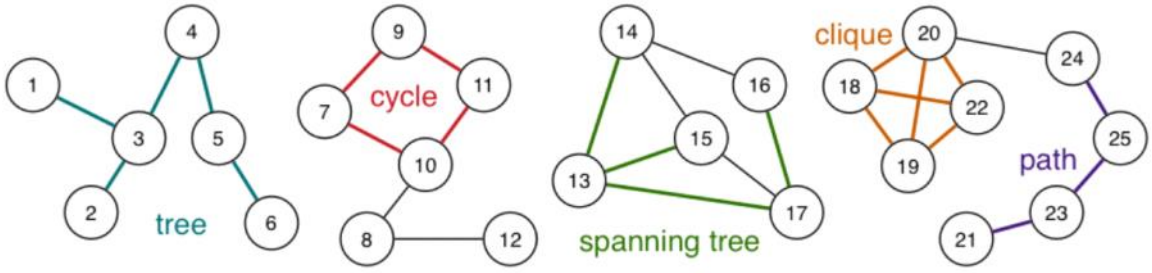
A ***complete graph*** K_v is a graph, where there is an edge from each vertex to every other vertex. In a complete graph, $E = \frac{V}{2}(V-1)$.

A ***tree*** is a connected (sub)graph with no cycles/circuit.

A ***spanning tree*** is a (sub)graph, that is a tree and contains every vertex of a graph G.

A ***clique*** is a complete subgraph.

Consider the following as a single graph:



The graph has 25 vertices, 32 edges, and 4 connected components.

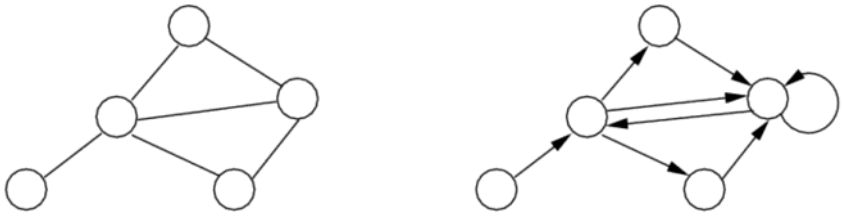
Note: The entire graph has no spanning tree;
what is shown in green is a spanning tree of the third connected component.

A ***spanning tree*** of a connected graph G = (V, E) is a subgraph of G containing all vertices of V, and is a **single** tree (connected, and no cycles).

A ***spanning forest*** of a non-connected graph G = (V, E) is a subgraph of G containing all vertices of V, and is a **set** of trees (not connected, and no cycles), where there is one tree for each connected component.

An ***undirected graph*** is a graph, where edge(u, v) = edge(v, u) and there are no self-loops (i.e. no edge(v, v)).

A ***directed graph*** is a graph, where edge(u, v) ≠ edge(v, u), and it can have self-loops (i.e. edge(v, v)).



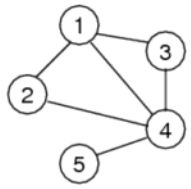
Undirected graph *Directed graph*

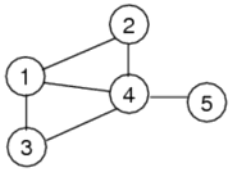
A ***weighted graph*** is a graph where each edge has an associated value (weight). e.g. a road mad, where the weights on the edges are the distances between cities.

A ***multi-graph*** is a graph, which allows multiple edges between two vertices.

Graph Data Structures

Defining a graphs needs some way of identifying vertices.
We could give diagram showing edges and vertices, or we could give a list of edges.
E.g. four representations of the same graph:





1–2 1–3 1–4
2–4
3–4
4–5

1–3
2–1 2–4
4–1 4–3
5–4

(a) *(b)* *(c)* *(d)*

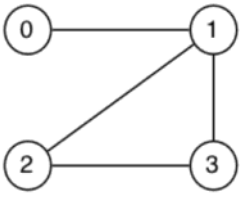
We will discuss three different graph data structures:

1. Array of edges
2. Adjacency matrix
3. Adjacency list

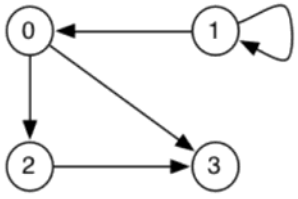
Array of Edges Representations

Edges are represented as an array of Edge values (= pairs of vertices).
It is a space efficient representation, however adding and deleting edges is slightly complex.
When the graph is

- undirected: order of vertices in an Edge doesn't matter
- directed: order of vertices in an Edge encodes direction



[(0,1), (1,2), (1,3), (2,3)]



[(1,0), (1,1), (0.2), (0,3), (2,3)]

For simplicity, we always assume vertices to be numbered 0..V-1.

Graph initialisation:

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V   // #vertices (numbered 0..V-1)
|  g.nE = 0   // #edges
|  allocate enough memory for g.edges[]
|  return g
```

Edge insertion:

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  i=0
|  while i<g.nE ∧ (v,w)≠g.edges[i] do
|     i=i+1
|  end while
|  if i=g.nE then      // (v, w) not found
|     g.edges[i]=(v,w)
|     g.nE=g.nE+1
|  end if
```

Edge removal:

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
```

```
|  i=0
|  while i<g.nE ∧ (v,w)≠g.edges[i] do
|     i=i+1
|  end while
|  if i<g.nE then                  // (v, w) found
|     g.edges[i]=g.edges[g.nE-1] // replace by last edge in array
|     g.nE=g.nE-1
|  end if
```

Cost Analysis

Storage cost: *O(E)*

Cost of operations:

- initialisation: *O(1)*
- insert edge: *O(E)* (assuming edge array has space)
- delete edge: *O(E)* (need to find edge in edge array)

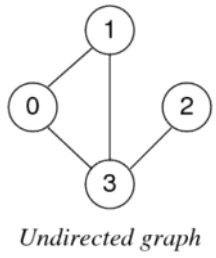
If array is full on insert, allocate space for a bigger array, copy edges across ⇒ still *O(E)*

If we maintain edges in order, use binary search to find edge ⇒ *O(log E)*

Note: not a very popular way of representing graphs

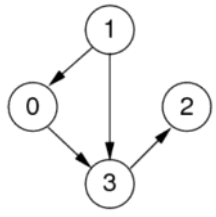
Adjacency Matrix Representation

In an adjacency matrix, the edges are represented by a $V \times V$ matrix



Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0



Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

Advantages of Adjacency Matrix Representation:

- Easily implemented as a 2-d array
- Can represent graphs, digraphs, and weighted graphs
  - Graphs: symmetric boolean matric
  - Digraphs: non-symmetric boolean matrix
  - Weighted graphs: non-symmetric matrix of weight values

Disadvantages of Adjacency Matrix Representation:

- If the graph is spares (a few edges), it is an inefficient use of memory

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V   // #vertices (numbered 0..V-1)
|  g.nE = 0   // #edges
|  allocate memory for g.edges[][]
|  for all i,j=0..V-1 do
|     g.edges[i][j]=0   // false
|  end for
|  return g
```

Edge insertion

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]=0 then  // (v,w) not in graph
|     g.edges[v][w]=1       // set to true
|     g.edges[w][v]=1
|     g.nE=g.nE+1
|  end if
```

Edge removal

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]≠0 then  // (v, w) in graph
|     g.edges[v][w]=0       // set to false
|     g.edges[w][v]=0
|     g.nE=g.nE-1
|  end if
```

Cost Analysis

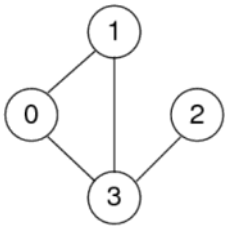
Storage cost: $O(V^2)$

If the graph is sparse, most of the storage is wasted

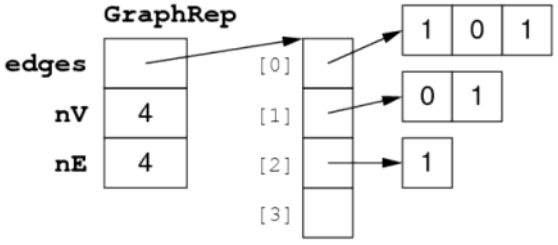
Cost of operations:

- Initialisation: $O(V^2)$ (initialise V×V matrix)
- Insert edge: O(1) (set two cells in matrix)
- Remove edge: O(1) (unset two cells in matrix)

A storage optimisation would be to store only the op-right part of the matrix: e.g.



Undirected graph

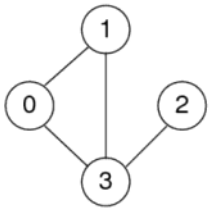


New storage cost: V-1 int pointers $\frac{V}{2}(V-1)$ ints (but still $O(V^2)$)

This method requires us to always use edges (*v, w*) such that *v* < *w*

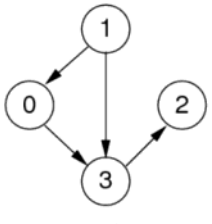
Adjacency List Representation

In an adjacency list representation, for each vertex, you store a linked list of adjacent vertices:



Undirected graph

A[0] = <1, 3>
A[1] = <0, 3>
A[2] = <3>
A[3] = <0, 1, 2>



Directed graph

A[0] = <3>
A[1] = <0, 3>
A[2] = < >
A[3] = <2>

Advantages of Adjacency List Representation:

- Relatively easy to implement in languages like C
- Can represent graphs and digraphs
- Is memory efficient if *E:V* ratio is relatively small

Disadvantages of Adjacency List Representation:

- One graph has many possible representations (unless the lists are ordered by the same criterion. e.g. ascending)

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V   // #vertices (numbered 0..V-1)
|  g.nE = 0   // #edges
|  allocate memory for g.edges[]
|  for all i=0..V-1 do
|     g.edges[i]=NULL   // empty list
|  end for
|  return g
```

Edge insertion:

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if ~inLL(g.edges[v],w) then  // (v, w) not in graph
|     insertLL(g.edges[v],w)
|     insertLL(g.edges[w],v)
|     g.nE=g.nE+1
|  end if
```

Edge removal:

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if inLL(g.edges[v],w) then  // (v, w) in graph
|     deleteLL(g.edges[v],w)
|     deleteLL(g.edges[w],v)
|     g.nE=g.nE-1
|  end if
```

Cost Analysis

Storage cost: O(*V* + *E*)

Cost of operations:

- Initialisation: O(*V*) (initialise *V* lists)
- Insert edge: O(1) (insert one vertex into list)
  - If you don't check for duplicates
- Delete edge: O(*E*) (need to find vertex in list)

If the lists are sorted:

- Inserting an edges requires a search of the list ⇒ *O(E)*
- Deleting an edge always requires a search, regardless of the list order

Comparison of Graph Representations

	array of edges	adjacency matrix	adjacency list
space usage	*E*	V^2	*V+E*
initialise	*1*	V^2	*V*
insert edge	*E*	*1*	*1*

remove edge	*E*	*1*	*E*

Other operations:

	array of edges	adjacency matrix	adjacency list
disconnected(v)?	*E*	*V*	*1*
isPath(x,y)?	*E·log V*	V^2	*V+E*
copy graph	*E*	V^2	*V+E*
destroy graph	*1*	*V*	*V+E*

Graph Abstract Data Types

Graph ADT

The data of a graph ADT is a set of edges and a set of vertices.

Its operations include:

- Building: creating a graph, adding an edge
- Deleting: removing an edge, dropping a whole graph
- Scanning: checking is a graph contains a given edge

Some things to node:

- The set of vertices is **fixed** when the graph is initialised
- We treat vertices as `ints`, but they could be arbitrary `Items`

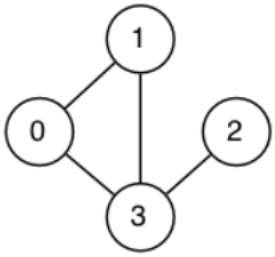
A graph ADT interface `graph.h`

```
// graph representation is hidden
typedef struct GraphRep *Graph;

// vertices denoted by integers 0..N-1
typedef int Vertex;

// edges are pairs of vertices (end-points)
typedef struct Edge { Vertex v; Vertex w; } Edge;

// operations on graphs
Graph newGraph(int V);                  // new graph with V vertices
void  insertEdge(Graph, Edge);
void  removeEdge(Graph, Edge);
bool  adjacent(Graph, Vertex, Vertex); // is there an edge b/w two vertices

void  freeGraph(Graph);
```

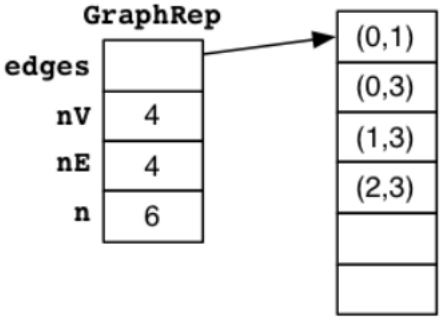
Graph ADT (Array of Edges)

Implementation of `GraphRep` (array-of-edges representation)

```
typedef struct GraphRep {
   Edge *edges; // array of edges
   int   nV;    // #vertices (numbered 0..V-1)
   int   nE;    // #edges
   int   n;     // size of edge array
} GraphRep;
```



Undirected graph



Implementation of graph initialisation (array-of-edges representation)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));    assert(g != NULL);
   g->nV = V; g->nE = 0;
   // allocate enough memory for edges
   g->n = Enough;
   g->edges = malloc(g->n*sizeof(Edge));  assert(g->edges != NULL);
   return g;
}
```

How much is *enough*? … No more than $\frac{V}{2}(V-1)$ In fact, much less in practice (especially for a sparse graph)

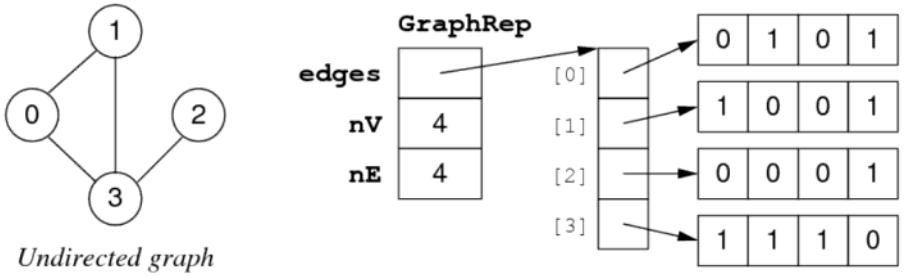
Implementation of edge insertion/removal (array-of-edges representation)

```
// check if two edges are equal
bool eq(Edge e1, Edge e2) {
   return ( (e1.v == e2.v && e1.w == e2.w)
            || (e1.v == e2.w && e1.w == e2.v) );
}

void insertEdge(Graph g, Edge e) {
   // ensure that g exists and array of edges isn't full
   assert(g != NULL && g->nE < g->n);
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i == g->nE)                        // edge e not found
      g->edges[g->nE++] = e;
}

void removeEdge(Graph g, Edge e) {
   assert(g != NULL);                     // ensure that g exists
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i < g->nE)                         // edge e found
      g->edges[i] = g->edges[--g->nE];
}
```

Graph ADT (Adjacency Matrix)

Implementation of GraphRep (adjacency-matrix representation)

```
typedef struct GraphRep {
   int  **edges;
   int    nV;
   int    nE;
} GraphRep;
```



Implementation of graph initialisation (adjacency-matrix representation)

```
Graph newGraph(int V) {
   assert(V >= 0);
   int i;
   Graph g = malloc(sizeof(GraphRep));      assert(g != NULL);
   g->nV = V;  g->nE = 0;

   // make an array of pointers
   g->edges = malloc(V * sizeof(int *));    assert(g->edges != NULL);

   // for each pointer allocate make an array of edges
   // so that we make an array of arrays (a matrix)
   for (i = 0; i < V; i++) {
      g->edges[i] = calloc(V, sizeof(int)); assert(g->edges[i] != NULL);
   }
   return g;
}
```

Note: the standard library function **`calloc(size_t nelems, size_t nbytes)`**

- allocates a memory block of size `nelems*nbytes`
- and sets all bytes in that block to ***zero***

Implementation of edge insertion/removal (adjacency-matrix representation)

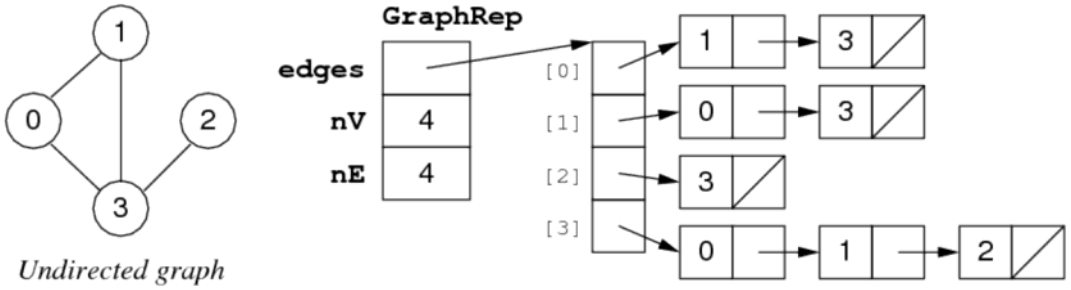
```
// check if vertex is valid in a graph
bool validV(Graph g, Vertex v) {
   return (g != NULL && v >= 0 && v < g->nV);
}

void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));

   if (!g->edges[e.v][e.w]) {  // edge e not in graph
      g->edges[e.v][e.w] = 1;
      g->edges[e.w][e.v] = 1;
      g->nE++;
   }
}

void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));

   if (g->edges[e.v][e.w]) {   // edge e in graph
      g->edges[e.v][e.w] = 0;
      g->edges[e.w][e.v] = 0;
      g->nE--;
   }
}
```

Graph ADT (Adjacency List)

Implementation of GraphRep (adjacency-list representation)

```
typedef struct GraphRep {
   Node **edges;  // array of lists
   int    nV;     // #vertices
   int    nE;     // #edges
} GraphRep;
typedef struct Node {
   Vertex       v;
   struct Node *next;
} Node;
```

In this representation, you have an array of nodes/vertices, where each node points to a linked list of adjacent nodes that it is attached to.



Implementation of graph initialisation (adjacency-list representation)

```
Graph newGraph(int V) {
   assert(V >= 0);
   int i;

   Graph g = malloc(sizeof(GraphRep));     assert(g != NULL);
   g->nV = V;  g->nE = 0;

   // allocate memory for array of lists
   g->edges = malloc(V * sizeof(Node *));  assert(g->edges != NULL);
   for (i = 0; i < V; i++)
      g->edges[i] = NULL;

   return g;
}
```

Implementation of edge insertion/removal (adjacency-list representation)

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));
if (!inLL(g->edges[e.v], e.w)) {
      g->edges[e.v] = insertLL(g->edges[e.v], e.w);
      g->edges[e.w] = insertLL(g->edges[e.w], e.v);
      g->nE++;
   }
}
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));
if (inLL(g->edges[e.v], e.w)) {
      g->edges[e.v] = deleteLL(g->edges[e.v], e.w);
      g->edges[e.w] = deleteLL(g->edges[e.w], e.v);
      g->nE--;
   }
}
```

`inLL`, `insertLL`, `deleteLL` are standard linked list operations (as discussed in week 3)

Graph Algorithms 1

Thursday, 23 August 2018 10:41 AM

Problems on Graphs

What kind of problems do we want to solve on/via graphs?

- is the graph fully-connected?
- can we remove an edge and keep it fully-connected?
- is one vertex reachable starting from some other vertex?
- what is the cheapest cost path from *v* to *w*?
- which vertices are reachable from *v*? (transitive closure)
- is there a cycle that passes through all vertices? (circuit)
- is there a tree that links all vertices? (spanning tree)
- what is the minimum spanning tree?
- …
- can a graph be drawn in a plane with no crossing edges? (planar graphs)
- are two graphs "equivalent"? (isomorphism)

Graph Algorithms

We will examine algorithms for:

- Connectivity (simple graphs)
- Path finding (simple/directed graphs)
- Minimum spanning trees (weighted graphs)
- Shortest paths (weighted graphs)

We will also look at generic methods for traversing graphs

Graph Traversal

Finding a Path

Questions on paths:

- is there a path between two given vertices (*src*,*dest*)?
- what is the sequence of vertices from *src* to *dest*?

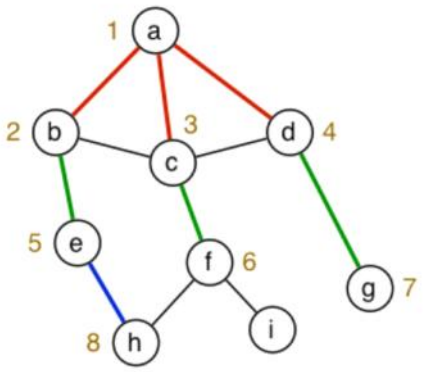
An approach to solving problem:

- examine vertices adjacent to *src v*
- if any of them is *dest*, then we are done
- otherwise try vertices two edges from *v*
- Repeat, looking further and further from *v*

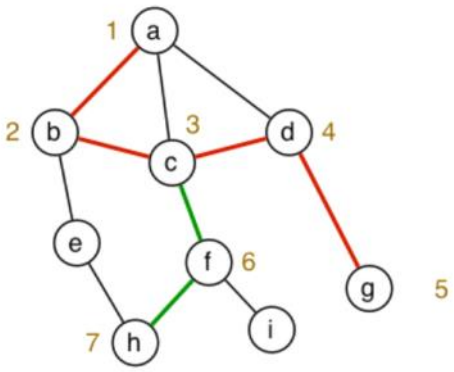
There are two strategies for graph traversal/search:

- ***depth-first*** (DFS) - follows one path to completion before considering others
- ***breadth-first*** (BFS) - "fans-out" from the starting vertex ("spreading" subgraph)

A comparison of BFS/DFS search for checking if there is a path from *a* to *h*



Breadth-first Search



Depth-first Search

Both approaches ignore some edges by remembering previously visited vertices

Depth-first favours following a path rather than looking at its neighbours. It can be implemented recursively or iteratively (via stack). A full traversal produces a ***depth-first spanning tree***.

Breadth-first favours looking at its neighbours rather than path following. It can be implemented iteratively (via queue). A full traversal produces a ***breadth-first spanning tree***.

Depth-first Search

A depth-first search can be described recursively as:

```
depthFirst(G,v):
  1. mark v as visited
  2. for each (v,w)∈edges(G) do
       if w has not been visited then
          depthFirst(w)
```

The recursion induces *backtracking.*

Recursive DFS path checking:

```
hasPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if there is a path from src to dest in G,
|         false otherwise
|
|  return dfsPathCheck(G,src,dest)
dfsPathCheck(G,v,dest):
|  mark v as visited
|  for all (v,w)∈edges(G) do
|  |  if w=dest then       // found edge to destination
|  |     return true
|  |  else if w has not been visited then
|  |     if dfsPathCheck(G,w,dest) then
|  |        return true    // found path via w to destination
|  |     end if
|  |  end if
|  end for
|  return false            // no path from v to destination
```

Cost analysis:

- Each vertex is visited at most once ⇒ cost = *O(V)*
- Visit all edges incident on visited vertices ⇒ cost = *O(E)*
  assuming an adjacency list representation

Time complexity of DFS: *O(V + E)* (adjacency list representation)

Knowing whether a path exists can be useful, but knowing what the path is is even more useful. If we record the previously visited node as we search through the graph, we can trace the path through the graph. To do this, make use of a global variable:

`visited[]` - an array to store the previously visited node, for each node being visited

```
visited[] // store the previously visited node, for each vertex 0..nV-1
findPath(G,src,dest):
|  Input graph G, vertices src,dest
|
|  for all vertices v∈G do          // set all vertices to not visited
|     visited[v]=-1
|  end for
|  visited[src]=src                 // starting node for the path
|  if dfsPathCheck(G,src,dest) then  // show path in dest..src order
|  |  v=dest
|  |  while v≠src do
|  |     print v"-"
|  |     v=visited[v]
|  |  end while
|  |  print src
|  end if
dfsPathCheck(G,v,dest):
|  for all (v,w)∈edges(G) do
|  |  if visited[w]=-1 then
|  |     visited[w]=v               // set visited[w] to current node
|  |     if w=dest then             // found edge from v to dest
|  |        return true
|  |     else if dfsPathCheck(G,w,dest) then
|  |        return true             // found path via w to dest
|  |     end if
|  |  end if
|  end for
|  return false                     // no path from v to dest
```

DFS can also be described non-recursively (via a stack):

```
visited[] // array of visiting orders, indexed by vertex 0..nV-1

findPathDFS(G,src,dest):
|  Input graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  found=false
|  visited[src]=src
|  push src onto new stack s
|  while ~found ∧ s is not empty do
|  |  pop v from s
|  |  if v=dest then
|  |     found=true
|  |  else
|  |  |  for each (v,w)∈edges(G) such that visited[w]=-1 do
|  |  |     visited[w]=v
|  |  |     push w onto s
|  |  |  end for
|  |  end if
|  end while
```

```
|  if found then
|     display path in dest..src order
|  end if
```

Uses standard stack operations (push, pop, check if empty)

Time complexity is the same: *O(V+E)* (each vertex added to stack once, each element in vertex's adjacency list visited once)

Breadth-first Search

The basic approach to breadth-first search is:

- Visit and mark current vertex
- Visit all neighbours of current vertex
- Then consider the neighbours of neighbours

Notes:

- It is tricky to describe this recursively
- A minor variation on non-recursive DFS search works i.e. switch the stack for a queue

BFS algorithm (records visiting order):

```
visited[] // array of visiting orders, indexed by vertex 0..nV-1
findPathBFS(G,src,dest):
|  Input  graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  found=false
|  visited[src]=src
|  enqueue src into new queue q
|  while ~found ∧ q is not empty do
|  |  dequeue v from q
|  |  if v=dest then
|  |     found=true
|  |  else
|  |  |  for each (v,w)∈edges(G) such that visited[w]=-1 do
|  |  |     visited[w]=v
|  |  |     enqueue w into q
|  |  |  end for
|  |  end if
|  end while
|  if found then
|     display path in dest..src order
|  end if
```

Uses standard queue operations (enqueue, dequeue, check if empty)

Time complexity of BFS: O*(V + E)* (same as DFS)

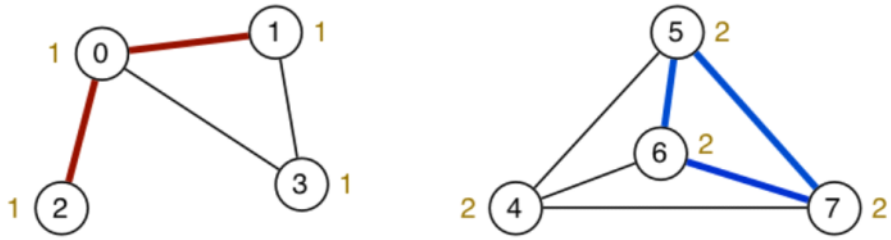
BFS finds a "shortest" path bases on the minimum no. of edges between *src* and *dest*. It stops with the first found path if there are multiple ones.

In many applications, the edges are weighted and we want a path based on minimum sum-of-weights along the path from *src* to *dest.*

Other DFS Examples

Other problems to solves via DFS graph search.

- Checking for the existence of a cycle
- Determining which connected component each vertex is in.



Graph with two connected components, a path and a cycle

A graph has a *cycle* if

- it has a path of length > 1
- with start vertex *src* = end vertex *dest*
- and without using any edge more than once

We are not required to give the path, just indicate its presence.

Computing Connected Components

Some problems we may encounter:

- How many connected subgraphs are there?
- Are two vertices in the same connected subgraph?

Both of the above can be solved if we can build an array, one element for each vertex V indicating which connected component V is in. We can use: `componentof[]` - an array `[0..nV-1]` of component IDs.

An algorithm to assign vertices to connected components:

```
components(G):
|  Input graph G
|
|  for all vertices v∈G do
|     componentOf[v]=-1
|  end for
|  compID=0
|  for all vertices v∈G do
|  |  if componentOf[v]=-1 then
|  |     dfsComponents(G,v,compID)
|  |     compID=compID+1
|  |  end if
|  end for
dfsComponents(G,v,id):
|  componentOf[v]=id
|  for all vertices w adjacent to v do
|     if componentOf[w]=-1 then
|        dfsComponents(G,w,id)
|     end if
|  end for
```

Consider an application, where connectivity is critical. We frequently ask questions of the kind above, but we cannot afford to run `component()` each time. Add a new field to the `GraphRep` structure:

```
typedef struct GraphRep *Graph;

struct GraphRep {
   ...
   int nC;  // # connected components
   int *cc; // which component each vertex is contained in
   ...      // i.e array[0..nV-1] of 0..nC-1
}
```

With this structure, the above tasks become trivial

```
// How many connected subgraphs are there?
int nConnected(Graph g) {
   return g->nC;
}

// Are two vertices in the same connected subgraph?
bool inSameComponent(Graph g, Vertex v, Vertex w) {
   return (g->cc[v] == g->cc[w]);
}
```

Consider maintenance of such a graph representation:

- initially, nC = nV (because no edges)
- adding an edge may reduce nC
- removing an edge may increase nC
- `cc[]` can simplify path checking
  (ensure vertices *v* and *w* are in same component before starting search)

Additional maintenance cost amortised by reduced cost for `nConnected()` and `inSameComponent()`

Hamilton and Euler Paths

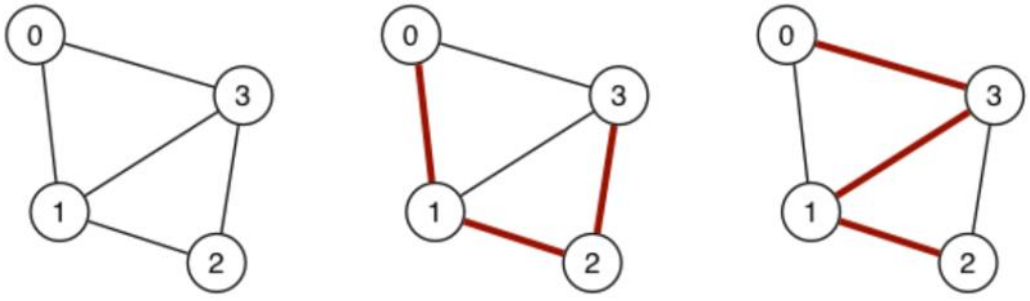
Hamilton Path and Circuit

The ***Hamiltonian path*** problem: find a simple path (a path which does not visit repeat vertices) connecting two vertices *v,w* in graph G, such that the path includes each vertex in G exactly once.
If *v = w* , the we have a ***Hamiltonian circuit*****.**

The problem is simple to state, but difficult to solve (it is NP-complete).

Note: the problem is named after Irish mathematician, physicist and astronomer Sir William Rowan Hamilton (1805-1865)

Example: the following graph has two possible Hamilton paths:



Approach to making a Hamilton path:

- Generate all possible simple paths (using say a DFS)
- Keep a counter of vertices visited in the current path
- Stop with you find a path containing *V* vertices

This can be expressed via a recursive DFS algorithm similar to a simple path finding approach except:

- It keeps track of the path length and is successful it `length = v`
- It resets "visited" marker after an unsuccessful path

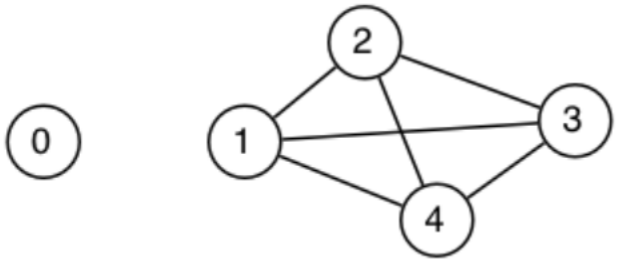
Algorithm for finding a Hamiltonian path:

```
visited[] // array [0..nV-1] to keep track of visited vertices
hasHamiltonianPath(G,src,dest):
|  for all vertices v∈G do
|     visited[v]=false
|  end for
|  return hamiltonR(G,src,dest,#vertices(G)-1)
```

```
|
hamiltonR(G,v,dest,d):
|  Input G    graph
|        v    current vertex considered
|        dest destination vertex
|        d    distance "remaining" until path found
|
|  if v=dest then
|     if d=0 then return true else return false
|  else
|  |  visited[v]=true
|  |  for each (v,w)∈edges(G) ∧ ~visited[w] do
|  |     if hamiltonR(G,w,dest,d-1) then
|  |        return true
|  |     end if
|  |  end for
|  end if
|  visited[v]=false    // reset visited marker
|  return false
```

Analysis: worst case requires *(V-1)!* paths to be examined

Consider a graph with an isolated vertex and the rest fully-connected:



Checking `hasHamiltonPath(g, x, 0)` for any *x*:

- Requires us to consider every possible path
- `e.g 1-2-3-4, 1-2-4-3, 1-3-2-4, 1-3-4-2, 1-4-2-3, …`
- starting from any *x*, there are 3! paths ⇒ 4! total paths
- there is no path of length 5 in these *(V-1)!* possibilities

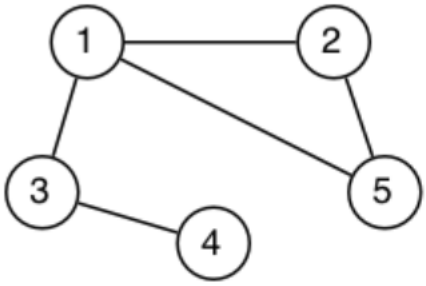
There is no known simpler algorithm for this task ⇒ *NP*-hard

Note, however, that the above case could be solved in constant time if we had a fast check for 0 and *x* being in the same connected component.

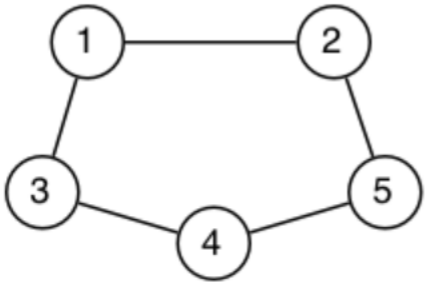
Euler Path and Circuit

The ***Euler path*** problem: find a path connecting two vertices *v,w* in graph G such that the path includes each *edge* exactly once (note that such path does not have to be simple; it can visit vertices more than once).

If *v = w*, then we have an ***Euler circuit***.



Euler Path: 4-3-1-5-2-1



Euler Circuit: 1-2-5-4-3-1

The problem was named after a Swiss mathematician, physicist, astronomer, logician and engineer Leonhard Euler (1707-1783) and was based on a circuitous route via bridges in Konigsberg.

A possible "brute-force" approach:

- Check if each path is an Euler path

This would result in a factorial time performance.

We can develop a better algorithm by exploiting theorems:

1. *A graph has an Euler circuit* if and only if it is connected and **all** vertices have and even degree.
2. *A graph has an non-circuitous Euler path* if and only if it is connected and exactly **two** vertices have an odd degree.

Assume the existence of degree(g,v) (degree of a vertex, cf. problem set week 6)
Algorithm to check whether a graph has an Euler path:

```
hasEulerPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if G has Euler path from src to dest
|         false otherwise
|
|  if src≠dest then
|     if degree(G,src) is even ∨ degree(G,dest) is even then
|        return false
|     end if
|  else if degree(G,src) is odd then
|     return false
|  end if
|  for all vertices v∈G do
|     if v≠src ∧ v≠dest ∧ degree(G,v) is odd then
|        return false
|     end if
|  end for
|  return true
```

Analysis of `hasEulerPath` algorithm:

- assume that connectivity is already checked
- assume that degree is available via *O(1)* lookup
- single loop over all vertices ⇒ *O(V)*

If find the degree requires iteration over vertices:

- Cost to compute degree of a single vertex is O(V)
- Overall cost is O(V^2)

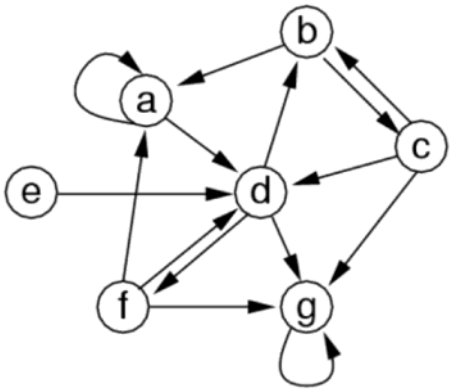
⇒ problem tractable, even for large graphs (unlike Hamiltonian path problem)

Directed Graphs (Digraphs)

In our previous discussion of graphs, an edge indicates a relationship between two vertices and indicates nothing more than a relationship.
In many real-world applications of graphs, edges are **directional** ($v \rightarrow w \neq w \rightarrow v$) and edges have

a **weight** (cost to go from $v \rightarrow w$).

Example of a digraph and an adjacency matric representation:



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

An undirected graph gives a symmetric matrix.
A directed graph gives a non-symmetric matrix.

The maximum no. of edges in a digraph with V vertices: V^2

Terminology of digraphs:
A ***directed path*** is a sequence of $n \geq 2$ vertices, $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$, where $(v_i,v_{i+1}) \in edges(G)$ for all v_i,v_{i+1} in sequence. If $v_1 = v_n$, we have a ***directed cycle***.

The ***degree of vertex*** (*deg(v)*) is the number of edges of the form $(v, \_) \in edges(G)$
The ***indegree*** of vertex ($deg^{-1}(v)$) *is the* number of edges of the form $(\_, v) \in edges(G)$

Reachability: *w* is reachable from *v* if there exists some directed path *v,…,w.*
Strong connectivity is when every vertex is reachable from every other vertex.
A ***directed acyclic graph*** (DAG) is a graph containing no directed cycles.

Potential application areas for digraphs:

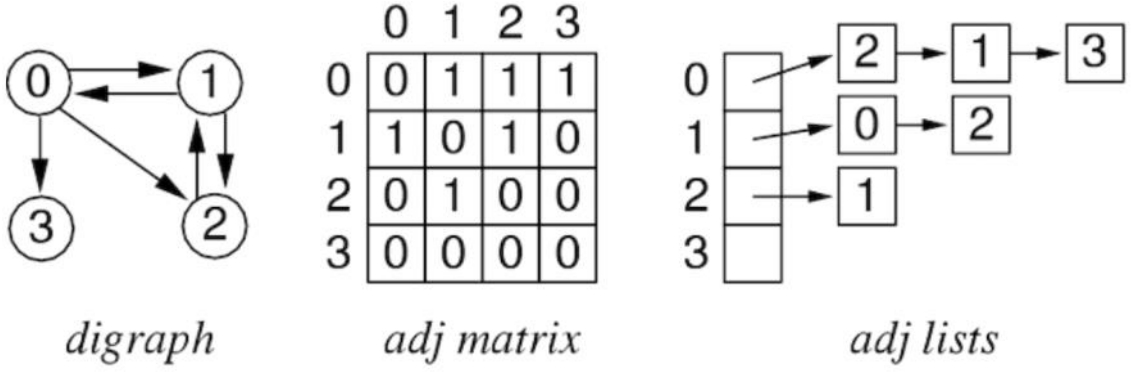
Domain	**Vertex**	**Edge**
Web	web page	hyperlink
scheduling	task	precedence
chess	board position	legal move
science	journal article	citation
dynamic data	malloc'd object	pointer
programs	function	function call
make	file	dependency

Some problems to solve on digraphs include:
- is there a directed path from *s* to *t*? (transitive closure)
- what is the shortest path from *s* to *t*? (shortest path)
- are all vertices mutually reachable? (strong connectivity)
- how to organise a set of tasks? (topological sort)
- which web pages are "important"? (PageRank)
- how to build a web crawler? (graph traversal)

Digraph Representation

The ways of representing digraphs are similar to the set of choices for representing undirected graphs:
- array of edges (directed)
- vertex-indexed adjacency matrix (non-symmetric)
- vertex-indexed adjacency lists

V vertices identified by *0 .. V-1*



Costs of representations: (where degree *deg(v)* = #edges leaving *v*)

	array of edges	adjacency matrix	adjacency list
space usage	*E*	V^2	*V+E*
insert edge	*E*	*1*	*1*
exists edge *(v,w)*?	*E*	*1*	*deg(v)*
get edges leaving *v*	*E*	*V*	*deg(v)*

Overall, adjacency list representation is best as real graphs tend to be sparse (large number of vertices, small average degree *deg(v)*) and algorithms frequently iterate over edges from *v.*

Reachability

Transitive Closure

Given a digraph *G* it is potentially useful to know "*is vertex t reachable from vertex s?*"
Example applications:

- can I complete a schedule from the current state?
- is a malloc'd object being referenced by any pointer?

So how do we compute transitive closure?

One possibility:

- implement it via `hasPath(G,s,t)` (itself implemented by DFS or BFS algorithm)
- feasible if *`reachable(G,s,t)`* is an infrequent operation

What if we have an algorithm that frequently needs to check reachability?
Would be very convenient/efficient to have:

```
reachable(G,s,t):
|  return G.tc[s][t]   // transitive closure matrix
```

Of course, if *V* is very large, then this is not feasible.

Goal: produce a matrix of reachability values:

- If *`tc[s][t]`* is 1, then *t* is reachable from *s*
- If *`tc[s][t]`* is 0, then *t* is not reachable from *s*

Observe that:
∀*i,s,t* ∈ vertices(*G*):

$(s,i) \in edges(G) \wedge (i,t) \in edges(G) \Rightarrow tc[s][t] = 1$

`tc[s][t]`=1 if there is a path from *s* to *t* of length 2 (*s*→*i*→*t*)

If we implement the above as:

```
make tc[][] a copy of edges[][]
for all i∈vertices(G) do
   for all s∈vertices(G) do
      for all t∈vertices(G) do
         if tc[s][i]=1 ∧ tc[i][t]=1 then
```

```
            tc[s][t]=1
         end if
      end for
   end for
end for
```

Then we get an algorithm to convert edges into a *tc*
This is known as ***Warshall's algorithm***.

How it works:
After iteration 1, `tc[s][t]` is 1 if either *s→t* exists or *s→0→t* exists
After iteration 2, `tc[s][t]` is 1 if any of the following exist:
s→t or *s→0→t* or *s→1→t* or *s→0→1→t* or *s→1→0→t*
Etc. so after the
V^{th} iteration, `tc[s][t]` is 1 if there is any directed path in the graph from *s* to *t*

Cost analysis:

- storage: additional V^2 items (each item may be 1 bit)
- computation of transitive closure: V^3
- computation of `reachable()`: *O(1)* after having generated `tc[][]`

Amortisation: would need many calls to `reachable()` to justify other costs

Alternative: use DFS in each call to `reachable()`
Cost analysis:

- storage: cost of queue and set during reachable
- computation of `reachable()`: cost of DFS = $O(V^2)$ (for adjacency matrix)

Digraph Traversal

Same algorithm as for undirected graphs:

```
depthFirst(v):
  1. mark v as visited
  2. for each (v,w)∈edges(G) do
        if w has not been visited then
           depthFirst(w)
```

```
breadth-first(v):
  1. enqueue v
  2. while queue not empty do
        dequeue v
        if v not already visited then
           mark v as visited
           enqueue each vertex w adjacent to v
```

Example of digraph traversal: Web Crawling

Goal: visit every page on the web
Solution: breadth-first search with "implicit" graph

```
webCrawl(startingURL):
|  mark startingURL as alreadySeen
|  enqueue(Q,startingURL)
|  while isEmpty(Q) do
|  |  nextPage=dequeue(Q)
|  |  visit nextPage
|  |  for each hyperLink on nextPage do
|  |  |  if hyperLink not alreadySeen then
|  |  |     mark hyperLink as alreadySeen
|  |  |     enqueue(Q,hyperLink)
|  |  |  end if
```

```
|   |  end for
|  end while
```

visit scans page and collects e.g. keywords and links

Page Rank

Goal: determine which Web pages are "important"

Approach: ignore the page contents and focus on hyperlinks. We treat the Web as graph where a page is a vertex, and a hyperlink is a di-edge. Pages with many incoming hyperlinks are important and we need to compute the "incoming degree" for vertices.

Problem: the Web is a *very* large graph (approx. 10^{14} pages, 10^{15} hyperlinks).
Assume for the moment that we could build a graph.
Most frequent operation in algorithm "Does edge (v,w) exist?"

Simple PageRank algorithm:

```
PageRank(myPage):
|  rank=0
|  for each page in the Web do
|  |  if linkExists(page,myPage) then
|  |     rank=rank+1
|  |  end if
|  end for
```

Note: requires *inbound* link check (not outbound as assumed above for cost of representation)

V = # pages in Web, E = # hyperlinks in Web
Costs for computing PageRank for each representation:

Representation	**linkExists(v,w)**	**Cost**
Adjacency *matrix*	edge[v][w]	*1*
Adjacency *lists*	inLL(list[v],w)	$\approx E/V$

Not feasible ...

- adjacency matrix - $V \cong 10^{14} \Rightarrow$ matrix has 10^{28} cells
- adjacency list - V lists, each with ≅10 hyperlinks $\Rightarrow 10^{15}$ list nodes

So what is a better way to do this?

Approach: the random web surfer - if we randomly follow links in the web, we are more likely to re-discover pages with many inbound links.

```
curr=random page, prev=null
for a long time do
|  if curr not in array ranked[] then
|     rank[curr]=0
|  end if
|  rank[curr]=rank[curr]+1
|  if random(0,100)<85 then            // with 85% chance…
|     prev=curr
|     curr=choose hyperlink from curr  // …crawl on
|  else
|     curr=random page                 // avoid getting stuck
|     prev=null
|  end if
end for
```

Could be accomplished while we crawl web to build search index

Graph Algorithms 2

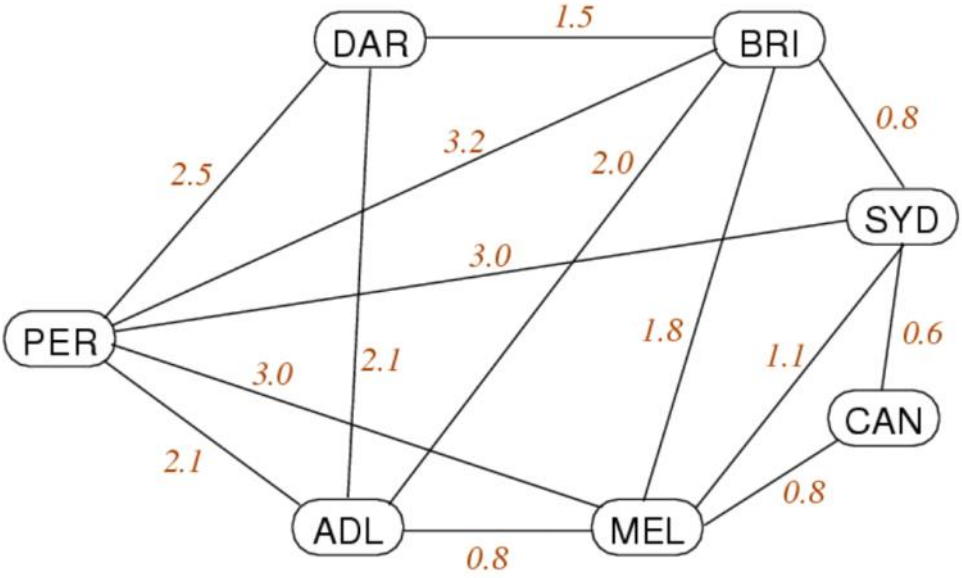
Monday, 3 September 2018 8:42 PM

Weighted Graphs

Graphs so far have considered edges as an associate between two vertices/nodes. There may be a precedence in the association (i.e. it may be directed).

Some applications requires us to consider a *cost* or *weight* of an association, modelled by assigning values to edges (e.g. positive reals). Weights can be used in both directed an undirected graphs.

E.g. major airline flight routes in Australia.



Representation: edge = directed flight; weight = approx. flying time (hours)

Weights lead to minimisation-type questions. e.g.

1. Cheapest way to connect all vertices? - a.k.a ***minimum spanning tree*** problem. This assumes edges are weighted and undirected.
2. Cheapest way to get from A to B? - a.k.a ***shortest path*** problem. This assumes that the edge weights are positive, directed or undirected.

Weighted Graph Representation

Weights can be easily added to:

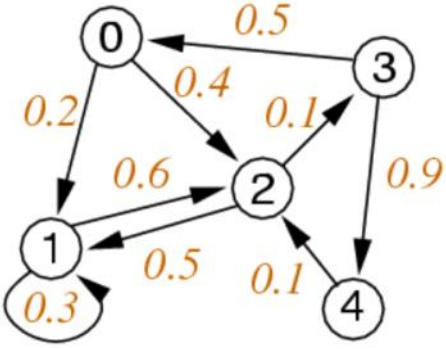
- Adjacency matrix representation (0/1 -> `int` or `float`)
- Adjacency list representation (add `int/float` to list node)

An alternative representation useful in this context is:

- Edge list representation (list of (s,t,w) triples).

All representation work whether the edges are directed or not.

Adjacency matrix representation with weights:

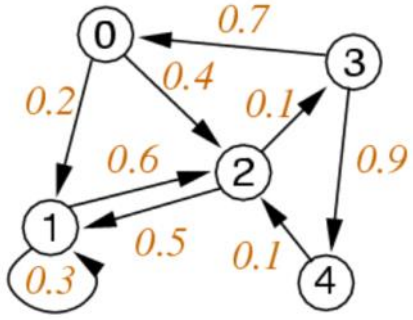


	0	1	2	3	4
0	*	0.2	0.4	*	*
1	*	0.3	0.6	*	*
2	*	0.5	*	0.1	*
3	0.5	*	*	*	0.9
4	*	*	0.1	*	*

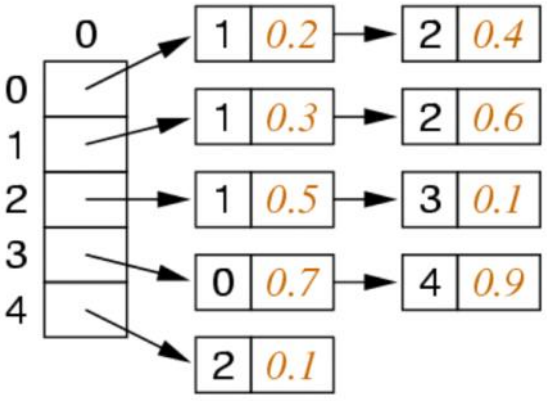
Weighted Digraph *Adjacency Matrix*

Note: need distinguished value to indicate "no edge".

Adjacency lists representation with weights:



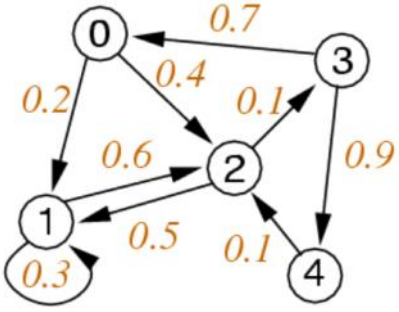
Weighted Digraph



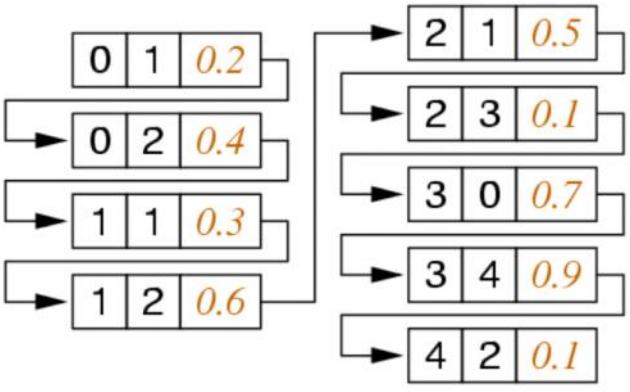
Adjacency Lists

Note: if undirected, each edge appears twice with same weight

Edge array / edge list representation with weights:



Weighted Digraph



Edge List

Note: not very efficient for use in processing algorithms, but does
give a possible representation for min spanning trees or shortest paths

A sample adjacency matrix implementation in C requires minimal changes to the previous graph ADT:

WGraph.h

```
// edges are pairs of vertices (end-points) plus positive weight
typedef struct Edge {
   Vertex v;
   Vertex w;
   int    weight;
```

```
} Edge;
// returns weight, or 0 if vertices are not adjacent
int adjacent(Graph, Vertex, Vertex);
```

WGraph.c

```
typedef struct GraphRep {
   int **edges;  // adjacency matrix storing positive weight
                 // 0 if nodes not adjacent
   int   nV;     // #vertices
   int   nE;     // edges
} GraphRep;
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));
   if (g->edges[e.v][e.w] == 0) {  // edge not in graph
      g->edges[e.v][e.w] = e.weight;
      g->edges[e.w][e.v] = e.weight;
      g->nE++;
   }
}
int adjacent(Graph g, Vertex v, Vertex w) {
   assert(g != NULL && validV(g,v) && validV(g,w));
   return g->edges[v][w];
}
```

Minimum Spanning Tree

Reminder: a ***spanning tree*** *ST* of graph *G=(V,E)* contains all vertices of G and no cycles. A *ST* is a subgraph of *G* (*G'=(V,E')* where *E'⊆E*). *ST* is *connected* and *acyclic.*

Minimum spanning tree *MST* of graph *G* is a spanning tree of *G, where the* sum of edge weights is no larger than any other spanning tree.
Applications: Computer networks, Electrical grids, Transportation networks …

Problem: how to (efficiently) find MST for graph *G*?
NB: MST may not be unique (e.g. all edges have same weight ⇒ every ST is MST)

Brute force solution:

```
findMST(G):
|  Input  graph G
|  Output a minimum spanning tree of G
|
|  bestCost=∞
|  for all spanning trees t of G do
|  |  if cost(t)<bestCost then
|  |     bestTree=t
|  |     bestCost=cost(t)
|  |  end if
|  end for
|  return bestTree
```

This is an example of *generate-and-test* algorithm.
It is not useful because #spanning trees is potentially large (e.g. n^{n-2} for a complete graph with n vertices).

To make explanations of the following algorithms easier, let us assume the edges in G are not directed (finding a MST for digraphs is harder).

Kruskal's Algorithm

One approach to computing MST for graph *G* with *V* nodes:

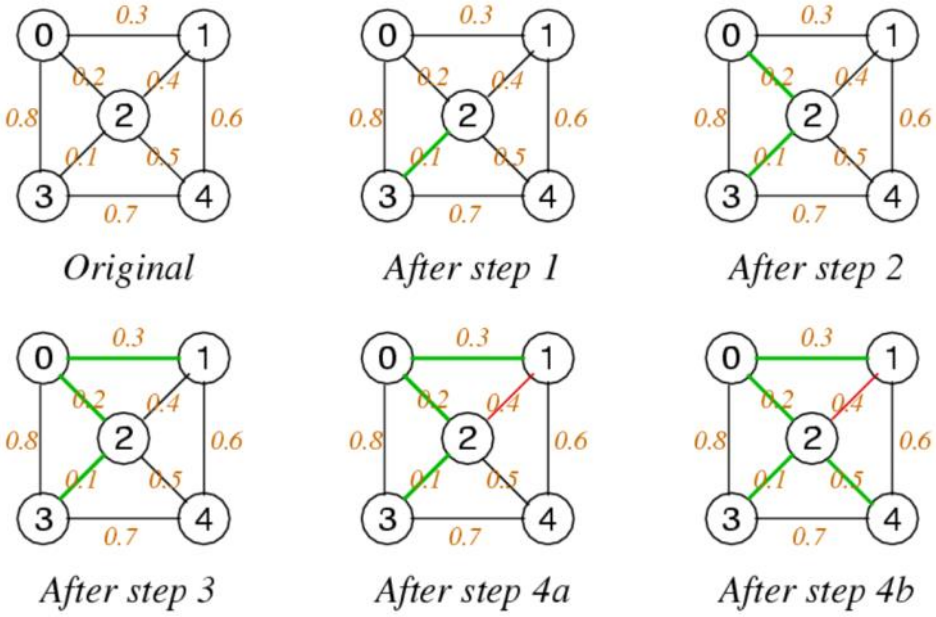
1. start with empty MST
2. consider edges in increasing weight order

    - add edge if it does not form a cycle in MST
3. repeat until *V-1* edges are added

Critical operations:

- iterating over edges in weight order
- checking for cycles in a graph

Execution trace of Kruskal's algorithm:



Pseudocode:

```
KruskalMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  sort edges(G) by weight
|  for each e∈sortedEdgeList do
|  |  MST = MST ∪ {e}
|  |  if MST has a cyle then
|  |     MST = MST \ {e}
|  |  end if
|  |  if MST has n-1 edges then
|  |     return MST
|  |  end if
|  end for
```

Rough time complexity analysis …

- sorting edge list is *O(E·log E)*
- at least *V* iterations over sorted edges
- on each iteration …
    - getting next lowest cost edge is *O(1)*
    - checking whether adding it forms a cycle: cost = ??

Possibilities for cycle checking:

- use DFS … too expensive?
- could use *Union-Find data structure* (see Sedgewick Ch.1)

Prim's Algorithm

Another approach to computing MST for graph *G=(V,E)*:

1. start from any vertex *v* and empty MST
2. choose edge not already in MST to add to MST

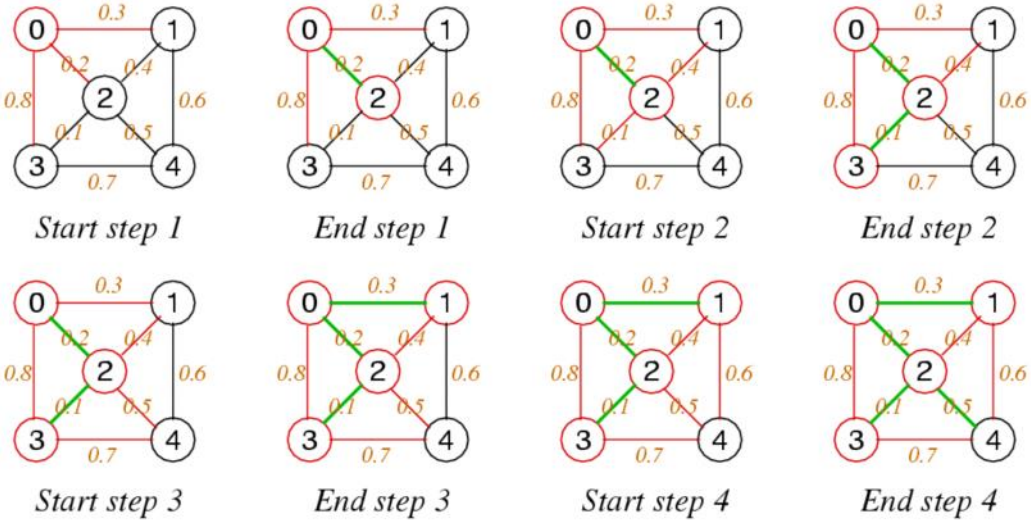
- must be incident on a vertex *s* already connected to *v* in MST
- must be incident on a vertex *t* not already connected to *v* in MST
- must have minimal weight of all such edges

3. repeat until MST covers all vertices

Critical operations:

- checking for vertex being connected in a graph
- finding min weight edge in a set of edges

Execution trace of Prim's algorithm (starting at *s*=0):



Pseudocode:

```
PrimMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  usedV={0}
|  unusedE=edges(g)
|  while |usedV|<n do
|  |  find e=(s,t,w)∈unusedE such that {
|  |     s∈usedV ∧ t∉usedV ∧ w is min weight of all such edges
|  |  }
|  |  MST = MST ∪ {e}
|  |  usedV = usedV ∪ {t}
|  |  unusedE = unusedE \ {e}
|  end while
|  return MST
```

Critical operation: finding best edge

Rough time complexity analysis …

- *V* iterations of outer loop
- in each iteration …
  - find min edge with set of edges is *O(E)* ⇒ *O(V·E)* overall
  - find min edge with *priority queue* is *O(log E)* ⇒ *O(V·log E)* overall

Note:

- Using a *priority queue* gives a variation of DFS (stack) and BFS (queue)graph traversal

Side-track: Priority Queues

Some applications of queues require items to be processed in the order of their "key" rathre than their order of entry (FIFO). Priority Queues (PQueues) provide this via:

- `join`: insert item into Pqueue (replacing enqueue)
- `leave`: remove item with largest key (replacing dequeue)

Comparison of different Priority Queue representations:

	sorted array	unsorted array	sorted list	unsorted list
space usage	*MaxN*	*MaxN*	*O(N)*	*O(N)*
join	*O(N)*	*O(1)*	*O(N)*	*O(1)*
leave	*O(N)*	*O(N)*	*O(1)*	*O(N)*
is empty?	*O(1)*	*O(1)*	*O(1)*	*O(1)*

for a PQueue containing *N* items

Other MST Algorithms

Boruvka's algorithm - complexity *O(E·log V)*
This is the oldest MST algorithm. It start with V separate components and join the components using min cost links. You continue this until you only have a single component.

Karger, Klein, and Tarjan - complexity *O(E)*
This is based on Boruvka's algorithm, but is non-deterministic. It randomly selects subset of edges to consider. Here's a paper describing the algorithm.

Shortest Path

A ***path*** is a sequence of edges in graph G; $p = (v_0,v_1), (v_1,v_2), \ldots, (v_{m-1},v_m)$
cost(path) = sum of edge weights along path

A ***shortest path*** between vertices *s* and *t is* a simple path *p(s,t)* where *s = first(p), t = last(p),* where no other simple path *q(s,t)* has *cost(q) < cost(p)*

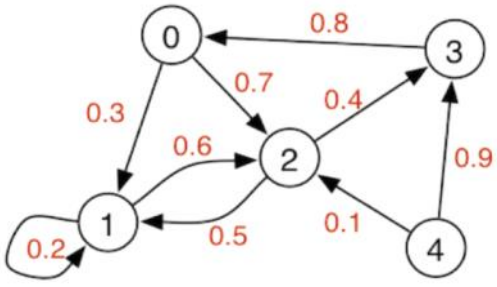
Assumptions: weighted digraph, no negative weights.
Finding shortest path between two given nodes is known as ***source-target*** SP problem.
Some variations are: *single-source* SP, *all-pairs* SP
Applications: navigation, routing in data networks, …

Single-source Shortest Path (SSSP)

Given: weighted digraph *G*, source vertex *s*
Result: shortest paths from *s* to all other vertices

- `dist[]` *V*-indexed array of cost of shortest path from *s*
- `pred[]` *V*-indexed array of the predecessor vertex in the shortest path from *s*

Example:

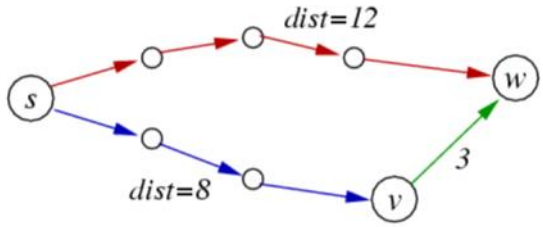


	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
pred	-	0	0	2	-

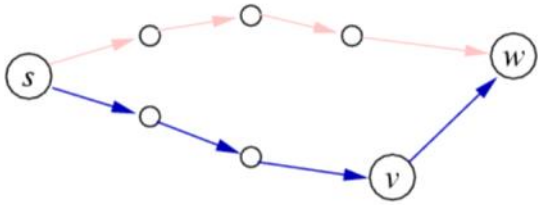
Shortest paths from s=0

Edge Relaxation

Assume: `dist[]` and `pred[]` as above (but containing data for shortest paths *discovered so far*)
`dist[v]` is length of shortest known path from *s* to *v*
`dist[w]` is length of shortest known path from *s* to *w*
Relaxation updates data for *w* if we find a shorter path from *s* to *w*:



dist[v]=8, dist[w]=12
pred[v]=?, pred[w]=?



dist[v]=8, dist[w]=11
pred[v]=?, pred[w]=v

Relaxation along edge `e = (v, w, weight)`:

```
if  dist[v]+weight < dist[w]  then
    update dist[w]:=dist[v]+weight and pred[w]:=v
```

Dijkstra's Algorithms

One approach to solving single-source shortest path …
Data: *G*, *s*, `dist[]`, `pred[]` and *vSet*: set of vertices whose shortest path from *s* is unknown
Algorithm:

```
dist[]  // array of cost of shortest path from s
pred[]  // array of predecessor in shortest path from s
dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise dist[] to all ∞, except dist[source]=0
|  initialise pred[] to all -1
|  vSet=all vertices of G
|  while vSet≠∅ do
|  |  find s∈vSet with minimum dist[s]
|  |  for each (s,t,w)∈edges(G) do
|  |     relax along (s,t,w)
|  |  end for
|  |  vSet=vSet\{s}
|  end while
```

Why Dijkstra's algorithm is correct:
Hypothesis.
(a) For visited *s* - *dist[s]* is shortest distance from source
(b) For unvisited *t* - *dist[t]* is shortest distance from source *via visited nodes*

Proof.
Base case: no visited nodes, *dist[source]=0*, *dist[s]=∞* for all other nodes
Induction step:

1. If *s* is unvisited node with minimum *dist[s]*, then *dist[s]* is shortest distance from source to *s*:
   - if ∃ shorter path via only visited nodes, then *dist[s]* would have been updated when processing the predecessor of *s* on this path
   - if ∃ shorter path via an unvisited node *u*, then *dist[u]<dist[s]*, which is impossible if *s* has min distance of all unvisited nodes
2. This implies that (a) holds for *s* after processing *s*
3. (b) still holds for all unvisited nodes *t* after processing *s*:
   - if ∃ shorter path via *s* we would have just updated *dist[t]*
   - if ∃ shorter path without *s* we would have found it previously

Time complexity analysis:
Each edge needs to be considered once ⇒ *O(E)*.
Outer loop has *O(V)* iterations.
Implementing "**find** s∈vSet **with** minimum dist[s]"

1. try all s∈vSet ⇒ cost = $O(V)$ ⇒ overall cost = $O(E + V^2) = O(V^2)$
2. using a PQueue to implement extracting minimum
    - can improve overall cost to $O(E + V \cdot \log V)$ (for best-known implementation)

Tree, BSTs, Balanced BSTs

Thursday, 6 September 2018 4:08 PM

Searching

Searching is an extremely common application in computing.
Given a (large) collection of *items* and a *key* value, we find the item(s) in the collection containing that key:

- item = (key, val_1, val_2, …) (i.e. a structured data type)
- key = value used to distinguish items (e.g. student ID)

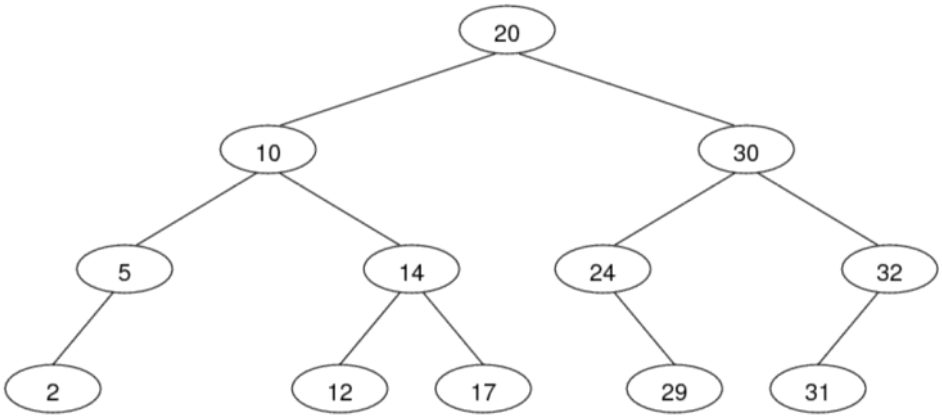
Applications: Google, databases, ...

Since searching is a very important/frequent operation, many approaches have been developed to do it.
We have linear structures: arrays, linked lists, files. Arrays have random access., while lists, and files have sequential access.
Cost of searching:

	Array	List	File
Unsorted	O(n) (linear scan)	O(n) (linear scan)	O(n) (linear scan)
Sorted	O(log n) (binary search)	O(n) (linear scan)	O(log n) (*seek*, *seek*>, …)

- *O(n)* - linear scan (search technique of last resort)
- *O(log n)* - binary search, *search trees* (trees also have other uses)

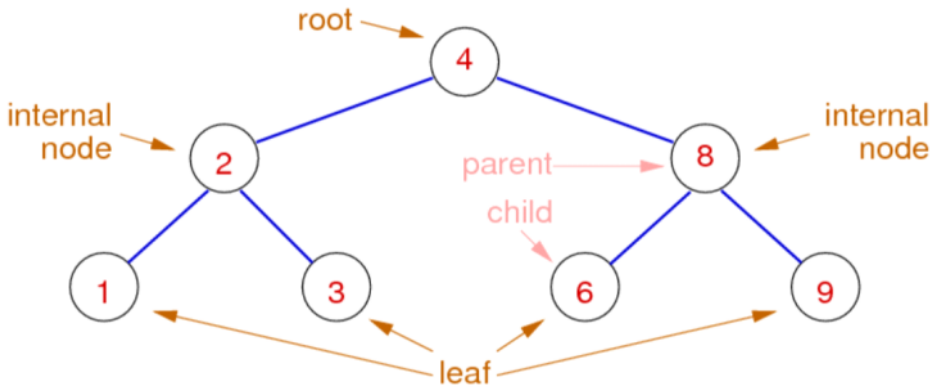
Maintaining order in sorted arrays and files is a costly operation. ***Search trees*** are as efficient to search and more efficient to maintain.
Example: the following tree corresponds to the sorted array
[2,5,10,12,14,17,20,24,29,30,31,32]:



Tree Data Structures

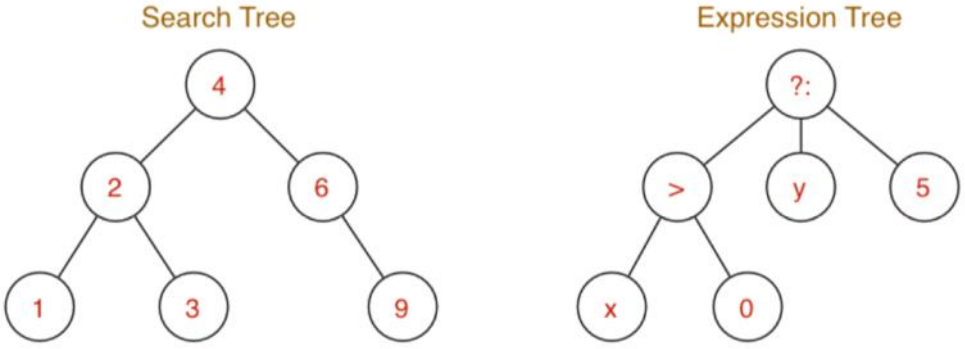
Trees

Trees are connected graphs consisting of notes and edges (called *links*), with no cycles (no "up-links"). Each node contains a ***data*** value (or a key and data), and each node has ***links*** to $\leq k$ other child nodes (the diagram below has $k=2$).

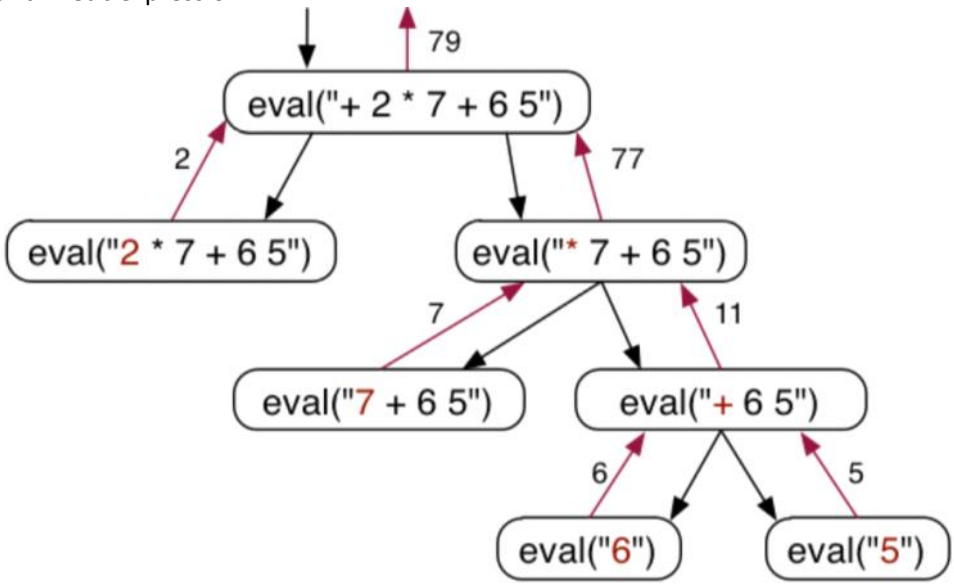


Trees are used in many contexts. E.g. representing hierarch data instructions (e.g. expressions) and

efficient searching (e.g. sets, symbol tables, …)

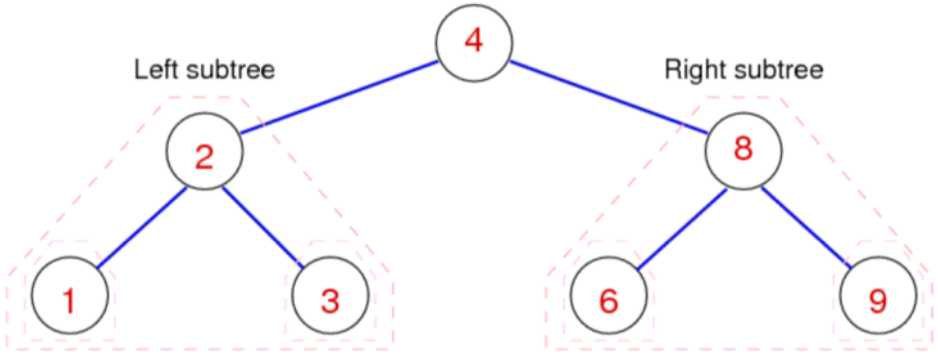


Trees can be used as a data structure, but also for illustration. E.g. showing evaluation for a prefix arithmetic expression.



Binary trees (*k=2* children per node) can be defined recursively, as follows:
A *binary tree* is either

- empty (contains no nodes)
- consists of a *node*, with two *subtrees.* The node contains a value and the left and right subtrees are *binary trees.*



Other special kinds of tree

- ***m-ary tree***: each internal node has exactly *m* children
- ***Ordered tree***: all left values < root, all right values > root
- ***Balanced tree***: has ≅minimal height for a given number of nodes
- ***Degenerate tree***: has ≅maximal height for a given number of nodes

Search Trees

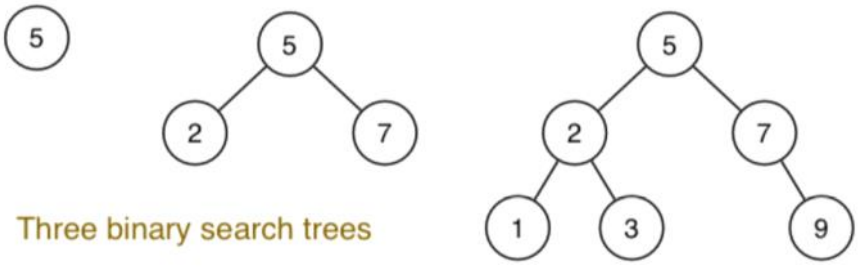
Binary Search Trees

Binary search trees (or *BSTs*) have the characteristic properties

- each node is the root of 0, 1 or 2 subtrees
- all values in any left subtree are less than root
- all values in any right subtree are greater than root
- these properties applies over all nodes in the tree

(perfectly) *balanced trees* have the properties

- #nodes in left subtree = #nodes in right subtree
- this property applies over all nodes in the tree



Operations on BSTs:

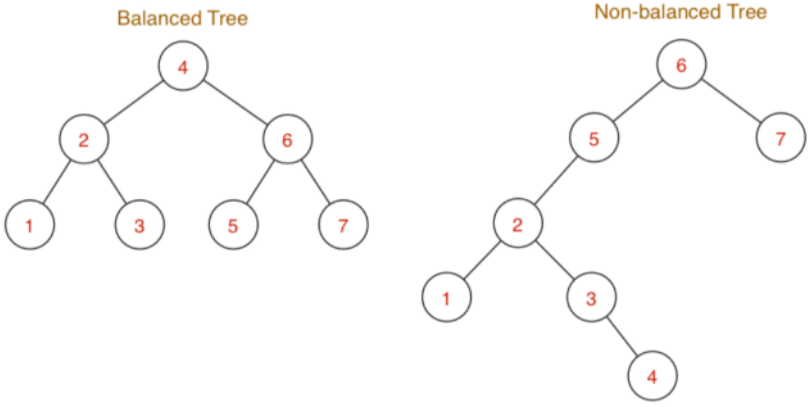
- *insert*(Tree,Item) - add new item to tree via key
- *delete*(Tree,Key) - remove item with specified key from tree
- *search*(Tree,Key) - find item containing key in tree
- plus, "bookkeeping" - new(), free(), show(), …

Notes:

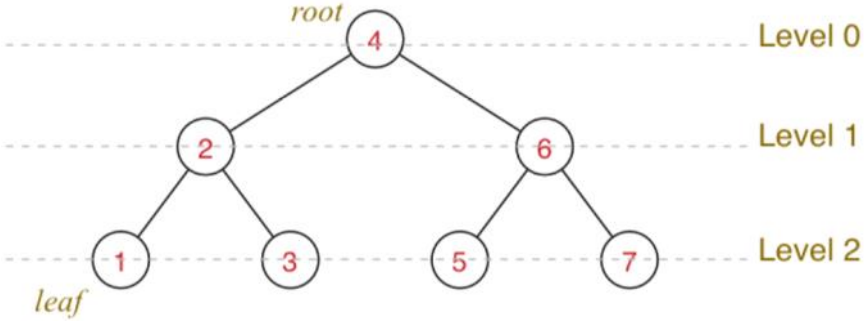
- nodes contain Items; we just show Item.key
- keys are unique (not technically necessary)

Examples of binary search trees:

Shape of tree is determined by order of insertion.



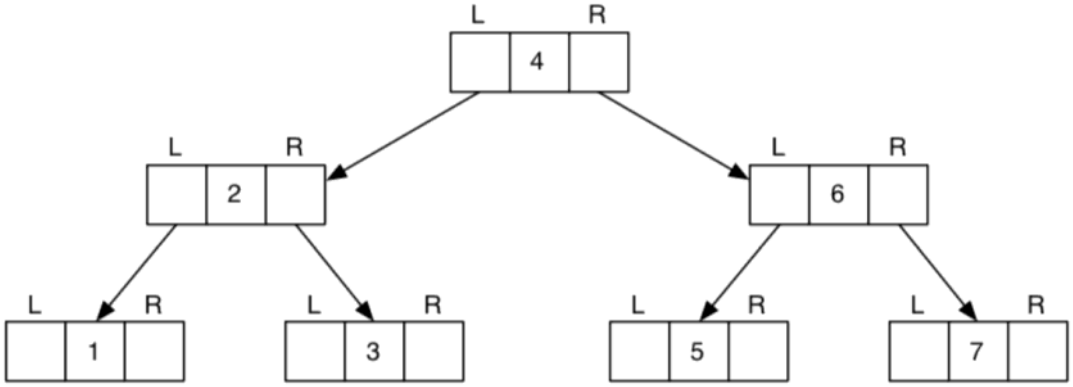
The ***level*** of a node is the path length from root to node.
The ***height*** (or ***depth***) of a tree is the max path length from the root to a leaf.



Height-balanced tree are trees, where ∀ nodes, height(left subtree) = height(right subtree)
The time complexity of tree algorithms is typically *O(height).*

Representing BSTs

Binary trees are typically represented by node structures, containing a value, and pointers to child nodes. Most tree algorithms move *down* the tree. If upward movement is needed, we can just a dd a pointer to the parent.

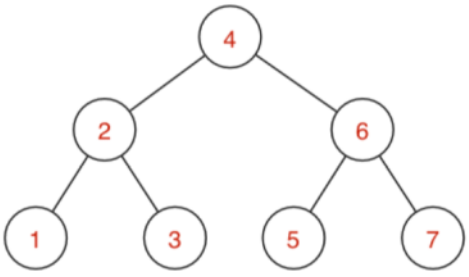


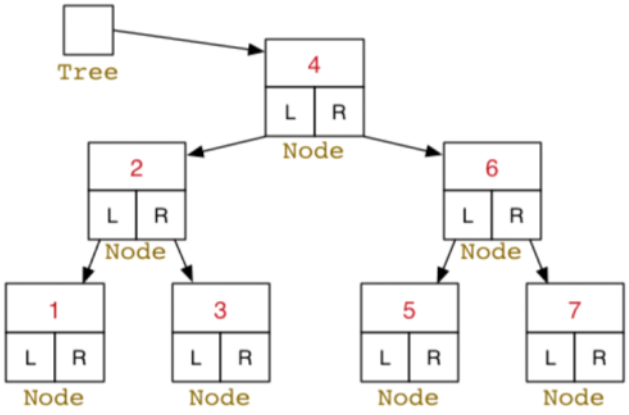
A typical data structure for trees:

```
// a Tree is represented by a pointer to its root node
typedef struct Node *Tree;
// a Node contains its data, plus left and right subtrees
typedef struct Node {
   int  data;
   Tree left, right;
} Node;
// some macros that we will frequently use
#define data(tree)  ((tree)->data)
#define left(tree)  ((tree)->left)
#define right(tree) ((tree)->right)
```

We ignore items ⇒ data in Node is just a key

Abstract vs concrete data:





Tree Algorithms

Searching in BSTs

Most tree algorithms are best described recursively:

```
TreeSearch(tree,item):
|  Input  tree, item
|  Output true if item found in tree, false otherwise
|
|  if tree is empty then
|     return false
|  else if item < data(tree) then
|     return TreeSearch(left(tree),item)
|  else if item > data(tree) then
|     return TreeSearch(right(tree),item)
|  else
|    return true
|  end if
```

Insertion into BSTs

Insert an item into appropriate subtree:

```
insertAtLeaf(tree,item):
|  Input  tree, item
```

```
|  Output tree with item inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item < data(tree) then
|     return insertAtLeaf(left(tree),item)
|  else if item > data(tree) then
|     return insertAtLeaf(right(tree),item)
|  else
|     return tree
|  end if
```

Tree Traversal

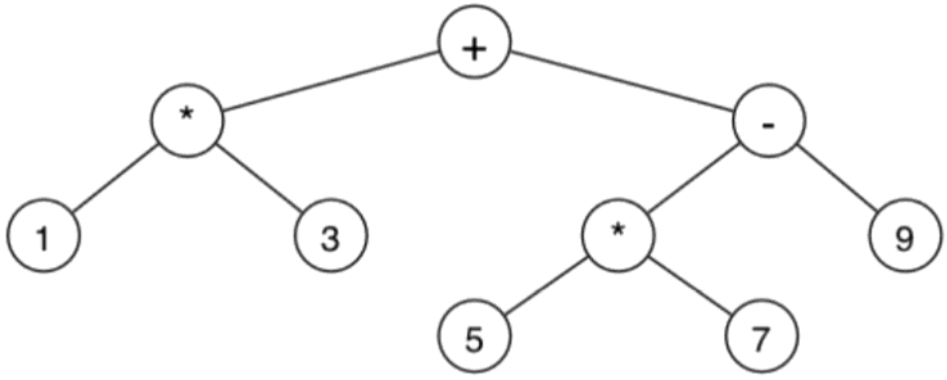
Iteration (traversal) on

- `Lists` - visit each value, from first to last
- `Graphs` - visit each vertex, order determined by DFS/BFS/...

For binary Trees, several well-defined visiting orders exist:

- *preorder* (NLR) - visit root, then left subtree, then right subtree
- *inorder* (LNR) - visit left subtree, then root, then right subtree
- *postorder* (LRN) - visit left subtree, then right subtree, then root
- *level-order* - visit root, then all its children, then all their children

Consider visiting an expression tree like:



NLR: + * 1 3 - * 5 7 9 (prefix-order: useful for building tree)

LNR: 1 * 3 + 5 * 7 - 9 (infix-order: "natural" order)

LRN: 1 3 * 5 7 * 9 - + (postfix-order: useful for evaluation)

Level: + * - 1 3 * 9 5 7 (level-order: useful for printing tree)

Joining Two Trees

Joining two trees is an auxiliary tree operation. So far our tree operations have involved just one tree.
An operation on two trees: t = joinTrees(t_1,t_2).
Pre-conditions:

- taking two BSTs; returning a single BST
- `max(key(`t_1`)) < min(key(`t_2`))`

Post-conditions:

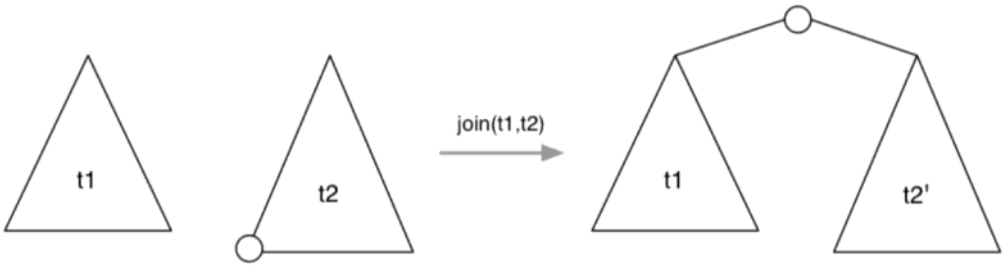
- the result is a BST (i.e. fully ordered)
- containing all items from t_1 and t_2

The method for performing tree-join is:

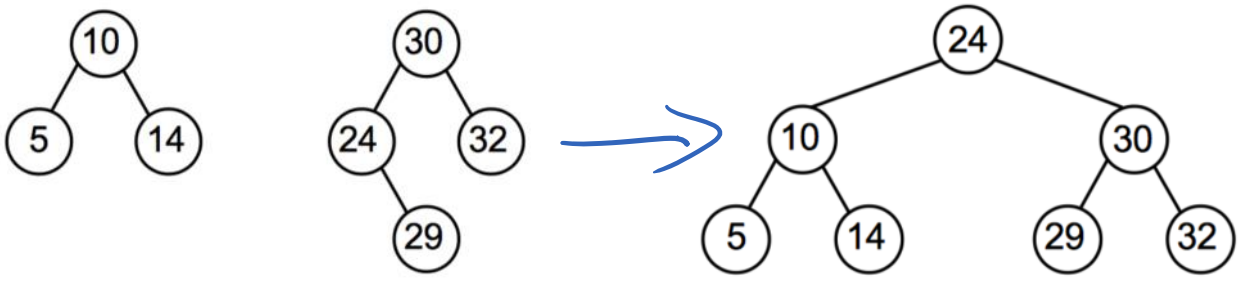
1. Find the min node in the right subtree (t_2)
2. Replace the min node by its right subtree
3. Elevate the min node to be the new root of both trees

The advantage of this method is that we don't increase the height of the tree significantly
$x \leq height(t) \leq x+1$, where $x = max(height(t_1), height(t_2))$
An alternative method would be to make t_1 the left subtree of the min node of t_2 but this would result in a deeper subtree.

Joining two trees:



Note: t2' may be less deep than t2



Implementation of tree-join:

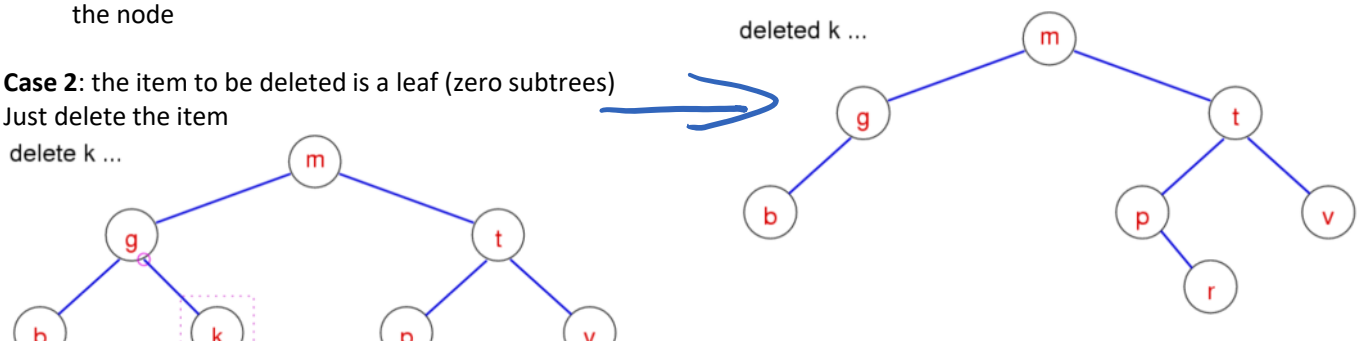
```
joinTrees(t1,t2):
|  Input  trees t1,t2
|  Output t1 and t2 joined together
|
|  if t1 is empty then return t2
|  else if t2 is empty then return t1
|  else
|  |  curr=t2, parent=NULL
|  |  while left(curr) is not empty do     // find min element in t2
|  |     parent=curr
|  |     curr=left(curr)
|  |  end while
|  |  if parent≠NULL then
|  |     left(parent)=right(curr)  // unlink min element from parent
|  |     right(curr)=t2
|  |  end if
|  |  left(curr)=t1
|  |  return curr                  // curr is the new root
|  end if
```

Deleting from BSTs

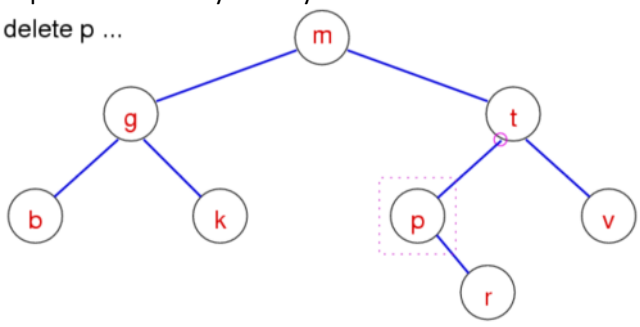
When deleting a node from a binary tree there are four cases to consider:

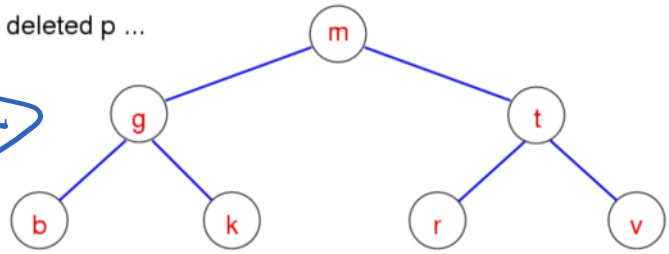
- An **empty** tree - the new tree is also empty
- A node with **zero subtrees** - unlink node from parent
- A node with **one subtree** - the node is replaced by child
- A node with **two subtrees** - replace the node with its successor, or join the two subtrees and replace the node

Case 2: the item to be deleted is a leaf (zero subtrees)
Just delete the item

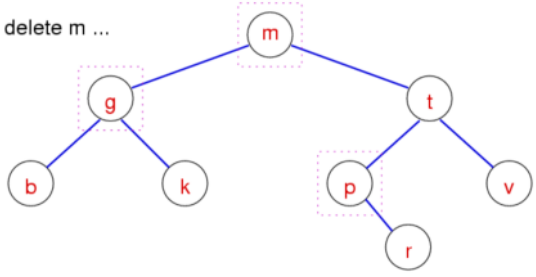


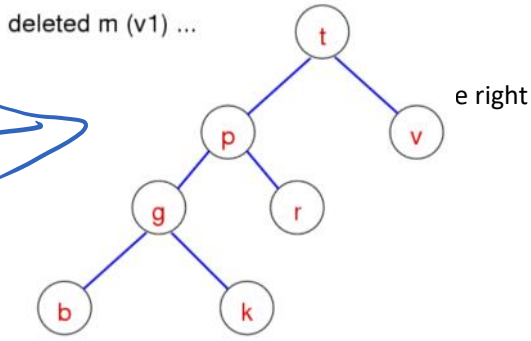
Case 3: the item to be deleted has one subtree
Replace the item by its only subtree





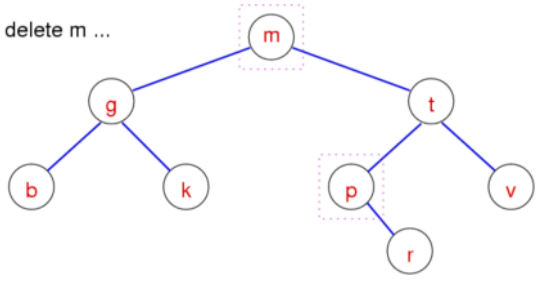
Case 4: the item to be deleted has two subtrees
Version 1: the right child becomes the new root, and attach

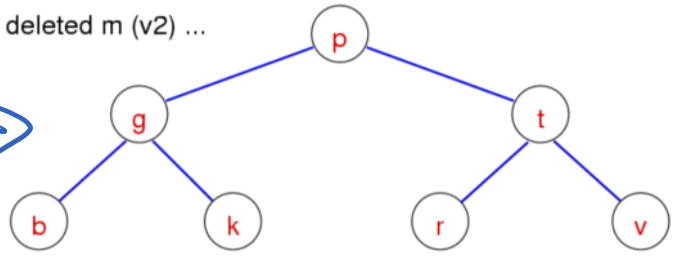




e right subtree

Version 2: join the left and right subtree





Pseudocode for version 2:

```
TreeDelete(t,item):
|  Input  tree t, item
|  Output t with item deleted
|
|  if t is not empty then
|  |  if item < data(t) then
|  |     left(t)=TreeDelete(left(t),item)
|  |  else if item > data(t) then
|  |     right(t)=TreeDelete(right(t),item)
|  |  else
|  |  |  if left(t) and right(t) are empty then
|  |  |     new=empty tree
|  |  |  else if left(t) is empty then
|  |  |     new=right(t)
|  |  |  else if right(t) is empty then
|  |  |     new=left(t)
|  |  |  else
|  |  |     new=joinTrees(left(t),right(t))
|  |  |  end if
|  |  |  free memory allocated for t
|  |  |  t=new
```

```
|   |  end if
|   end if
|   return t
```

Balanced BSTs

Balanced Binary Search Trees

Goal: build binary search trees which have

- minimum height ⇒ minimum worst case search cost

Perfectly balanced tree with *N* nodes has

- `abs(#nodes(LeftSubtree) - #nodes(RightSubtree)) < 2`, for every node
- height of $log_2 N$ ⇒ worst case search *O(log N)*

Three *strategies* to improving worst case search in BSTs:

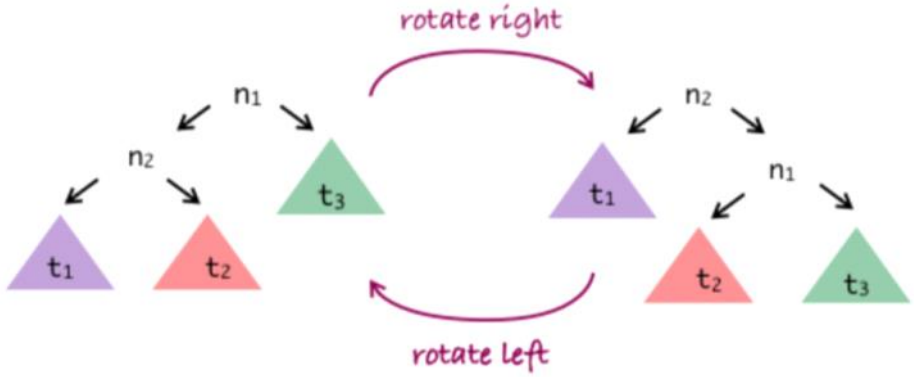
- ***randomise*** — reduce chance of worst-case scenario occurring
- ***amortise*** — do more work at insertion to make search faster
- ***optimise*** — implement all operations with performance bounds

Operations for Rebalancing

To assist with rebalancing, we consider new operations:
Left rotation - move the right child to root; rearrange links to retain order
Right rotation - move left child to root; rearrange links to retain order
Insertion at root - each new item is added as the new root node

Tree Rotation

In the tree below: $t_1 < n_2 < t_2 < n_1 < t_3$



Method for rotating tree T right:

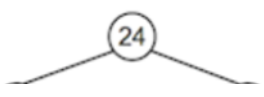
- N_1 is current root; N_2 is root of N_1's left subtree
- N_1 gets new left subtree, which is N_2's right subtree
- N_1 becomes root of N_2's new right subtree
- N_2 becomes new root

Left rotation: swap left/right in the above.
Cost of tree rotation: *O(1)*
Algorithm for rotations:

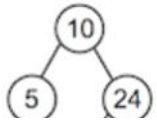
```
rotateRight(n1):
|  Input  tree n1
|  Output n1 rotated to the right
|
|  if n1 is empty ∨ left(n1) is empty
then
|     return n1
|  end if
|  n2=left(n1)
|  left(n1)=right(n2)
|  right(n2)=n1
|  return n2
```

```
rotateLeft(n2):
|  Input  tree n2
|  Output n2 rotated to the left
|
|  if n2 is empty ∨ right(n2) is empty
then
|     return n2
|  end if
|  n1=right(n2)
|  right(n2)=left(n1)
|  left(n1)=n2
|  return n1
```

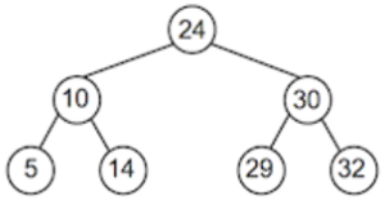
Example:
Consider the tree `t`:

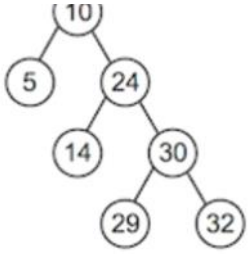


After `rotateRight(t)`:



Consider the tree t:





Insertion at Root

Previous description of binary search trees inserted at leaves. A different approach would be to insert a new item at the root.

Potential disadvantages:

- Large-scale rearrangement of the tree for each insert

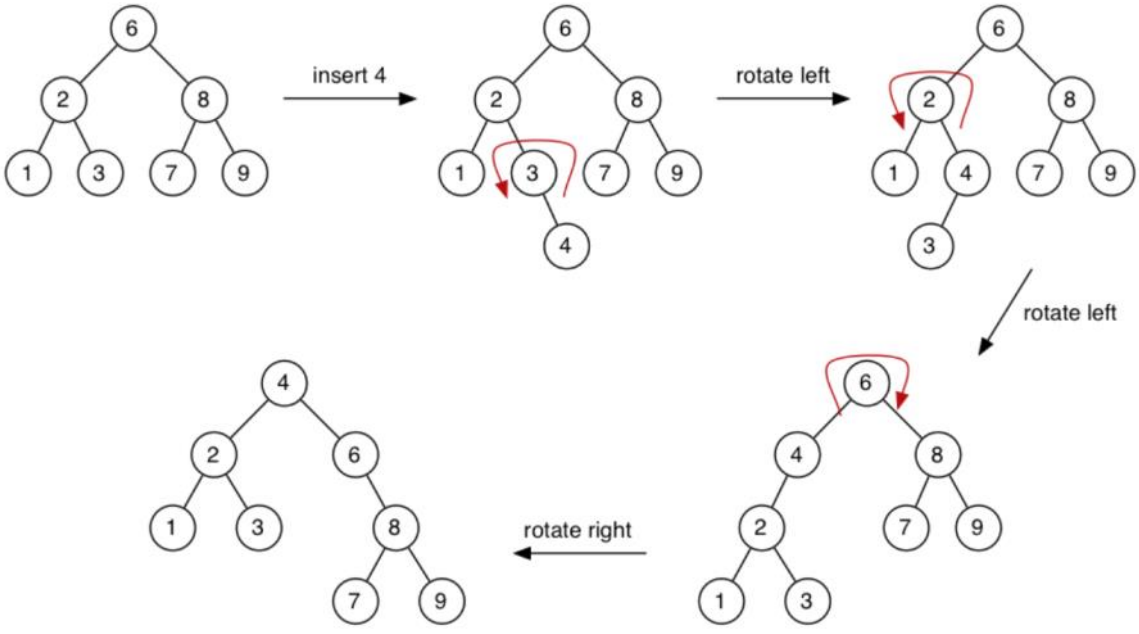
Potential advantages:

- Recently-inserted items are close to the root
- Low cost if recent items are more likely to be searched

Method for inserting at the root:

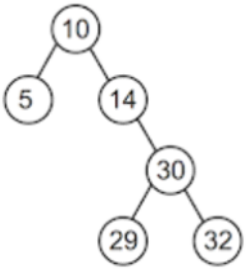
1. Base case:
    - the tree is empty; make a new node and make it the root
2. Recursive case:
    - insert a new node as the root of an appropriate subtree
    - lift the new node to the root by rotation

Example:

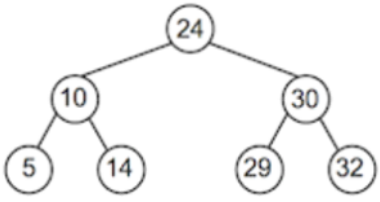


Example:

Consider the tree `t`:



After `insertAtRoot(t, 24)`:



Analysis of insertion-at-root:

- same complexity as for insertion-at-leaf: *O(height)*
- tendency to be balanced, but no balance guarantee
- benefit comes in searching
    - for some applications, search favours recently-added items
    - insertion-at-root ensures these are close to root

- could even consider "move to root when found"
  - effectively provides "self-tuning" search tree

Rebalancing Trees

An approach to balancing trees:
Insert your node into the leaves as with a simple BST, then periodically rebalance the tree.

How frequently should we rebalance the tree? After every *k* insertions for some integer *k*.

```
NewTreeInsert(tree,item):
|  Input  tree, item
|  Output tree with item randomly inserted
|
|  t=insertAtLeaf(tree,item)
|  if #nodes(t) mod k = 0 then
|     t=rebalance(t)
|  end if
|  return t
```

E.g. rebalance after every 20 insertions ⇒ choose *k=20*
Note: To do this efficiently we would need to change tree data structure and basic operations:

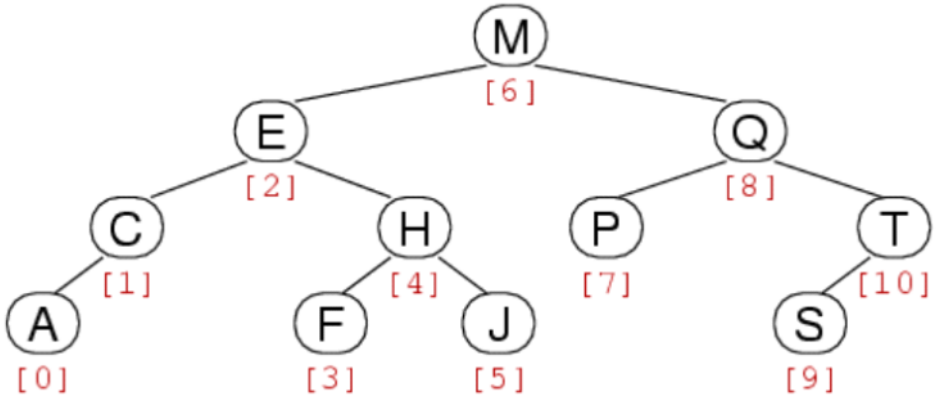
```
typedef struct Node {
   int  data;
   int  nnodes;      // #nodes in my tree
   Tree left, right; // subtrees
} Node;
```

How do we rebalance trees?
We recursively mode the median item to be the root of each respective subtree.

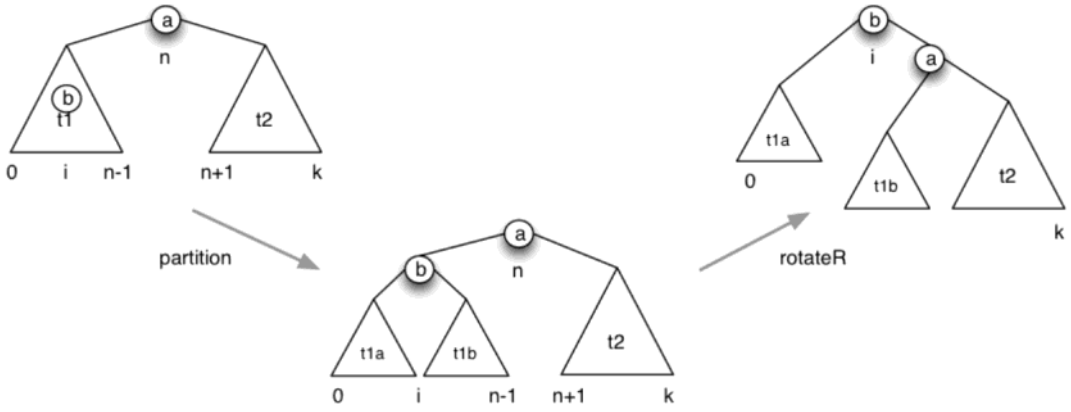
Implementation of rebalance:

```
rebalance(t):
|  Input  tree t with n nodes
|  Output t rebalanced
|
|  if n≥3 then
|  |  t=partition(t,floor(n/2))
|  |  left(t)=rebalance(left(t))
|  |  right(t)=rebalance(right(t))
|  end if
|  return t
```

New operation on trees:
partition(tree,i): re-arrange tree so that element with index *i* becomes root



For tree with *N* nodes, indices are *0..N-1*
Partition: moves the i^{th} node to the root.

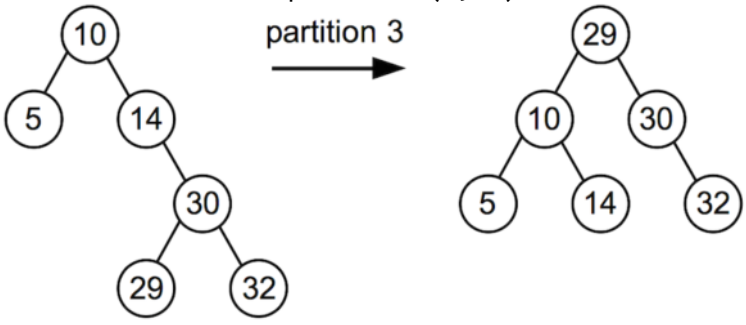


Implementation of the partition operation:

```
partition(tree,i):
|  Input  tree with n nodes, index i
|  Output tree with i^th item moved to the root
|
|  m=#nodes(left(tree))
|  if i < m then
|     left(tree)=partition(left(tree),i)
|     tree=rotateRight(tree)
|  else if i > m then
|     right(tree)=partition(right(tree),i-m-1)
|     tree=rotateLeft(tree)
|  end if
|  return tree
```

Note: `size(tree) = n, size(left(tree)) = m, size(right(tree)) = n-m-1` (-1 to ignore current node)

Example:
Consider the tree `t` after `partition(t, 3)`:



Analysis of rebalancing: visits every node ⇒*O(N)*
Cost means not feasible to rebalance after each insertion.
When to rebalance? … Some possibilities:

- after every *k* insertions
- whenever "imbalance" exceeds threshold

Either way, we tolerate worse search performance for periods of time.
Does it solve the problem? … Not completely ⇒ Solution: real balanced trees (next week)

Applications of Binary Search Trees: Sets

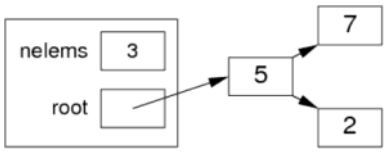
Trees provide efficient search. Sets require efficient search to find where to insert and delete and to test for membership. Therefore it is logical to implement a set ADT via BSTree

Assuming we have Tree implementation

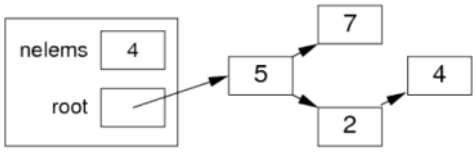
- which precludes duplicate key values
- which implements insertion, search, deletion

then Set implementation is

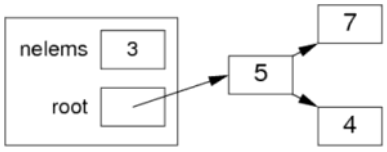
- `SetInsert(Set,Item)` ≡ `TreeInsert(Tree,Item)`
- `SetDelete(Set,Item)` ≡ `TreeDelete(Tree,Item.Key)`
- `SetMember(Set,Item)` ≡ `TreeSearch(Tree,Item.Key)`



After SetInsert(s,4):



After SetDelete(s,2):



Concrete representation:

```
#include <BSTree.h>

typedef struct SetRep {
   int   nelems;
   Tree  root;
} SetRep;

Set newSet() {
   Set S = malloc(sizeof(SetRep));
   assert(S != NULL);
   S->nelems = 0;
   S->root = newTree();
   return S;
}
```

Search Tree Algorithms 2

Monday, 17 September 2018 3:42 PM

Tree Review

Binary search trees:

- data structures designed for *O(log n)* search
- consist of nodes containing item (incl. key) and two links
- can be viewed as recursive data structure (subtrees)
- have overall ordering (`data(Left) < root < data(Right)`)
- insert new nodes as leaves (or as root), delete from anywhere
- have structure determined by insertion order *(worst: O(n))*
- operations: insert, delete, search, rotate, rebalance, …

Randomised BST Insertion

Effects of order of insertion of BST shape:

- best case (for at-leaf insertion): keys inserted in pre-order (median key first, then median of lower half, median of upper half, etc)
- worst case: keys inserted in ascending/descending order
- average case: keys inserted in *random* order $\Rightarrow O(\log_2 n)$

Tree ADTS have no control over the order that the keys are supplied. If we introduce some *randomness* it is possible that this randomness helps balance the tree.

Sidetrack: Random Numbers

A computer cannot pick a number at random. Software can only produce ***pseudo random numbers***. A pseudo random number is one that is unpredictable (although it may appear unpredictable). The implementation may deviate from the expected theoretical behaviour.

The most widely-used technique is called the ***Linear Congruential Generator*** *(LCG)*
It uses a recurrence relation:

- $X_{n+1} = (a \cdot X_n + c) \bmod m$, where:
  - m is the "modulus"
  - $a, 0 < a < m$ is the "multiplier"
  - $c, 0 \leq c \leq m$ is the "increment"
  - X_0 is the "*seed*"
- if $c=0$ it is called a *multiplicative congruential generator*

LCG is not good for applications that need extremely high-quality random numbers

- the period length is too short (length of the sequence at which point it repeats itself)
- a short period means the numbers are correlated

Trivial example:

- for simplicity assume c=0
- so the formula is $X_{n+1} = a \cdot X_n \bmod m$
- try a=11=X_0, m=31, which generates the sequence:

```
11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25,
27, 18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28, 29, 9, 6, 4, 13, 19, 23, 5,
24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27, 18, 12, 8, 26, 7, 15, 10, 17, 1,
11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27,
18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16,
21, 14, 30, 20, 3, 2, 22, 25, 27, 18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28,
29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27, 18,
12, 8, 26, 7, 15, 10, 17, 1, ...
```

- all the integers from 1 to 30 are here

Another trivial example:

- again let c=0
- try a=12=X_0 and m=30
  - that is, $X_{n+1} = 12 \cdot X_n \bmod 30$
  - which generates the sequence:

12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6,
12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, ...

- notice the period length … clearly a terrible sequence

It is a complex task to pick good numbers. A bit of history:
Lewis, Goodman and Miller (1969) suggested

- $X_{n+1} = 7^5 \cdot X_n \bmod (2^{31}-1)$
- note:
  - 7^5 is 16807
  - $2^{31}-1$ is 2147483674
  - $X_0 = 0$ is not a good seed value

Most compilers use LCG-based algorithms that are slightly more involved; seewww.mscs.dal.ca/~selinger/random/ for details (including a short C program that produces the exact same pseudo-random numbers as gcc for any given seed value)

- Two functions are required:

```
srandom(int seed) // sets its argument as the seed
random() // uses a LCG technique to generate random
         // numbers in the range 0..RAND_MAX
```

  where the constant RAND_MAX is defined in stdlib.h
  (depends on the computer: on the CSE network, RAND_MAX = 2147483647)
- The period length of this random number generator is very large
  approximately $16 \cdot ((2^{31}) - 1)$

To convert the return value of random() to a number between 0 .. RANGE

- compute the remainder after division by RANGE+1

Using the remainder to compute a random number is not the best way:

- can generate a 'better' random number by using a more complex division
- but good enough for most purposes

Some applications require more sophisticated, *cryptographically secure* pseudo random numbers

Seeding
There is one significant problem:

- every time you run a program with the same seed, you get exactly the same sequence of 'random' numbers (why?)

To vary the output, can give the random seeder a starting point that varies with time

- an example of such a starting point is the current time, ***time(NULL)***
  (NB: this is different from the UNIX command time, used to measure program running time)

```
#include <time.h>
time(NULL) // returns the time as the number of seconds
           // since the Epoch, 1970-01-01 00:00:00 +0000
// time(NULL) on October 10th, 2017, 12:59pm was 1507600763
// time(NULL) about a minute later was 1507600825
```

Randomised BST Insertion

Approach: normally do leaf insert, randomly do root insert.

```
insertRandom(tree,item)
|  Input  tree, item
|  Output tree with item randomly inserted
|
|  if tree is empty then
|     return new node containing item
|  end if
|
|  if random() mod q < p then
|     return insertAtRoot(tree,item)
|  else
|     return insertAtLeaf(tree,item)
```

```
|  end if
```

E.g. 30% chance ⇒ choose *p=3, q=10*

Cost analysis:

- similar to cost for inserting keys in random order: $O(\log_2 n)$
- does not rely on keys being supplied in random order

Approach can also be applied to deletion:

- standard method promotes inorder successor to root
- for the randomised method …
  - promote inorder successor from right subtree, OR
  - promote inorder predecessor from left subtree

Splay Trees

Splay trees are a kind of "self-balancing" tree. Splay tree insertion modifies the insertion-at-root method by:

- considering parent-child-grandchild (three level analysis)
- performing double-rotations based on p-c-g orientation

The idea: appropriate double-rotations to improve tree balance

Splay trees provide ***fast access*** to recently searched items, with the idea that a *recently searched item is more likely to be searched again*.

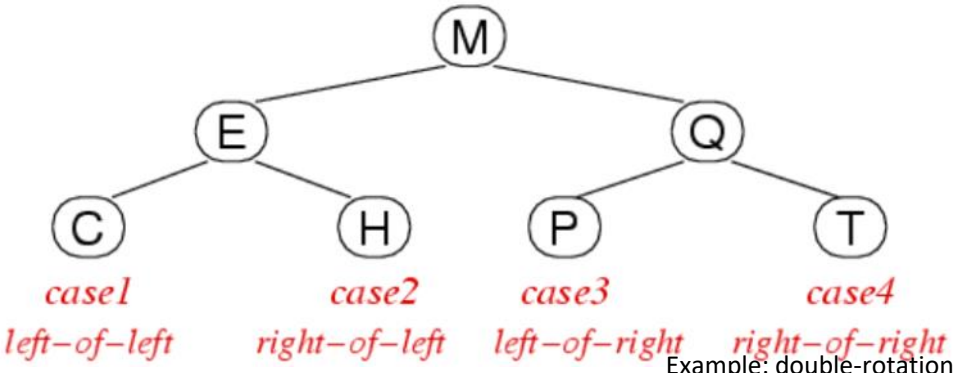
Splay tree implementations also do ***rotation-in-search***:

When it searches for an item, it brings that node to the root.

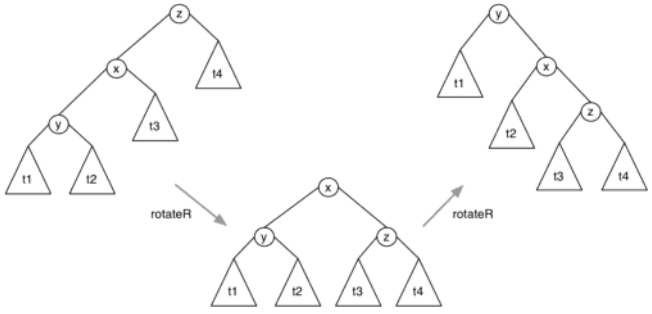
This can provide a similar effect to periodic rebalance and improves balance, but makes searching more expensive.

There are 4 cases for splay tree double-rotations:

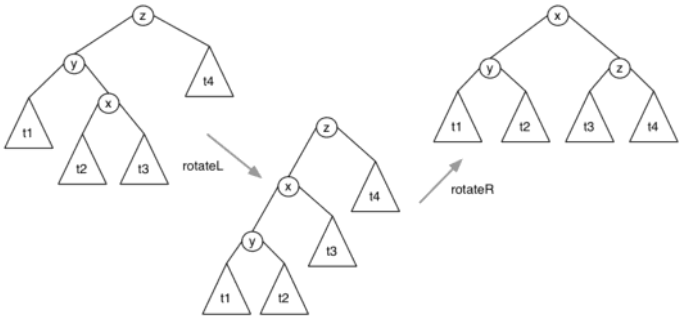
- case 1: grandchild is left-child of left-child
- case 2: grandchild is right-child of left-child
- case 3: grandchild is left-child of right-child
- case 4: grandchild is right-child of right-child



Example: double rotation case for left-child of left-child



Example: double-rotation case for right-child of left-child:

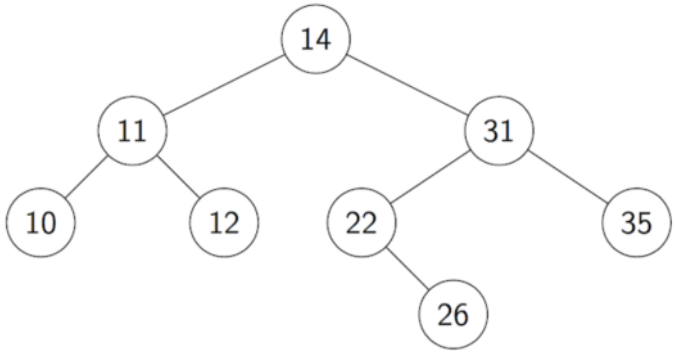


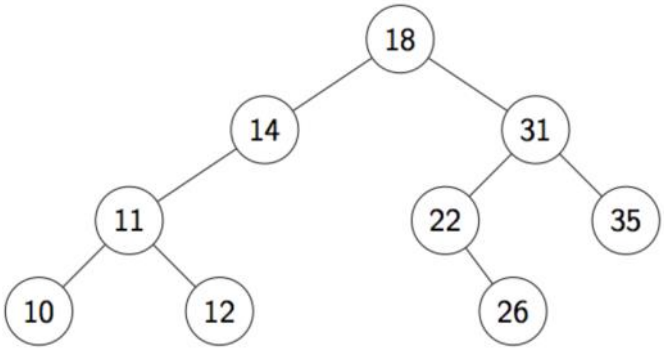
Algorithm for splay tree insertion:

```
insertSplay(tree,item):
|  Input  tree, item
|  Output tree with item splay-inserted
|
|  if tree is empty then return new node containing item
|  else if item=data(tree) then return tree
```

```
|  else if item<data(tree) then
|  |  if left(tree) is empty then
|  |     left(tree)=new node containing item
|  |  else if item<data(left(tree)) then
|  |        // case 1: left-child of left-child
|  |     left(left(tree))=insertSplay(left(left(tree)),item)
|  |     left(tree)=rotateRight(left(tree))
|  |     tree=rotateRight(tree)
|  |  else  // case 2: right-child of left-child
|  |     right(left(tree))=insertSplay(right(left(tree)),item)
|  |     left(tree)=rotateLeft(left(tree))
|  |  end if
|  |  return rotateRight(tree)
|  else if item>data(tree) then
|  |  if right(tree) is empty then
|  |     right(tree)=new node containing item
|  |  else if item<data(right(tree)) then
|  |        // case 3: left-child of right-child
|  |     left(right(tree))=insertSplay(left(right(tree)),item)
|  |     right(tree)=rotateRight(right(tree))
|  |  else  // case4: right-child of right-child
|  |     right(right(tree))=insertSplay(right(right(tree)),item)
|  |     right(tree)=rotateLeft(right(tree))
|  |     tree=rotateLeft(tree)
|  |  end if
|  |  return rotateLeft(tree)
|  end if
```

Example: Insert 18 into this splay tree:



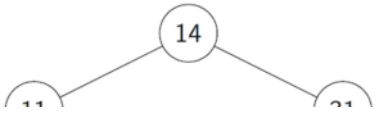


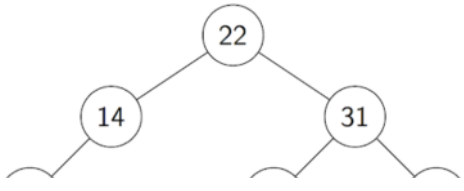
Searching in splay trees:

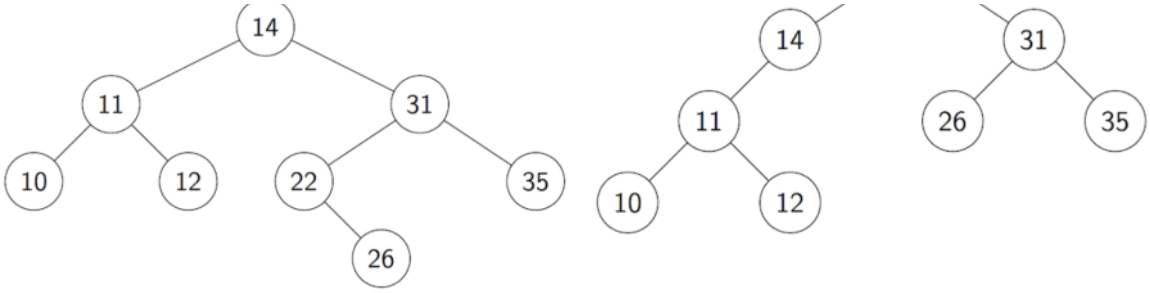
```
searchSplay(tree,item):
|  Input  tree, item
|  Output address of item if found in tree
|         NULL otherwise
|
|  if tree=NULL then
|     return NULL
|  else
|  |  tree=splay(tree,item)
|  |  if data(tree)=item then
|  |     return tree
|  |  else
|  |     return NULL
|  |  end if
|  end if
```

where `splay()` is similar to `insertSplay()`, except that it doesn't add a node. It simply moves item to root if found, or nearest node if not found

Example: search for 22 in the splay tree







Analysis of splay tree performance:

- assume that we "splay" for both insert and search
- consider: *m* insert+search operations, *n* nodes
- total number of comparisons: average $O((n+m)\cdot log_2(n+m))$

Gives good overall (amortized) cost.

- insert cost not significantly different to insert-at-root
- search cost increases, but …
  - improves balance on each search
  - moves frequently accessed nodes closer to root

But … still has worst-case search cost *O(n)*

Real Balanced Trees

Better Balanced Binary Search Trees

So far, we have seen:

- Randomised trees - these make poor performance unlikely
- Occasional rebalance - these fix the balance periodically
- Splay trees - these have a reasonable amortized performance

All still have O(n) as their worst case.

Ideally, we want both average/worst case to be O(logn).

- AVL trees - fix imbalances as soon as they occur
- 2-3-4 trees - use varying-sized nodes to assist balance
- Red-black trees - are isomorphic to 2-3-4. but use binary nodes

AVL Trees

AVL trees are invented by Georgy Adelson-Velsky and Evgenii Landis

Approach:

- insertion (at leaves) may cause imbalance
- repair balance as soon as we notice imbalance
- repairs done locally, not by overall tree restructure

A tree is unbalanced when: `abs(height(left)-height(right)) > 1`
This can be repaired by a single rotation:

- if left subtree too deep, rotate right
- if right subtree too deep, rotate left

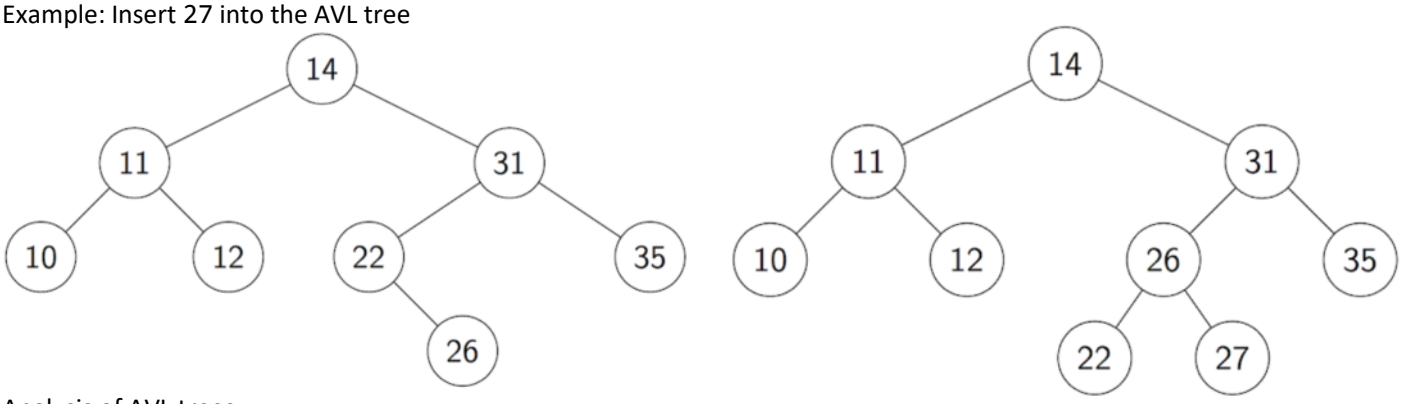
Problem: determining height/depth of subtrees may be expensive.

Implementation of AVL insertion:

```
insertAVL(tree,item):
|  Input  tree, item
|  Output tree with item AVL-inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item=data(tree) then
|     return tree
|  else
|  |  if item<data(tree) then
|  |     left(tree)=insertAVL(left(tree),item)
|  |  else if item>data(tree) then
```

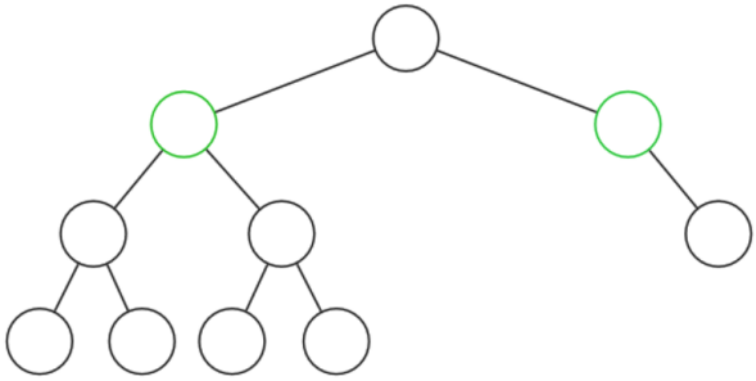
```
|   |     right(tree)=insertAVL(right(tree),item)
|   |   end if
|   |   if height(left(tree))-height(right(tree)) > 1 then
|   |     if item>data(left(tree)) then
|   |       left(tree)=rotateLeft(left(tree))
|   |     end if
|   |     tree=rotateRight(tree)
|   |   else if height(right(tree))-height(left(tree)) > 1 then
|   |     if item<data(right(tree)) then
|   |       right(tree)=rotateRight(right(tree))
|   |     end if
|   |     tree=rotateLeft(tree)
|   |   end if
|   |   return tree
|  end if
```

Example: Insert 27 into the AVL tree



Analysis of AVL trees:

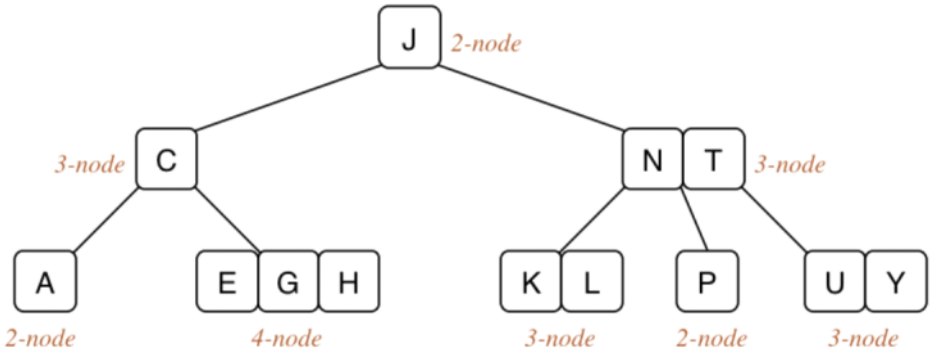
- trees are *height*-balanced; subtree depths differ by +/-1
- average/worst-case search performance of *O(log n)*
- *require* extra data to be stored in each node (efficiency)
- may not be *weight*-balanced; subtree sizes may differ



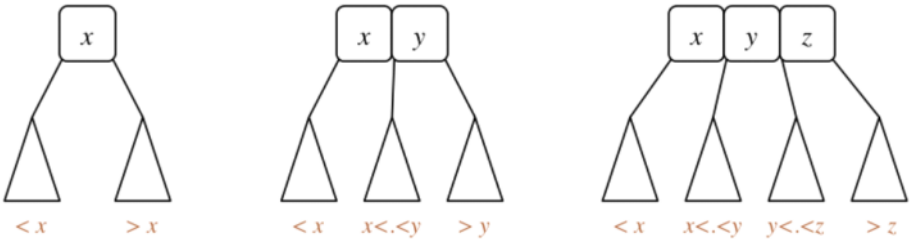
2-3-4 Trees

2-3-4 trees have three kinds of nodes

- 2-nodes, with 2 children/subtrees (same as a normal BST)
- 3-nodes, with 2 values and 3 children/subtrees
- 4-nodes, with 3 values and 4 children/subtrees



2-3-4 trees are ordered similarly to BSTs.



In a balanced 2-3-4 tree all leaves are the same distance from the root.
2-3-4 trees grow *'upwards'* from the leaves.

Possible 2-3-4 tree data structure:

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

Searching in 2-3-4 trees:

```
Search(tree,item):
|  Input  tree, item
|  Output address of item if found in 2-3-4 tree
|         NULL otherwise
|
|  if tree is empty then
|     return NULL
|  else
|  |  i=0
|  |  while i<tree.order-1 ∧ item>tree.data[i] do
|  |     i=i+1   // find relevant slot in data[]
|  |  end while
|  |  if item=tree.data[i] then   // item found
|  |     return address of tree.data[i]
|  |  else       // keep looking in relevant subtree
|  |     return Search(tree.child[i],item)
|  |  end if
|  end if
```

2-3-4 tree searching cost analysis:

- as for other trees, worst case determined by height *h*
- 2-3-4 trees are always balanced ⇒height is *O(log n)*
- worst case for height: all nodes are 2-nodes
  same case as for balanced BSTs, i.e. $h \cong log_2\ n$
- best case for height: all nodes are 4-nodes
  balanced tree with branching factor 4, i.e. $h \cong log_4\ n$

External node = leaf
No. of external nodes = no. of internal nodes + 1
B+ trees?
All nodes are at least half full

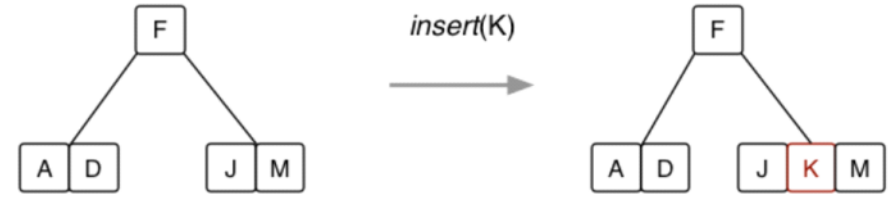
Insertion into 2-3-4 trees

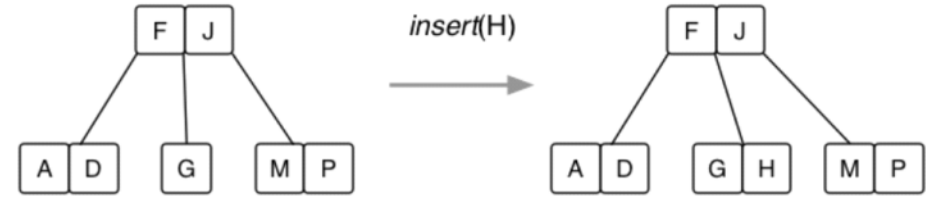
Insertion algorithm:

1. find leaf node where Item belongs (via search)
2. if not full (i.e. order < 4)
   a. insert Item in this node, order++
3. if node is full (i.e. contains 3 items)
   a. split into two 2-nodes as leaves
   b. promote middle element to parent
   c. insert item into appropriate leaf 2-node
   d. if parent is a 4-node
      i. continue split/promote upwards

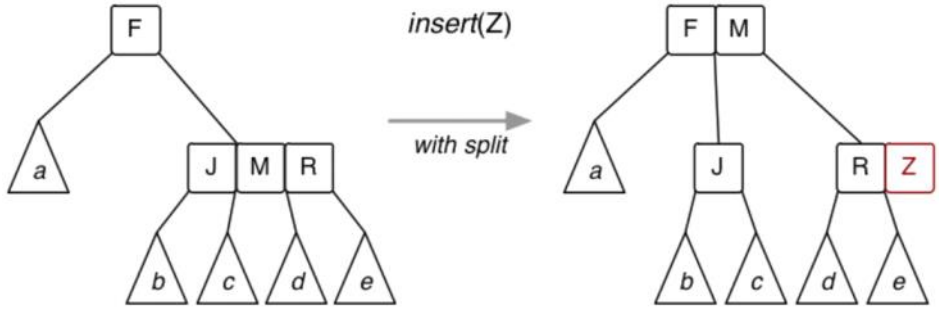
e. if promote to root, and root is a 4-node
   i. split root node and add new root

Insertion into a 2-node or 3-node:

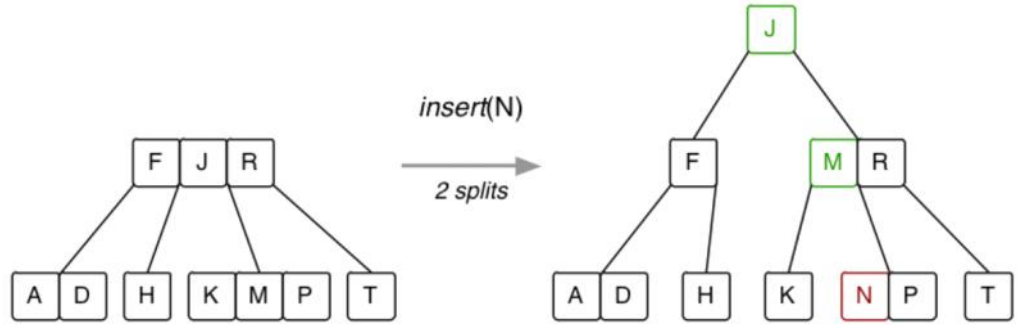




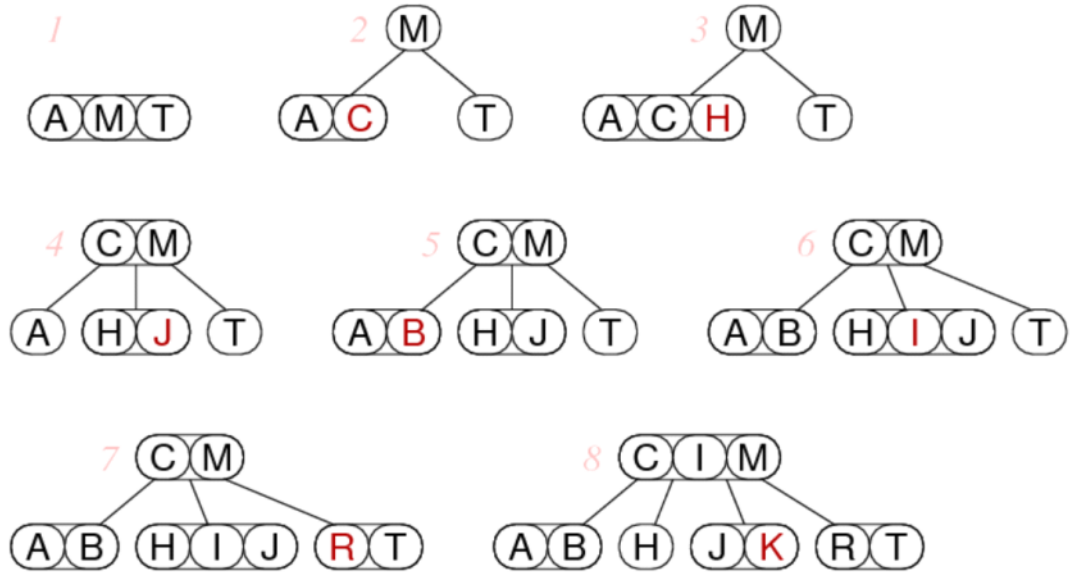
Insertion into a 4-node (requires a split)



Splitting the root:



Example: building a 2-3-4 tree with 7 insertions:



Insertion algorithm:

```
insert(tree,item):
```

```
Input  2-3-4 tree, item
Output tree with item inserted

if tree is empty then
   return new node containing item
end if
node=Search(tree,item)
parent=parent of node
if node.order<4 then
   insert item into node
   increment node.order
else
   promote = node.data[1]   // middle value
   nodeL   = new node containing data[0]
   nodeR   = new node containing data[2]
   if item<node.data[1] then
      insert(nodeL,item)
   else
      insert(nodeR,item)
   end if
   insert(parent,promote)
   while parent.order=4 do
      continue promote/split upwards
   end while
   if parent is root ∧ parent.order=4 then
      split root, making new root
   end if
end if
```

Variations on 2-3-4 trees:

Variation #1: why stop at 4? why not 2-3-4-5 trees? or *M*-way trees?

- allow nodes to hold up to *M-1* items, and at least *M/2*
- if each node is a disk-page, then we have a *B-tree* (databases)
- for B-trees, depending on Item size, *M > 100/200/400*

Variation #2: don't have "variable-sized" nodes

- use standard BST nodes, augmented with one extra piece of data
- implement similar strategy as 2-3-4 trees → red-black trees.

Red-Black Trees

Red-black trees are representations of 2-3-4 trees using BST nodes. Each node needs one extra value to encode link type, but we no longer have to deal with different kinds of nodes.

Link types:

- ***Red*** links - combine nodes to represent 3- and 4- nodes
- ***Black*** links - analogous to "ordinary" BST links (child)

Advantages:

- standard BST search procedure works unmodified
- get benefits of 2-3-4 tree self-balancing (although deeper)

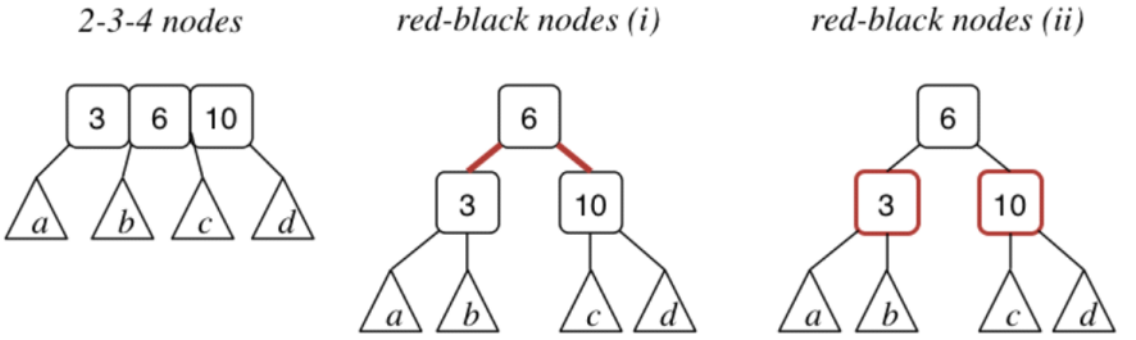
Definition of a ***red-black tree***

It is a BST in which each node is marked red or black. No two red nodes appear consecutively on any path. A red node corresponds to a 2-3-4 sibling of its parent, while a black node corresponds to a 2-3-4 child of its parent.

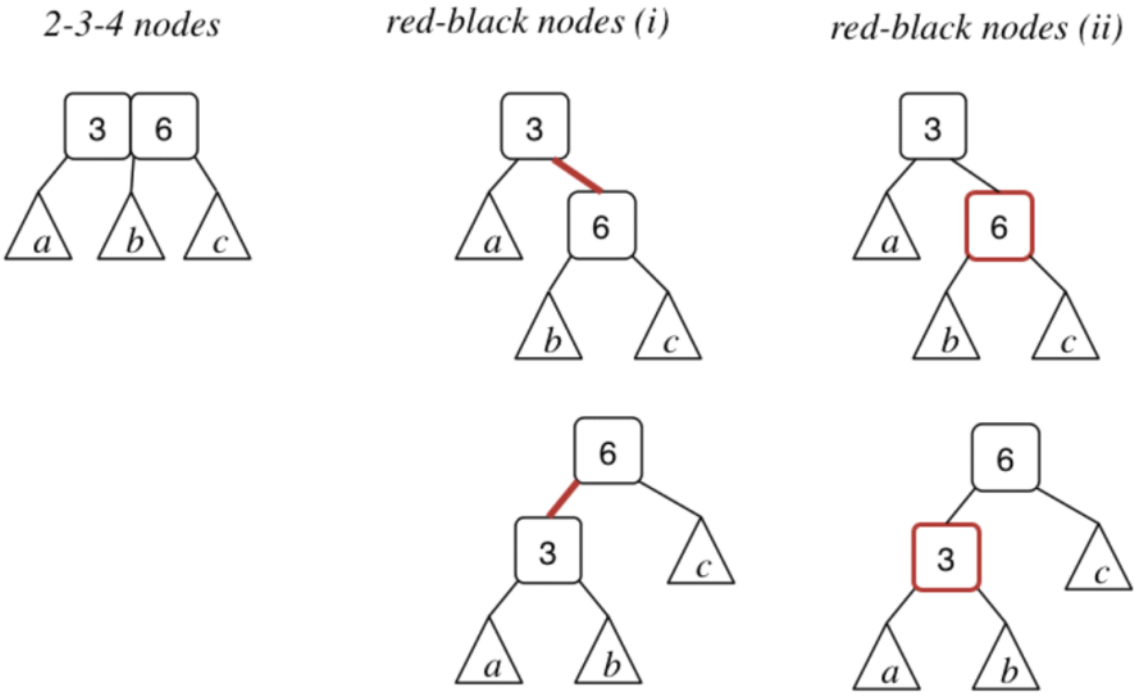
A ***balanced*** red-black tree is where all paths from root to leaf have same number of ***black*** nodes.

Insertion algorithm: avoids worst case *O(n)* behaviour
Search algorithm: standard BST search

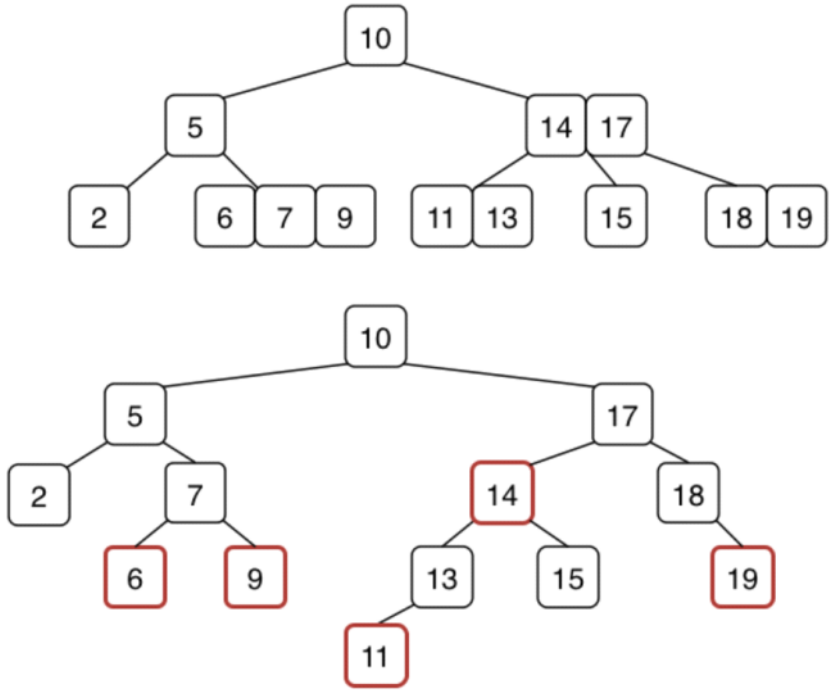
Representing 4-nodes in red-black trees:



Representing 3-nodes in red-black trees (two possibilites):



Equivalent trees (one 2-3-4, one red-black):



Note: Some texts colour the links rather than the nodes.

Red-black tree implementation:

```
typedef enum {RED,BLACK} Colour;
typedef struct node *RBTree;
typedef struct node {
   int    data;    // actual data
```

```
   Colour colour;  // relationship to parent
   RBTree left;    // left subtree
   RBTree right;   // right subtree
} node;

#define colour(tree) ((tree)->colour)
#define isRed(tree)  ((tree) != NULL && (tree)->colour == RED)
```

RED = node is part of the same 2-3-4 node as its parent (sibling)
BLACK = node is a child of the 2-3-4 node containing the parent

New nodes are always red:

```
RBTree newNode(Item it) {
   RBTree new = malloc(sizeof(Node));
   assert(new != NULL);
   data(new) = it;
   colour(new) = RED;
   left(new) = right(new) = NULL;
   return new;
}
```

Node.red allows us to distinguish links

- black = parent node is a "real"parent
- red = parent node is a 2-3-4 neighbour

Search method is standard BST search:

```
SearchRedBlack(tree,item):
|  Input  tree, item
|  Output true if item found in red-black tree
|         false otherwise
|
|  if tree is empty then
|     return false
|  else if item<data(tree) then
|     return SearchRedBlack(left(tree),item)
|  else if item>data(tree) then
|     return SearchRedBlack(right(tree),item)
|  else          // found
|    return true
|  end if
```

Red-black tree insertion

Insertion is more complex than for standard BSTs

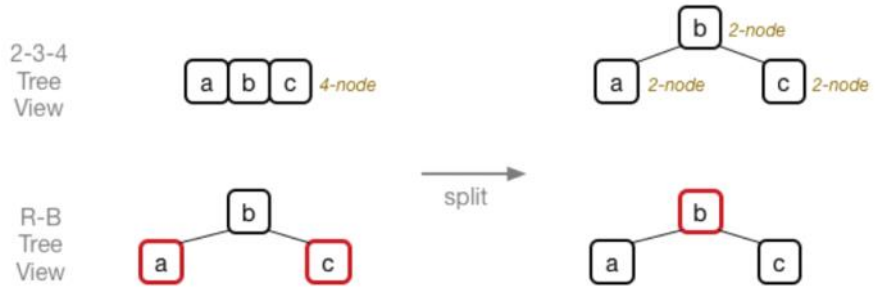
- need to recall direction of last branch (L or R)
- need to recall whether parent link is red or black
- splitting/promoting implemented by rotateLeft/rotateRight
- several cases to consider depending on colour/direction combinations

High-level description of insertion algorithm:

```
insertRB(tree,item,inRight):
|  Input  tree, item, inRight indicating direction of last branch
|  Output tree with it inserted
|
|  if tree is empty then
|     return newNode(item)
|  end if
|  if left(tree) and right(tree) both are RED then
|     split 4-node
|  end if
|  recursive insert cases (cf. regular BST)
|  re-arrange links/colours after insert
|  return modified tree
```

```
insertRedBlack(tree,item):
|  Input  red-black tree, item
|  Output tree with item inserted
|
|  tree=insertRB(tree,item,false)
|  colour(tree)=BLACK
|  return tree
```

Splitting a 4-node, in a red-black tree:



Algorithm:

```
if isRed(left(currentTree)) ∧ isRed(right(currentTree)) then
   colour(currentTree)=RED
   colour(left(currentTree))=BLACK
   colour(right(currentTree))=BLACK
end if
```

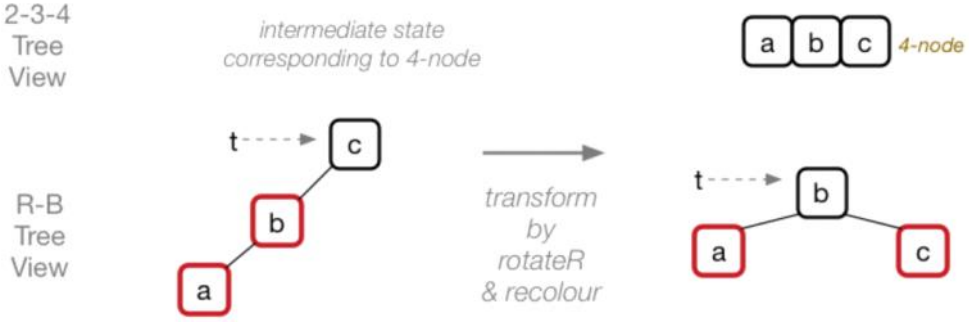
Simple recursive insert (a la BST):

Algorithm:

```
if item<data(tree)) then
   left(tree)=insertRB(left(tree),item,false)
   re-arrange links/colours after insert
else        // items larger than data in root
   right(tree)=insertRB(right(tree),item,true)
   re-arrange links/colours after insert
end if
```

Not affected by colour of `tree` node.

Re-arrange after insert (1): two successive red links = newly-created 4-node



Algorithm:

```
if isRed(left(currentTree)) ∧ isRed(left(left(currentTree))) then
   currentTree=rotateRight(currentTree)
   colour(currentTree)=BLACK
   colour(right(currentTree))=RED
```

```
end if
```

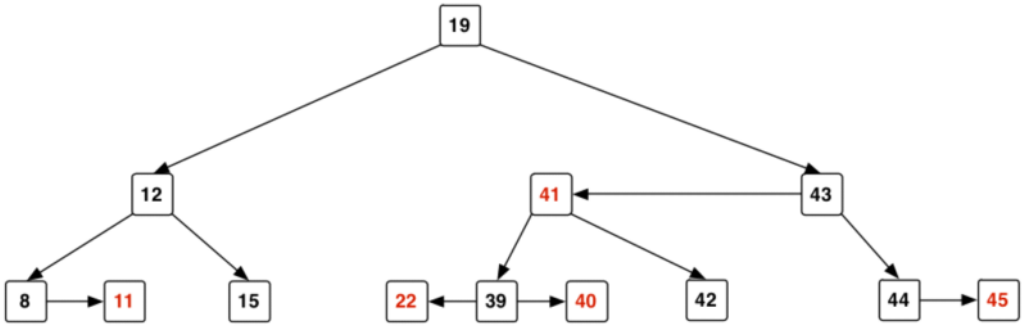
Re-arrange after insert (2): "normalise" direction of successive red links

Algorithm:

```
if inRight ∧ isRed(currentTree) ∧ isRed(left(currentTree)) then
   currentTree=rotateRight(currentTree)
end if
```

Example of insertion, starting from empty tree:

`22, 12, 8, 15, 11, 19, 43, 44, 45, 42, 41, 40, 39`



Red-black tree performance

Cost analysis for red-black trees:

- tree is well-balanced; worst case search is $O(log_2 n)$
- insertion affects nodes down one path; max #rotations is $2 \cdot h$ (where h is the height of the tree)

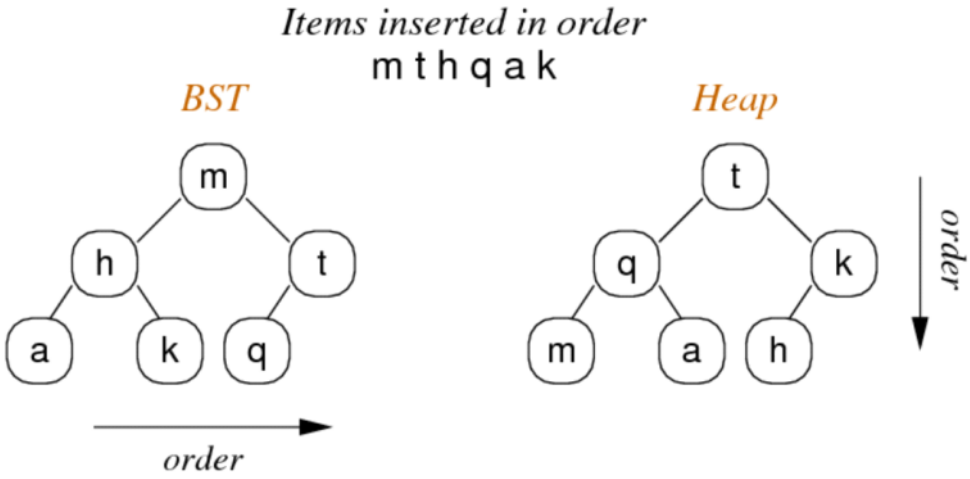
Only disadvantage is complexity of insertion/deletion code.

Note: red-black trees were popularised by Sedgewick.

Heaps

Thursday, 4 October 2018 9:16 AM

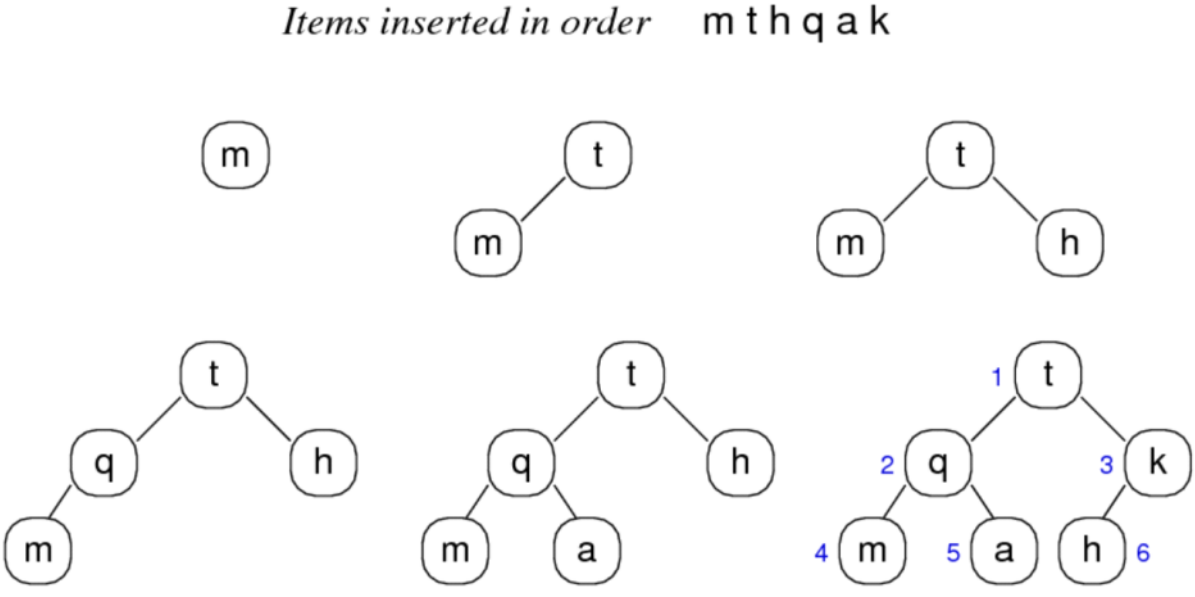
Heaps can be viewed as trees with top-to-bottom ordering, where the root is always has the highest priority. Heaps have the property that the parent node always has higher priority than its child nodes.



Heaps are typically used for implementing priority queues. The priorities are determined by order on keys. New items are added initially at the lower-most, right-most leaf. Then the item "*drifts up*" to the appropriate level in the tree. Items are always deleted by removing the root (which has the highest priority).

Since heap trees are ***dense*** trees, its depth = *floor(log_2N)+1*
Insertion cost = *O(logN)*
Deletion cost = *O(logN)*

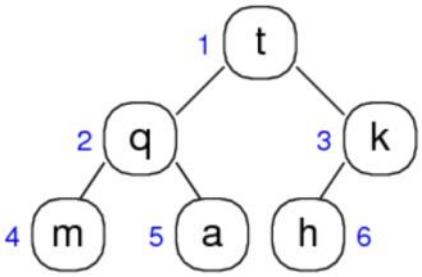
Heaps grow as follows (level-order):

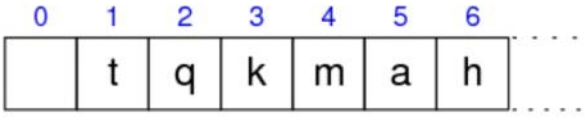


BSTs are typically implemented as linked data structures.
Heaps on the other hand are often implemented via arrays, provided that we know/assume the size.

Simple index calculations allow navigation through the *tree*

- left child of `Item` at index *i* is located at $2i$
- right child of `Item` at index *i* is located at $2i + 1$
- parent of `Item` at index *i* is located at $\frac{i}{2}$





Heap data structure:

```
typedef struct HeapRep {
   Item *items;  // array of Items
   int  nitems;  // #items in array
   int  nslots;  // #elements in array
} HeapRep;
typedef HeapRep *Heap;
```

Initialisation is similar to that for simple Hash Tables.
The only difference is that we use indexes from `1..nitems`
Note: unlike "normal" C arrays, `nitems` also gives index of last item because we ignore the slot at index `0`.

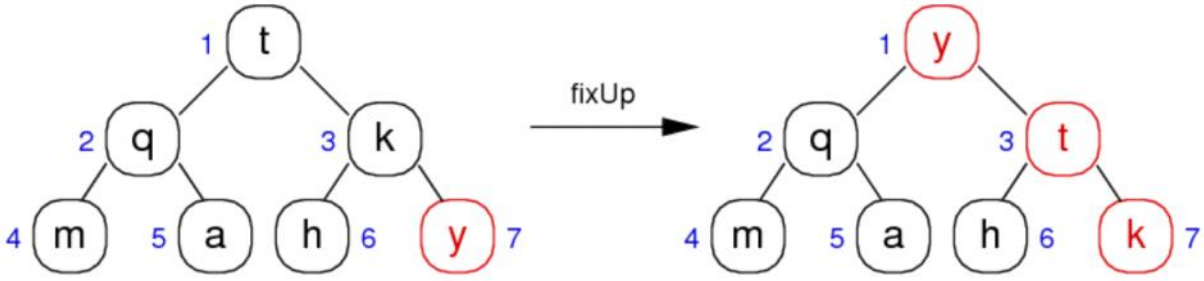
Creating new heap:

```
Heap newHeap(int N)
{
   Heap new = malloc(sizeof(HeapRep));
   Item *a = malloc((N+1)*sizeof(Item));
   assert(new != NULL && a != NULL);
   new->items = a;   // no initialisation needed
   new->nitems = 0;  // counter and index
   new->nslots = N;  // index range 1..N
}
```

Insertion with Heaps

Insertion is a two-step process:

1. Add a new element as the next available position in the bottom row. This might violate the heap property, where the new value is larger than the parent.
2. Re-organise the values along the path to root to restore the heap



Insertion into heap:

```
void insert(Heap h, Item it)
{
   assert(h->nitems < h->nslots);
   h->nitems++;
   h->items[h->nitems] = it;
   fixUp(h->items, h->nitems);
}
```

Always store the new item at next available position on bottom level
(this corresponds to next free element in the array (i.e. items[nitems])

Deletion with Heaps

Deletion is a three-step process:

1. Remove the root by replacing it with the bottom-most, right-most value
2. Remove the bottom-most, right-most value
3. Re-organise the values along the path from the root to restore the heap

Deletion from the heap always removed the root.

Deletion from heap:

```
Item delete(Heap h)
{
   Item top = h->items[1];
   // overwrite the first element with the last element
   h->items[1] = h->items[h->nitems];
   h->nitems--;
   // move the new root to the correct position
   fixDown(h->items, 1, h->nitems);
   return top;
}
```

Top-down heapify:

// force the value at a[i] into the correct position

// note that N gives max index *and* #items

```
void fixDown(Item a[], int i, int N)
{
   while (2*i <= N) {
      // compute the address of the left child
      int j = 2*i;
      // choose the larger of the two children
      if (j < N && less(a[j], a[j+1])) j++;
      if (!less(a[i], a[j])) break;
      swap(a, i, j);
      // move one level down the heap
      i = j;
   }
}
```

Hashing/Hash Tables

Monday, 8 October 2018 7:09 PM

Key-indexed arrays have "perfect" search performance *O(1)* but required a dense range of index values. The use a fixed-size array (max size ever needed).
A bigger array ⇒ more useful but wastes more space.

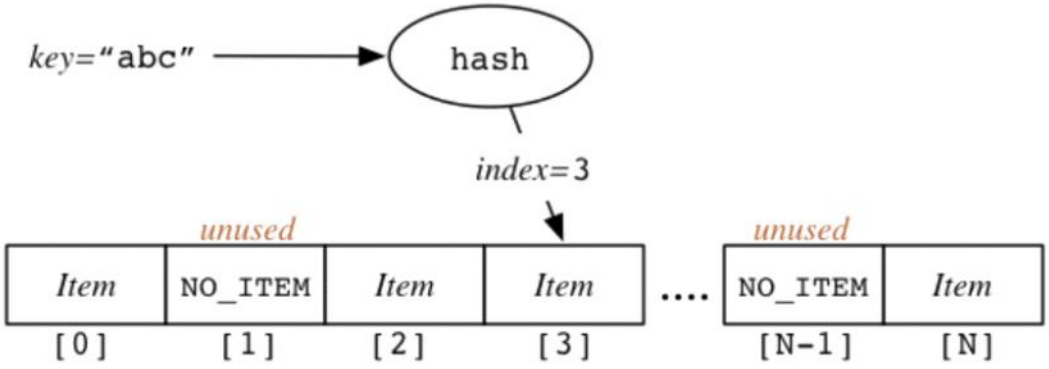
Hashing allows us to approximate this performance, but allows arbitrary types of keys.
It maps (hash) keys into compact range of index values and stores the items in a array, accessed by index value

The ideal for key-indexed collections:

```
courses["COMP3311"] = "Database Systems";
printf("%s\n", courses["COMP3311"]);
```

Almost as good:

```
courses[h("COMP3311")] = "Database Systems";
printf("%s\n", courses[h("COMP3311")]);
```



In practice:

```
item = {"COMP3311","Database Systems"};
courses = insert(courses, item);
printf("%s\n", search(courses, "COMP3311"));
```

To use arbitrary values as keys, we need three things:

- set of Key values, each key identifies one Item
- an array (of size *N*) to store Items
- a ***hash function*** *h( )* of type Key→[0..*N*-1]
  - requirement: if *(x == y)* then *h(x) == h(y)*
  - requirement: *h(x)* always returns same value for given *x*
- a ***collision*** *resolution* method
  - `collision = (x != y && h(x) == h(y))` i.e hashing is not one-to-one
  - collisions are inevitable when *dom(Key) >> N*

Generalised ADT for a *collection* of Items
Interface:

```
typedef struct CollectionRep *Collection;

Collection newCollection();    // make new empty collection
Item *search(Collection, Key); // find item with key
void insert(Collection, Item); // add item into collection
void delete(Collection, Key);  // drop item with key
```

Implementation:

```
typedef struct CollectionRep {
   ... Some data structure to hold multiple Items
```

```
} CollectionRep;
```

For hash tables, we make one change to interface:

```
typedef struct HashTabRep *HashTable;
HashTable newHashTable(int);
Item *search(HashTable, Key); // find item with key
void insert(HashTable, Item); // add item into collection
void delete(HashTable, Key);  // drop item with key
```

Example hash table implementation:

```
typedef struct HashTabRep {
   int  N;       // size of array
   Item **items; // array of (Item *)
} HashTabRep;

HashTable newHashTable(int N)
{
   HashTable new = malloc(sizeof(HashTabRep));
   new->items = malloc(N*sizeof(Item *));
   new->N = N;
   for (int i = 0; i < N; i++)
      { new->items[i] = NULL; }
   return new;
}
```

Hash Functions

Points to note:

- converts Key value to index value [0..N-1]
- deterministic (key value *k* **always** maps to same value)
- use mod function to map hash value to index value
- spread key values **uniformly** over address range
  (assumes that keys themselves are uniformly distributed)
- as much as possible, $h(k) \neq h(j)$ if $j \neq k$
- cost of computing hash function must be cheap

Basic idea behind hash function

```
int hash(Key key, int N)
{
   int val = convert key to int;
   return val % N;
}
```

If keys are ints, conversion is easy (identity function)
How to convert keys which are strings? (e.g. "COMP1927" or "9300035")

A slightly more sophisticated hash function

```
int hash(char *key, int N)
{
   int h = 0;  char *c;
   int a = 127; // a prime number
   for (c = key; *c != '\0'; c++)
      h = (a * h + *c) % N;
   return h;
}
```

Converts strings into integers in table range.
But poor choice of a (e.g. 128) can result in poor hashing.

To use all of value in hash, with suitable "randomization":

```
int hash(char *key, int N)
{
   int h = 0, a = 31415, b = 21783;
   char *c;
   for (c = key; *c != '\0'; c++) {
      a = a*b % (N-1);
      h = (a * h + *c) % N;
   }
   return h;
}
```

This approach is known as ***universal hashing***.

A real hash function (from PostgreSQL DBMS):

```
hash_any(unsigned char *k, register int keylen, int N)
{
    register uint32 a, b, c, len;

    len = keylen;
    a = b = 0x9e3779b9;
    c = 3923095;

    while (len >= 12) {
        a += (k[0] + (k[1] << 8) + (k[2] << 16) + (k[3] << 24));
        b += (k[4] + (k[5] << 8) + (k[6] << 16) + (k[7] << 24));
        c += (k[8] + (k[9] << 8) + (k[10] << 16) + (k[11] << 24));
        mix(a, b, c);
        k += 12; len -= 12;
    }

    mix(a, b, c);
    return c % N;
}
```

Where `mix` is defined as:

```
#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8);  \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12); \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5);  \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}
```

i.e. scrambles all of the bits from the bytes of the key value

Hash Table ADT

Enhanced concrete data representation:

```
#include "Item.h"
#define NoItem distinguished Item value
typedef struct HashTabRep {
   Item *items; // array of Items
   int  nslots; // # elements in array  (was called N)
```

```
   int  nitems; // # items stored in array
} HashTabRep;
typedef HashTabRep *HashTable;
```

Hash table initialisation: create a Rep and an array, fill array with `NoItem`

```
HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->items = malloc(N*sizeof(Item));
   assert(new->items != NULL);
   for (int i = 0; i < N; i++)
      new->items[i] = NoItem;
   new->nitems = 0; new->nslots = N;
   return new;
}
```

Problems with Hashing

In ideal scenarios, search cost in hash table is *O(1)*.

Problems with hashing:

- hash function relies on size of array (which can't expand)
  - changing the size of the array changes the hash function
  - could make array larger, but would need to re-insert all Items
- items are stored in (effectively) random order
- if size(KeySpace) >> size(IndexSpace), collisions inevitable
  - *collision*: k != j && hash(k,N) == hash(j,N)
- if nitems > nslots, collisions inevitable

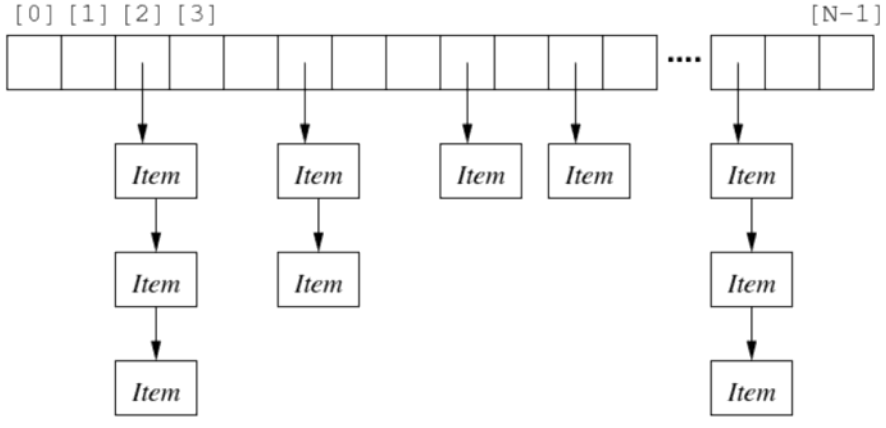
Collision Resolution

Three approaches to dealing with hash collisions:

- allow multiple Items in a single array location
  - e.g. array of linked lists (mix of *O(1)* and *O(N)*)
- systematically compute new indexes until find a free slot
  - need strategies for computing new indexes (aka *probing*)
- increase the size of the array
  - needs a method to "adjust" hash() (e.g. linear hashing)

Separate/External Chaining

Solve collisions by having multiple items per array entry.
Make each element the start of linked-list of Items.



Concrete data structure for hashing via chaining

```
typedef struct HashTabRep {
   List *lists; // array of Lists of Items
```

```
   int  nslots; // # elements in array
   int  nitems; // # items stored in HashTable
} HashTabRep;
HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->lists = malloc(N*sizeof(List));
   assert(new->lists != NULL);
   for (int i = 0; i < N; i++)
      new->lists[i] = newList();
   new->nslots = N; new->nitems = 0;
   return new;
}
```

Using the List ADT, search becomes:

```
#include "List.h"
Item *search(HashTable ht, Key k)
{
   int i = hash(k, ht->nslots);
   return ListSearch(ht->lists[i], k);
}
```

Even without List abstraction, easy to implement.
Using sorted lists gives only small performance gain.

Other list operations are also simple:

```
#include "List.h"
void insert(HashTable ht, Item it) {
   Key k = key(it);
   int i = hash(k, ht->nslots);
   ListInsert(ht->lists[i], it);
}
void delete(HashTable ht, Key k) {
   int i = hash(k, ht->nslots);
   ListDelete(ht->lists[i], k);
}
```

Essentially: select a list; operate on that list.

Cost analysis:

- N array entries (slots), M stored items
- average list length $L = M/N$
- best case: all lists are same length L
- worst case: $h(k)=0$, one list of length M
- searching within a list of length n:
  - best: 1, worst: n, average: $n/2$
- if good hash and $M \leq N$, cost is 1
- if good hash and $M>N$, cost is $(M/N)/2$

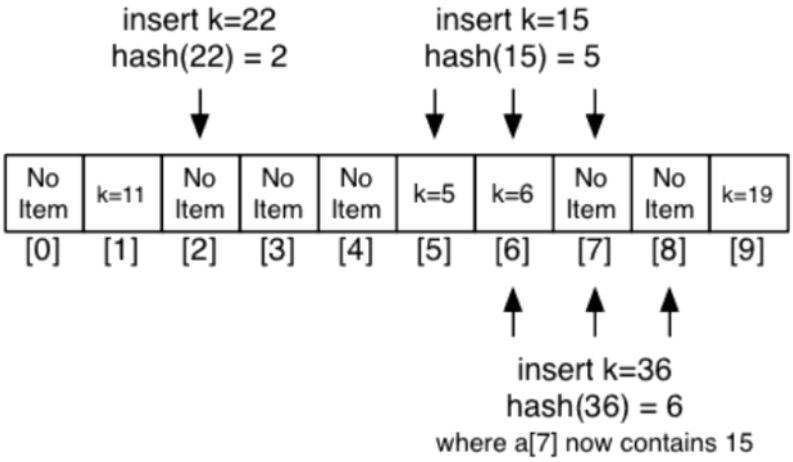
Ratio of items/slots is called **load $\alpha = M/N$**

Linear Probing

Collision resolution by finding a new location for Item

- hash indicates slot i which is already used
- try next slot, then next, until we find a free slot
- insert item in available slot

Examples:



Insert function for linear probing:

```
void insert(HashTable ht, Item it)
{
   assert(ht->nitems < ht->nslots);
   int N = ht->nslots;
   Item *a = ht->items;
   Key k = key(it);
   int i, j, h = hash(k,N);
   for (j = 0; j < N; j++) {
      i = (h+j)%N;
      if (a[i] == NoItem) break;
      if (eq(k,key(a[i]))) break;
   }
   if (a[i] == NoItem) ht->nitems++;
   a[i] = it;
}
```

Search function for linear probing:

```
Item *search(HashTable ht, Key k)
{
   int N = ht->nslots;
   Item *a = ht->items;
   int i, j, h = hash(k,N);
   for (j = 0; j < N; j++) {
      i = (h+j)%N;
      if (a[i] == NoItem) return NULL;
      if (eq(k,key(a[i]))) return &(a[i]);
   }
   return NULL;
}
```

Search cost analysis:

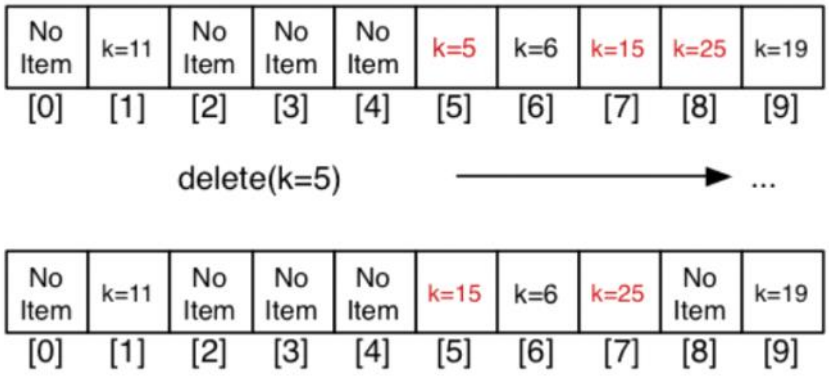
- cost to reach first Item is *O(1)*
- subsequent cost depends how much we need to scan
- affected by **load** α = M/N (i.e. how "full" is the table)
- Avg Cost for successful search = $0.5*(1 + 1/(1-\alpha))$
- Avg Cost for unsuccessful search = $0.5*(1 + 1/(1-\alpha)^2)$

Example costs:

load (α)	1/2	2/3	3/4	9/10
search hit	1.5	2.0	3.0	5.5
search miss	2.5	5.0	8.5	55.5

Assumes reasonably uniform data and good hash function.

Deletion slightly tricky for linear probing.
Need to ensure no NoItem in middle of "probe path"
(i.e. previously relocated items moved to appropriate location)



Delete function for linear probing:

```
void delete(HashTable ht, Key k)
{
   int N = ht->nslots;
   Item *a = ht->items;
   int i, j, h = hash(k,N);
   for (j = 0; j < N; j++) {
      i = (h+j)%N;
      if (a[i] == NoItem) return;
      if (eq(k,key(a[i]))) break;
   }
   a[i] = NoItem;
   ht->nitems--;

   j = i+1;
   while (a[j] != NoItem) {
      Item it = a[j];
      a[j] = NoItem; // remove 'it'
      ht->nitems--;
      insert(ht, it); // insert 'it' again
      j = (j+1)%N;
   }
}
```

A problem with linear probing: *clusters*
E.g. insert 5, 6, 15, 16, 7, 17, with hash = k%10

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	1	No Item	No Item	4	No Item	No Item	No Item	No Item	No Item
No Item	1	No Item	No Item	4	5	No Item	No Item	No Item	No Item
No Item	1	No Item	No Item	4	5	6	No Item	No Item	No Item
No Item	1	No Item	No Item	4	5	6	15	No Item	No Item
No Item	1	No Item	No Item	4	5	6	15	16	No Item
No Item	1	No Item	No Item	4	5	6	15	16	7
17	1	No Item	No Item	4	5	6	15	16	7

Double Hashing

Double hashing improves on linear probing:

- by using an increment which ...
  - is based on a secondary hash of the key
  - ensures that all elements are visited
    (can be ensured by using an increment which is relatively prime to N)
- tends to eliminate clusters ⇒ shorter probe paths

The index becomes h(x) + h2(x);

To generate relatively prime

- set table size to prime e.g. N=127
- hash2() in range [1..N1] where N1 < 127 and prime

Concrete data structures for hashing via double hashing:

```
typedef struct HashTabRep {
   Item *items;
   int  nslots;
   int  nitems;
   int  nhash2;
} HashTabRep;
#define hash2(k,N2) (((k)%N2)+1)
HashTable newHashTable(int N)
{
   HashTabRep *new = malloc(sizeof(HashTabRep));
   assert(new != NULL);
   new->items = malloc(N*sizeof(Item));
   assert(new->items != NULL);
   for (int i = 0; i < N; i++)
      new->items[i] = NoItem;
   new->nslots = N; new->nitems = 0;
   new->nhash2 = findSuitablePrime(N);
   return new;
}
```

Search function for double hashing:

```
Item *search(HashTable ht, Key k)
```

```
{
   int N = ht->nslots;
   Item *data = ht->items;
   int i, j, h = hash(k,N);
   int incr = hash2(k,ht->nhash2);
   for (j = 0, i = h; j < N; j++) {
      if (eq(k,key(data[i]) == 0)
         return &(data[i]);
      i = (i+incr)%N;
   }
   return NULL;
}
```

Insert function for double hashing:

```
void insert(HashTable ht, Item it)
{
   int N = ht->nslots;
   Item *data = ht->items;
   Key k = key(it);
   int i, j, h = hash(k,N);
   int incr = hash2(k,ht->nhash2);
   for (j = 0; j < N; j += incr) {
      i = (h+j)%N;
      if (cmp(k,key(data[i]) == 0)
         break;
      else if (data[i] == NoItem)
         break;
   }
   assert(j != N);
   if (data[i] == NoItem) ht->nitems++;
   data[i] = it;
}
```

Costs for double hashing:

load (α)	1/2	2/3	3/4	9/10
search hit	1.4	1.6	1.8	2.6
search miss	1.5	2.0	3.0	5.5

Can be significantly better than linear probing

- especially if table is heavily loaded

Summary

Collision resolution approaches:

- chaining: easy to implement, allows $\alpha > 1$
- linear probing: fast if $\alpha << 1$, complex deletion
- double hashing: faster than linear probing, esp for $\alpha \cong 1$

Only chaining allows $\alpha > 1$, but performance degrades once $\alpha > 1$

Once *M* exceeds initial choice of *N*,

- need to expand size of array (*N*)
- problem: hash function relies on *N*,
  so changing array size potentially requires rebuiling whole table
- dynamic hashing methods exist to avoid this

Text Processing Algorithms

Thursday, 11 October 2018 10:13 AM

Strings

A ***string*** is a sequence of characters.
An ***alphabet*** Σ is the set of possible characters in strings.

Examples of strings:	Examples of alphabets:
• C program	• ASCII
• HTML document	• Unicode
• DNA sequence	• {0,1}
• Digitized image	• {A,C,G,T}

Notation:

length(P)	#characters in *P*
λ	*empty* string (*length*(λ) = 0)
Σ^m	set of all strings of length *m* over alphabet Σ
Σ^*	set of all strings over alphabet Σ
substring of *P*	any string *Q* such that $P = vQ\omega$, for some $v,\omega \in \Sigma^*$
prefix of *P*	any string *Q* such that $P = Q\omega$, for some $\omega \in \Sigma^*$
suffix of *P*	any string *Q* such that $P = \omega Q$, for some $\omega \in \Sigma^*$

Notes:
$v\omega$ denotes the *concatenation* of strings v and ω
length(vω) = length(v)+length(ω) $\lambda\omega = \omega = \omega\lambda$

Example: the string **"a/a"** of length 3 over the ASCII alphabet has

- 4 prefixes: "" "a" "a/" "a/a"
- 4 suffixes: "a/a" "/a" "a" ""
- 6 substrings: "" "a" "/" "a/" "/a" "a/a"

Note: **""** means the same as λ (= empty string)

ASCII (American Standard Code for Information Interchange) specifies mappings of 128 characters to integer 0..127. the characters encoded include:

- Upper and lower case English letters: *A-Z* and *a-z*
- Digits: *0-9*
- Common punctuation symbols
- Special non-printing characters. e.g. *newline* and *space*

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	16	10	DLE	32	20		48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	24	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

Reminder:
In C, a string is an array of `char`s containing ASCII codes. These arrays have an extra element containing a `0`. the extra `0` can also be written `'\0'` (and is called the *null character* or *null-terminator*). It is convenient because we don't have to track the length of the string.

Because strings are so common, C provides convenient syntax:

```
char str[] = "hello";  // is the same as char str[] =  {'h', 'e', 'l', 'l', 'o', '\0'};
```

Note: `str[]` will have 6 elements

C provides a number of string manipulation functions via #include <string.h>, e.g.

```
strlen()   // length of string
strncpy()  // copy one string to another
strncat()  // concatenate two strings
strstr()   // find a substring inside string
```

Example:

```
char *strncat(char *dest, char *src, int n)
```

- appends string src to the end of dest overwriting the `'\0'` at the end of dest and adds terminating `'\0'`
- returns start of string dest
- will never add more than n characters
  (If `src` is less than n characters long, the remainder of `dest` is filled with `'\0'` characters. Otherwise, `dest` is not null-terminated.)

Pattern Matching

Given two strings **T** (text) and **P** (pattern), the pattern matching problem consists of finding a substring of T equal to P. This has applications in text editors, search engines and biological research.

The brute-force pattern matching algorithm checks for each possible shift of P relative to T until a match is found, or all placements of the pattern have been tried. If all placements have been tried, then there is no substring in T equal to P.

```
BruteForceMatch(T,P):
|  Input  text T of length n, pattern P of length m
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  for all i=0..n-m do
|  |  j=0                            // check from left to right
|  |  while j<m ∧ T[i+j]=P[j] do     // test ith shift of pattern
|  |     j=j+1
|  |     if j=m then
|  |        return i                 // entire pattern checked
|  |     end if
|  |  end while
|  end for
|  return -1                         // no match found
```

Brute-force pattern matching runs in O(*n.m*). Examples of worst case (forward checking):

- **T** = aaa…ah and **P** = aaah
- Looking for patterns in DNA sequences

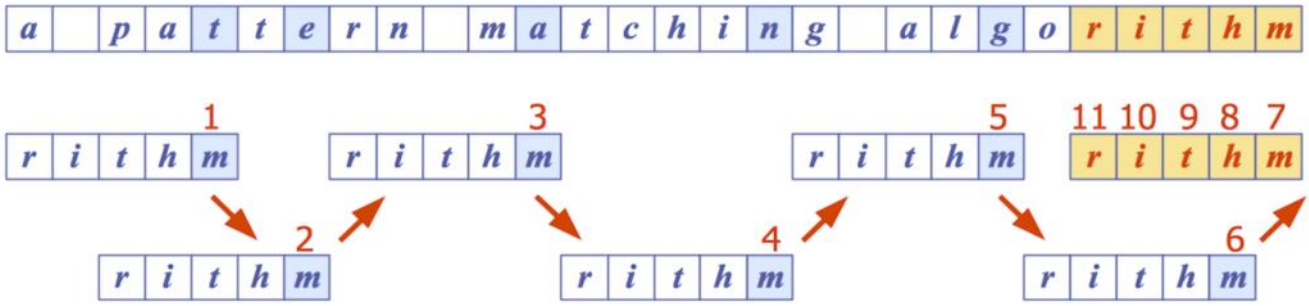
Worst case scenarios for brute-force pattern matching are less likely to occur in English text.

Boyer-Moore Algorithm

The ***Boyer-Moore*** pattern matching algorithm is based on two *heuristics*:

- ***Looking-glass heuristic:*** Compare *P* with subsequence of *T* moving ***backwards***
- ***Character-jump heuristic:*** When a mismatch occurs at *T[i]*=c
  - if *P* contains c ⇒ shift *P* so as to align the last occurrence of c in *P* with *T[i]*
  - otherwise ⇒ shift *P* so as to align *P[0]* with *T[i+1]* (a.k.a. "big jump")

Example:



Boyer-Moore algorithm pre-processes pattern *P* and alphabet Σ to build a ***last-occurrence function*** *L()*

- *L()* maps Σ to integers such that *L(c)* is defined as
  - the largest index `i` such that *P[i]=c*, or
  - `-1` if no such index exists

Example: `Σ = {a,b,c,d}, P = acab`

c	a	b	c	d
L(c)	2	3	1	-1

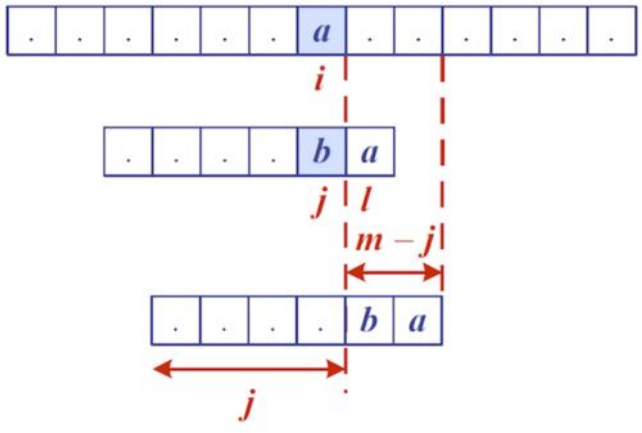
- *L()* can be represented by an array indexed by the numeric codes of the characters
- *L()* can be computed in *O(m+s)* time (*m* … length of pattern, *s* … size of Σ)

Pseudocode for Boyer-Moore algorithm:

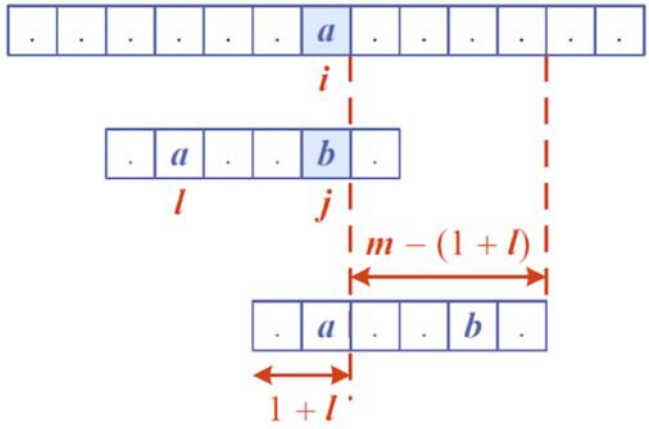
```
BoyerMooreMatch(T,P,Σ):
|  Input  text T of length n, pattern P of length m, alphabet Σ
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  L=lastOccurenceFunction(P,Σ)
|  i=m-1, j=m-1                  // start at end of pattern
|  repeat
|  |  if T[i]=P[j] then
|  |     if j=0 then
|  |        return i             // match found at i
|  |     else
|  |        i=i-1, j=j-1
|  |     end if
|  |  else                       // character jump
|  |     i=i+m-min(j,1+L[T[i]])
|  |     j=m-1
|  |  end if
|  until i≥n
|  return -1                     // no match
```

- Biggest jump (m characters ahead) occurs when `L[T[i]] = -1`

Case 1: j ≤ 1 + L[c]



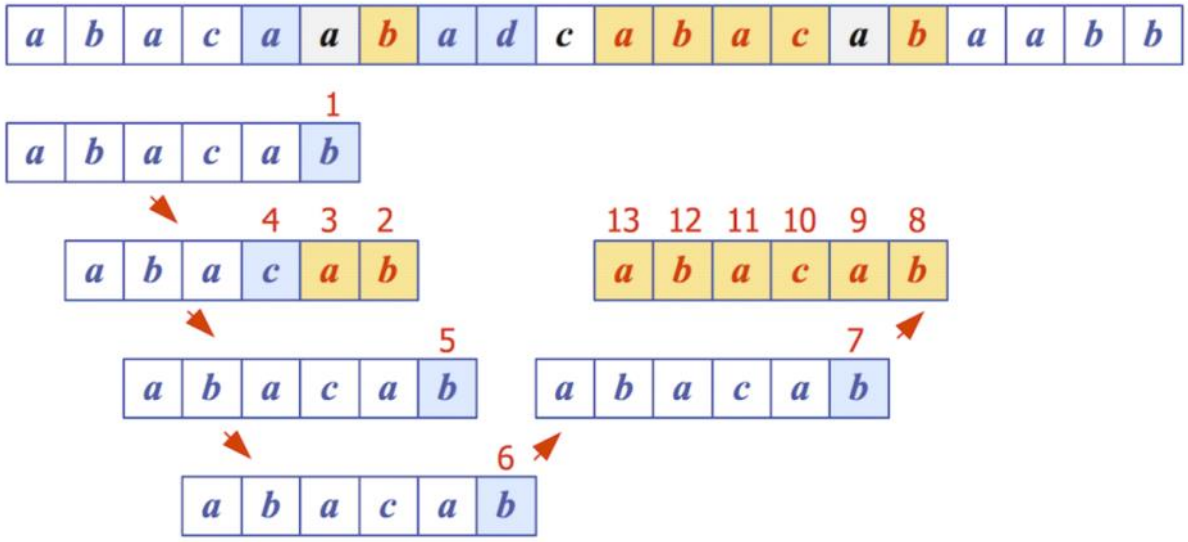
Case 2: 1 + L[c] < j



Example:

c	**a**	**b**	**c**	**d**

$L(c)$	4	5	3	-1



Analysis of Boyer-Moore algorithm:

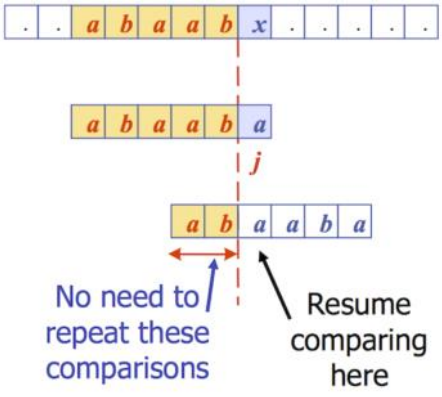
- Runs in $O(nm+s)$ time, where m = length of pattern, n = length of text, s = size of alphabet
- Example of worst case:
  - T = aaa … a
  - P = baaa
- Worst case may occur in images and DNA sequences but are unlikely to occur in English texts
  ⇒ Boyer-Moore significantly faster than brute-force on English text

Knuth-Morris-Pratt Algorithm

The ***Knuth-Morris-Pratt*** algorithm compares the pattern to the text ***left-to-right*** but shifts the pattern more intelligently than the brute-force algorithm.
Reminder:

- Q is a *prefix* of P … $P = Q\omega$, for some $\omega \in \Sigma^*$
- Q is a *suffix* of P … $P = \omega Q$, for some $\omega \in \Sigma^*$

When a mismatch we shift the search pattern P, so that the largest prefix of P[0..j] is the suffix of P[1..j].



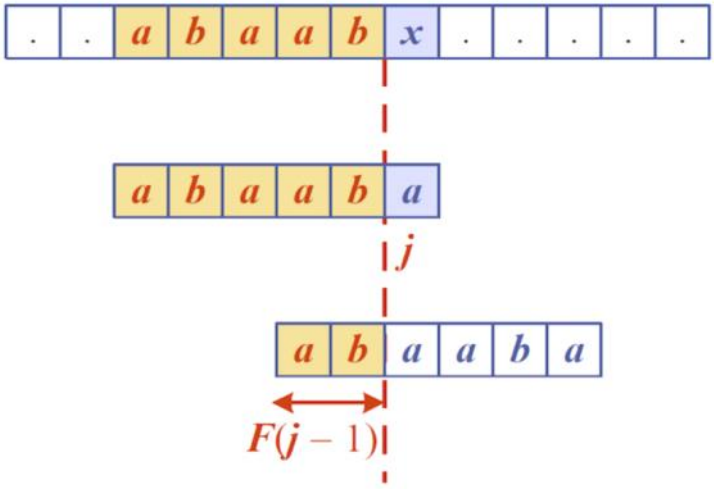
KMP pre-processes the pattern to find matches of its prefixes with itself.
The `failure function F(j)` is defined as the size of the largest prefix of `P[0..j]` that is also a suffix of `P[1..j]`.
If a mismatch occurs at P_j we advance j to `F(j-1)`.

To create the failure function table:

```
Let n = length of pattern
Let i = the iterator
for (i = 0; i < n; i++) {
    find length such that P[0..i] == P[i..n]
}
```

Example: `P = abaaba`

j	0	1	2	3	4	5
P_j	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3



Pseudocode for KMP algorithm:

```
KMPMatch(T,P):
|  Input  text T of length n, pattern P of length m
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  F=failureFunction(P)
|  i=0, j=0                    // start from left
|  while i<n do
|  |  if T[i]=P[j] then
|  |     if j=m-1 then
|  |        return i-j         // match found at i-j
|  |     else
|  |        i=i+1, j=j+1
|  |     end if
|  |  else                     // mismatch at P[j]
|  |     if j>0 then
|  |        j=F[j-1]           // resume comparing P at F[j]
|  |     else
|  |        i=i+1
|  |     end if
|  |  end if
|  end while
|  return -1                   // no match
```

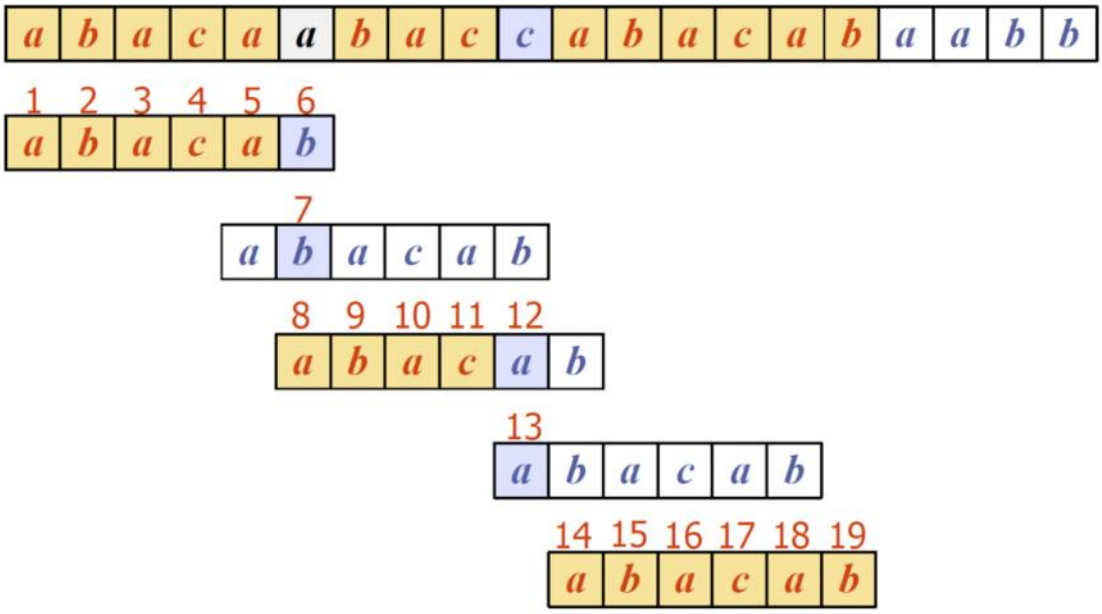
Construction of the failure function is similar to the KMP algorithm itself:

```
failureFunction(P):
|  Input  pattern P of length m
|  Output failure function for P
|
|  F[0]=0
|  i=1
|  j=0
|  while i<m do
|  |  if P[i]=P[j] then   // we have matches j+1 characters
|  |     F[i]=j+1
|  |     i=i+1, j=j+1
|  |  else if j>0 then    // use failure function to shift P
|  |     j=F[j-1]
|  |  else
```

```
|   |     F[i]=0            // no match
|   |     i=i+1
|   |  end if
|  end while
|  return F
```

Example: find P = abacab in *T* = abacaabadcabacabaabb
Failure function:

j	0	1	2	3	4	5
P_j	a	b	a	c	a	b
F(j)	0	0	1	0	1	2



Analysis of failure function computation:

- At each iteration of the while-loop, either
  - *i* increases by one, or
  - the "shift amount" *i-j* increases by at least one (observe that F(j-1)<j)
- Hence, there are no more than 2·*m* iterations of the while-loop

⇒ failure function can be computed in *O(m)* time

Analysis of Knuth-Morris-Pratt algorithm:

- Failure function can be computed in *O(m)* time
- At each iteration of the while-loop, either
  - *i* increases by one, or
  - the "shift amount" *i-j* increases by at least one (observe that F(j-1)<j)
- Hence, there are no more than 2·*n* iterations of the while-loop

⇒ KMP's algorithm runs in *optimal time O(m+n)*

Boyer-Moore vs Knuth-Morris-Pratt
Boyer-Moore algorithm

- decides how far to jump ahead based on the mismatched character in the text
- works best on large alphabets and natural language texts (e.g. English)

Knuth-Morris-Pratt algorithm

- uses information embodied in the pattern to determine where the next match could begin
- works best on small alphabets (e.g. A,C,G,T)

Pre-processing Strings

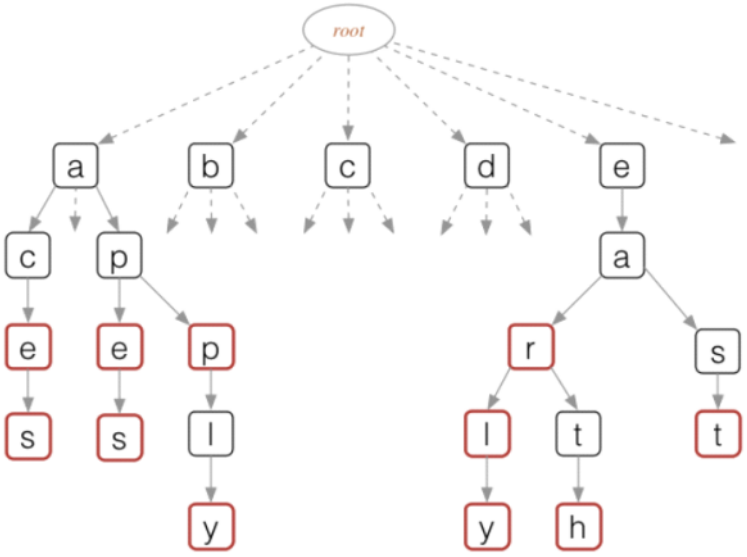
Pre-processing the pattern speeds up pattern matching queries.
After pre-processing P, the KMP algorithm performs pattern matching in time proportional to the text length.
If the text is larger, immutable and searched for often (e.g. works by Shakespeare), we can *pre-process the **text** instead of the **pattern***.

Tries

A ***trie*** is a compact data structure for representing a set of strings. e.g. all the words in a text, a dictionary. It supports pattern matching queries in time proportional to the pattern size.

Note: ***trie*** comes from *re**trie**val*, but is pronounced like "try" to distinguish it from "tree"

Tries are trees organised using parts of keys (rather than whole keys)



Each node in a trie:

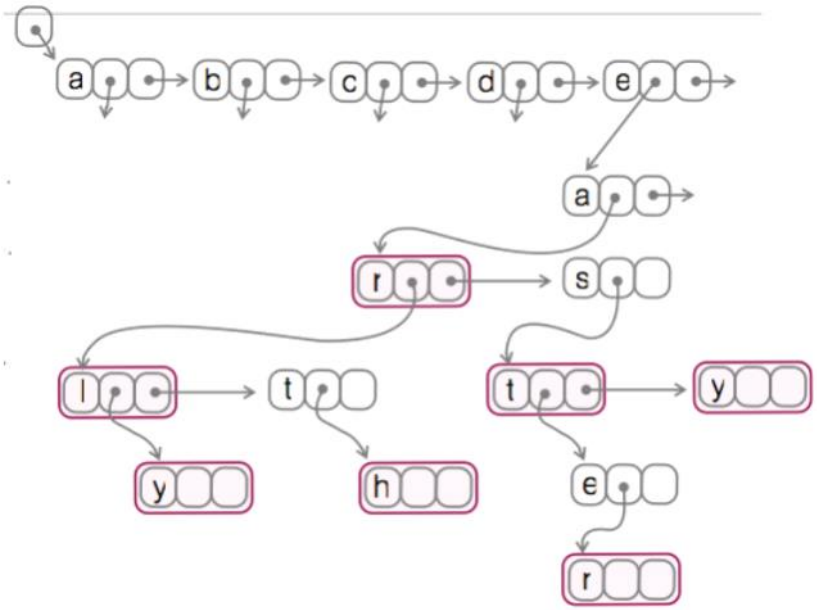
- contains one part of a key (typically one character)
- may have up to 26 children
- may be tagged as a "finishing" node
- but even "finishing" nodes may have children

Depth d of trie = length of longest key value
Cost of searching $O(d)$ (independent of *n*)

Possible trie representation:

```
#define ALPHABET_SIZE 26
typedef struct Node *Trie;
typedef struct Node {
   bool finish;      // last char in a key?
   Item data;        // no Item if !finish
   Trie child[ALPHABET_SIZE];
} Node;
typedef char *Key;
```

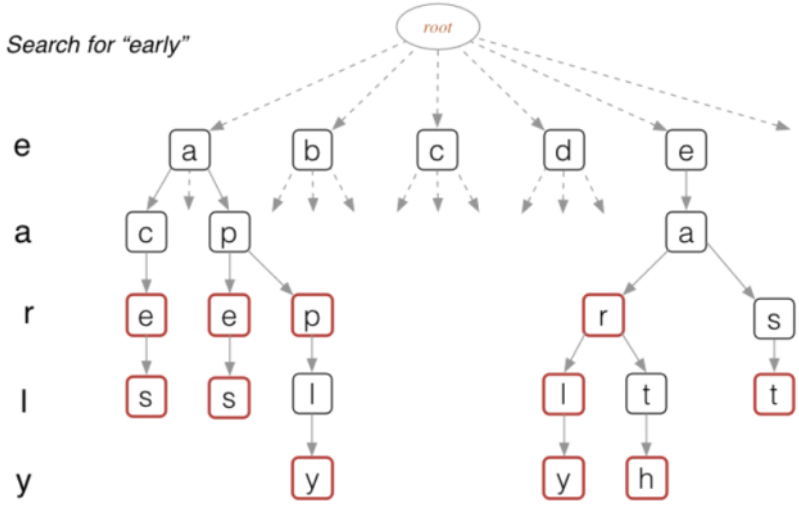
Note: You can also use BST-like nodes for more space efficient implementation of tries



Trie Operations

Basic operations for tries include:

1. Searching for a key
2. Inserting a key



Traversing a path, using char-by-char from Key:

```
find(trie,key):
|  Input  trie, key
|  Output pointer to element in trie if key found
|         NULL otherwise
|
|  node=trie
|  for each char in key do
|  |  if node.child[char] exists then
|  |     node=node.child[char]  // move down one level
|  |  else
|  |     return NULL
|  |  end if
|  end for
|  if node.finish then          // "finishing" node reached?
|     return node
|  else
|     return NULL
|  end if
```

Insertion into Trie:

```
insert(trie,item,key):
|  Input  trie, item with key of length m
```

```
|  Output trie with item inserted
|
|  if trie is empty then
|     t=new trie node
|  end if
|  if m=0 then
|     t.finish=true, t.data=item
|  else
|     t.child[key[0]]=insert(trie,item,key[1..m-1])
|  end if
|  return t
```

Analysis of standard tries:

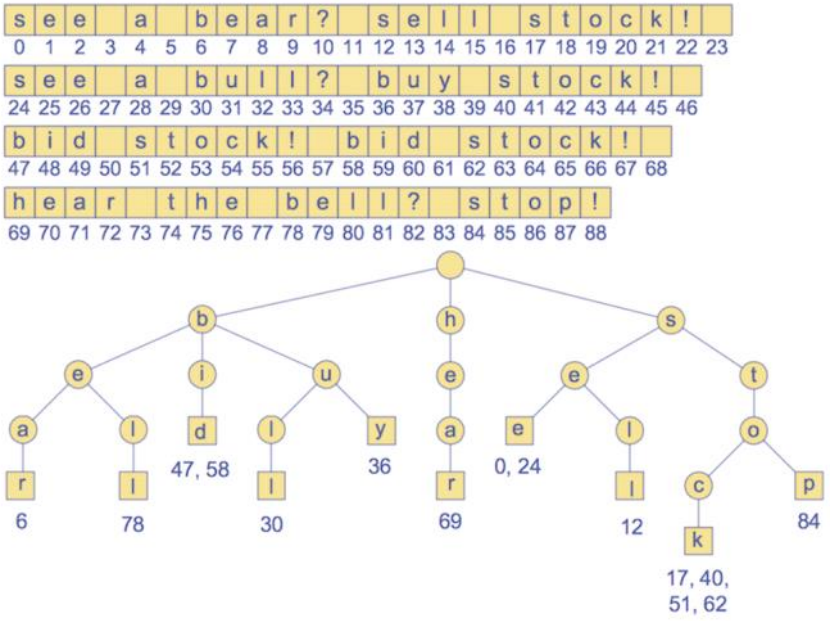
- *O(n)* space
- insertion and search in time *O(d·m)*
  - *n* = total size of text (e.g. sum of lengths of all strings in a given dictionary)
  - *m* = size of the string parameter of the operation (the "key")
  - *d* = size of the underlying alphabet (e.g. 26)

Word Matching With Tries

Pre-processing the text:

1. Insert all searchable words of a text into a trie
2. Each leaf stores the occurrence(s) of the associated word in the text

Example text and corresponding trie of searchable words:

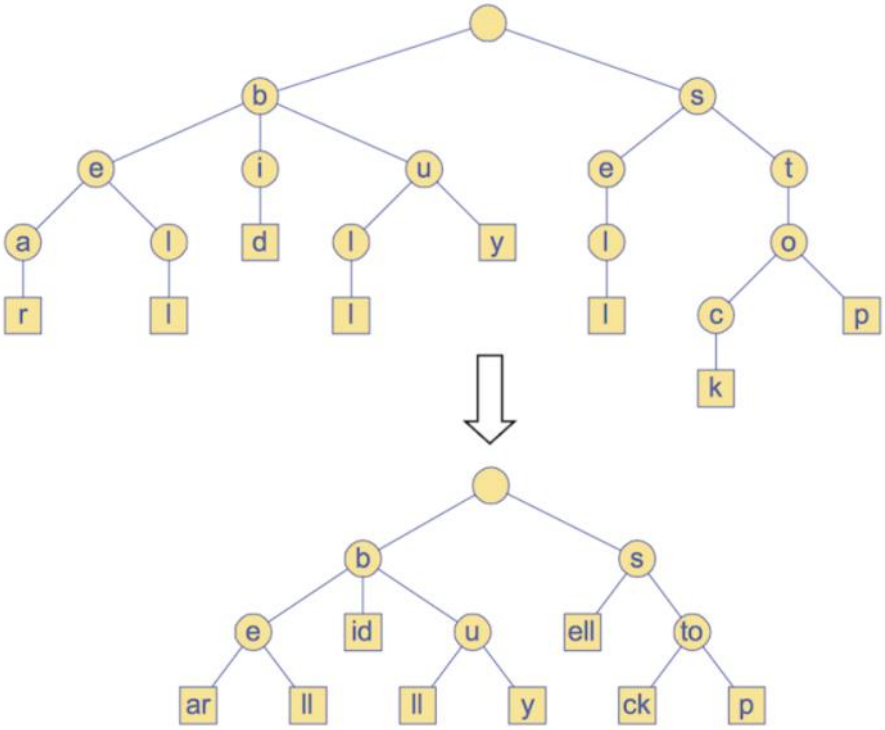


Compressed Tries

Compressed tries …

- have internal nodes of degree ≥ 2
- are obtained from standard tries by compressing "redundant" chains of nodes

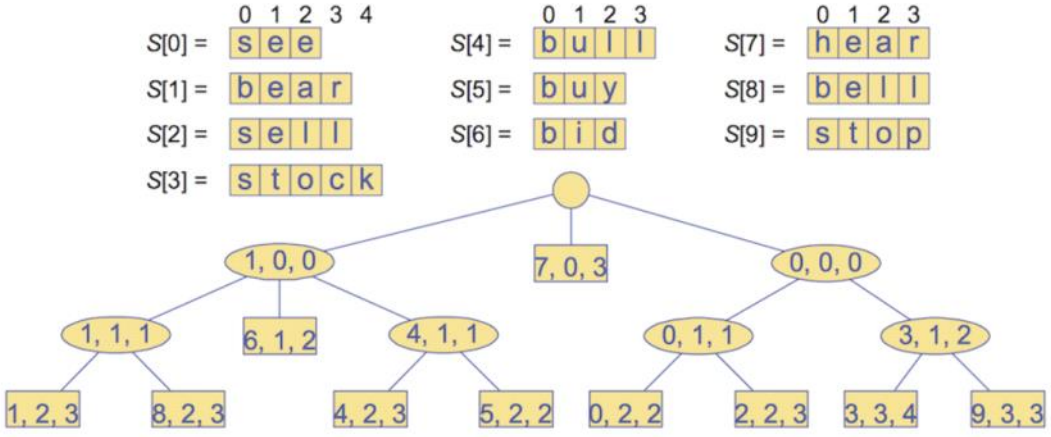
Example:



Possible compact representation of a compressed trie to encode an array *S* of strings:

- nodes store *ranges of indices* instead of substrings
  - use triple *(i,j,k)* to represente substring *S[i][j..k]*
- requires *O(s)* space (*s* = #strings in array *S*)

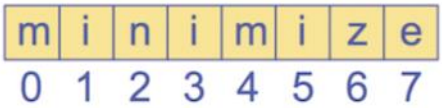
Example:

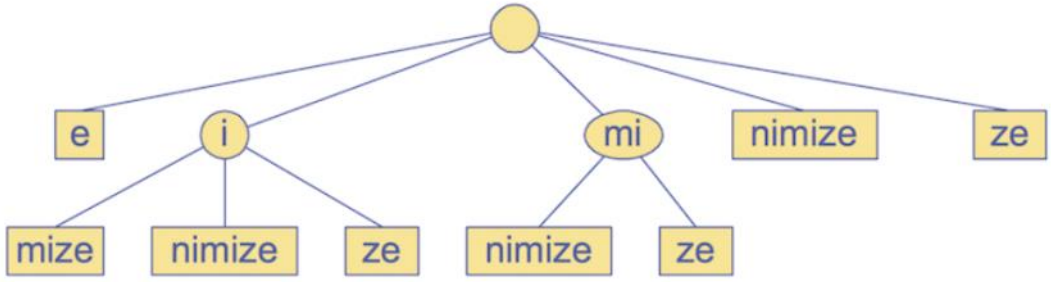


Pattern Matching With Suffix Tries

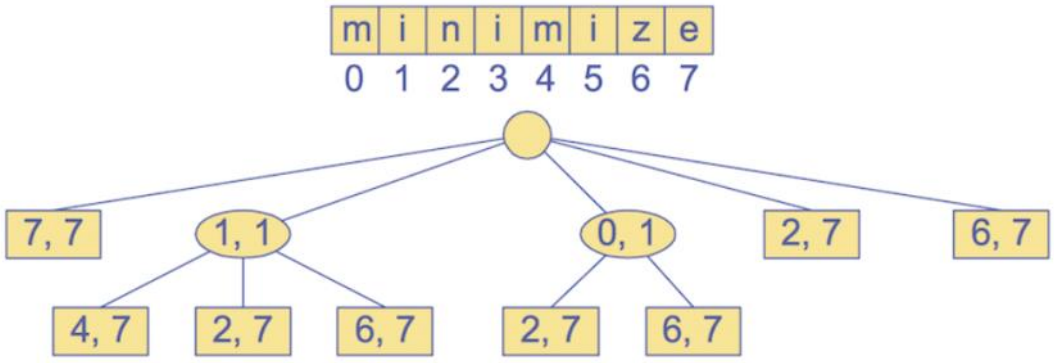
The *suffix trie* of a text *T* is the compressed trie of all the suffixes of *T*

Example:





Compact representation:



Input:

- compact suffix trie for text *T*
- pattern *P*

Goal:

- find starting index of a substring of *T* equal to *P*

Analysis of pattern matching using suffix tries:
Suffix trie for a text of size *n* …

- can be constructed in *O(n)* time
- uses *O(n)* space
- supports pattern matching queries in *O(s·m)* time
  - *m* … length of the pattern
  - *s* … size of the alphabet

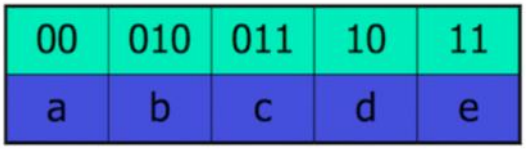
Text Compression

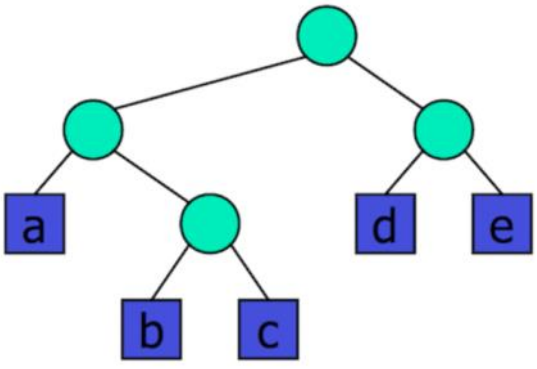
Our problem is that we need to efficiently encode a given string ***X*** by a smaller string ***Y***.
Applications: save memory and/or bandwidth

Huffman's algorithm computes the frequency `f(c)` of each character c and encodes high frequency characters with short code. It has a condition where no code word is a prefix of another code word. It uses an optimal ***encoding tree*** to determine the code words.

Code is a mapping of each character to a binary code word.
Prefix code is binary code such that no code word is the prefix of another code word.
An ***encoding tree*** represents a prefix code. Each leaf stores a character. The code word is given by the path from the root to the leaf (`0` for the left child and `1` for the right child).

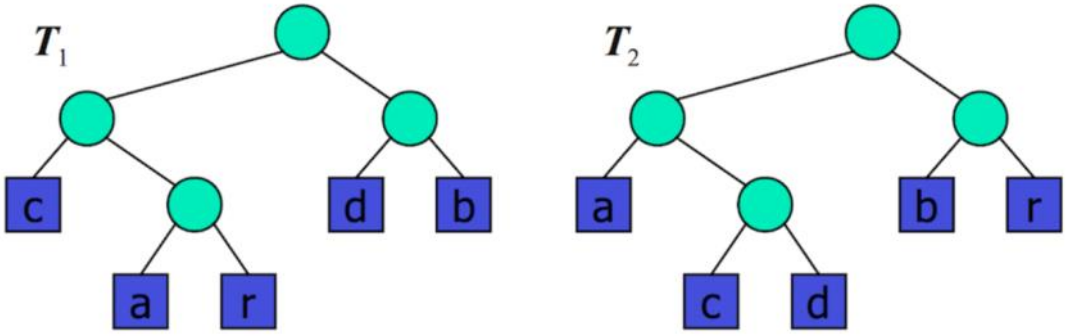
Example:





The *text compression problem*:
Given a text ***T***, find a prefix code that yields the shortest encoding of ***T***.
Solution: give short code words for frequent characters and long code words for rare character.

Example: *T* = abracadabra

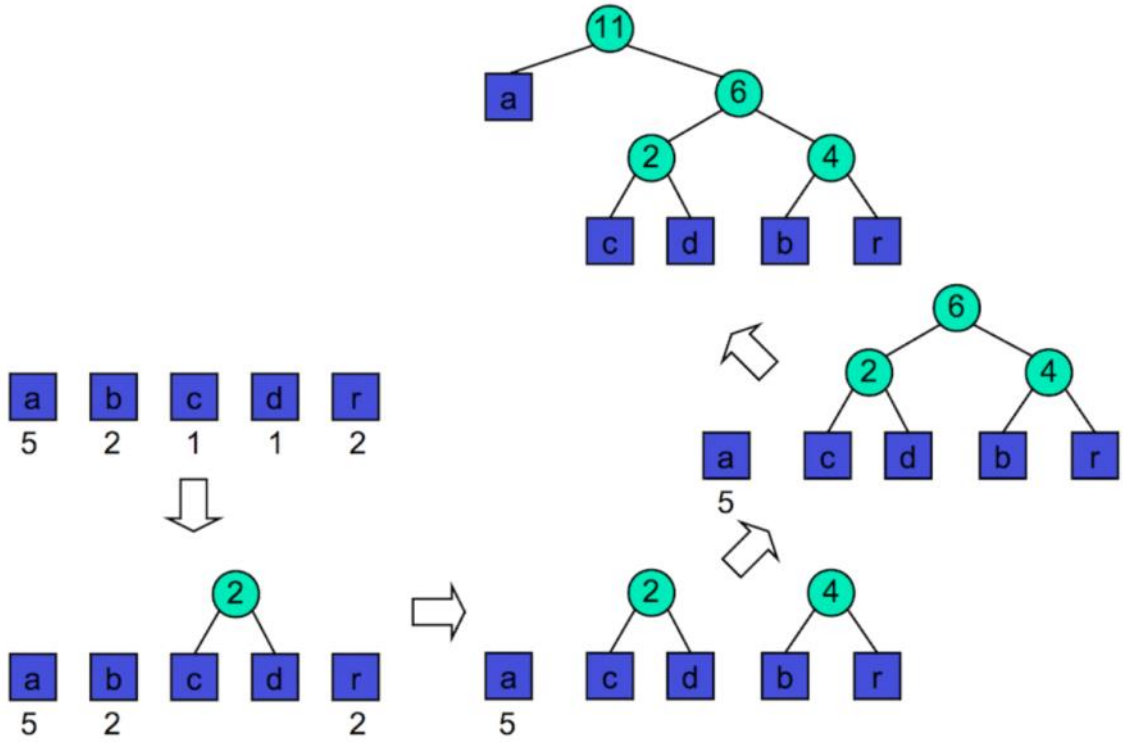


T_1 requires 29 bits to encode text T,
T_2 requires 24 bits

Huffman's algorithm

- computes frequency *f(c)* for each character
- successively combines pairs of lowest-frequency characters to build encoding tree "bottom-up"

Example: abracadabra



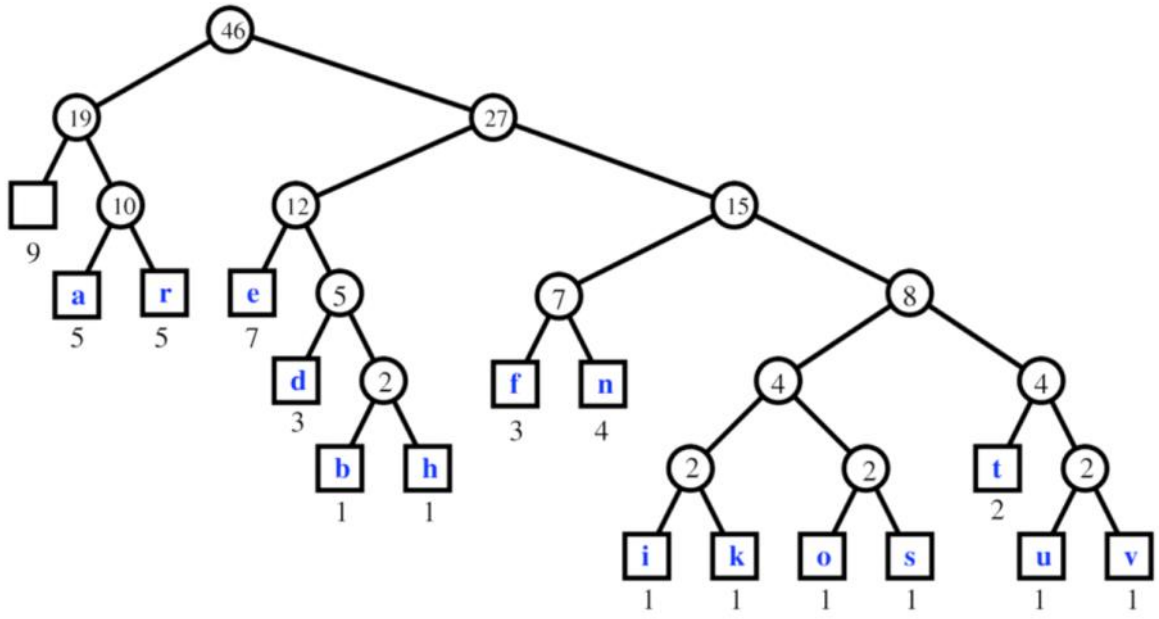
Huffman Code

Huffman's algorithm using priority queue:

```
HuffmanCode(T):
|  Input  string T of size n
|  Output optimal encoding tree for T
|
|  compute frequency array
|  Q=new priority queue
|  for all characters c do
|     T=new single-node tree storing c
|     join(Q,T) with frequency(c) as key
|  end for
|  while |Q|≥2 do
|     f₁=Q.minKey(), T₁=leave(Q)
|     f₂=Q.minKey(), T₂=leave(Q)
|     T=new tree node with subtrees T₁ and T₂
|     join(Q,T) with f₁+f₂ as key
|  end while
|  return leave(Q)
```

Larger example: a fast runner need never be afraid of the dark

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



Analysis of Huffman's algorithm:

- *O(n+d·log d)* time
  - *n* … length of the input text *T*
  - *s* … number of distinct characters in *T*

Randomised Algorithms (Non-Examinable)

Saturday, 20 October 2018 3:06 PM

Randomised Algorithms

Algorithms employ randomness to

- improve worst-case runtime
- compute correct solutions to hard problems more efficiently but with low probability of failure
- compute approximate solutions to hard problems

Randomness

- in computer games:
  - may want aliens to move in a random pattern
  - the layout of a dungeon may be randomly generated
  - may want to introduce unpredictability
- in physics/applied maths:
  - carry out simulations to determine behaviour. e.g. models of molecules are often assume to move randomly
- in testing:
  - *stress test* components by bombarding them with random data
  - random data is often seen as *unbiased data*
    - gives average performance (e.g. in sorting algorithms)
- in cryptography

Reminder: Random Numbers

In most programming languages,

```
random() // generates random numbers in a given 0 .. RAND_MAX
```

where the constant RAND_MAX may depend on the computer, e.g. `RAND_MAX = 2147483647`
To convert to a number between `0 .. RANGE`

- compute the remainder after division by RANGE+1
- Two functions are required:

```
srandom(int seed)  // sets its argument as the seed
random()  // uses a LCG technique to generate random
          // numbers in the range 0..RAND_MAX
```

  where the constant RAND_MAX is defined in `stdlib.h`
  (depends on the computer: on the CSE network, `RAND_MAX = 2147483647`)
- The period length of this random number generator is very large
  approximately $16 \cdot ((2^{31}) - 1)$

Analysis of Randomised Algorithms

Randomised algorithm to find *some* element with key *k* in an unordered list:

```
findKey(L,k):
|  Input  list L, key k
|  Output some element in L with key k
|
|  repeat
|     randomly select e∈L
|  until key(e)=k
|  return e
```

Analysis:

- *p* - ratio of elements in *L* with key *k* (e.g. p=13p=13)
- *Probability of success*: 1 (if $p > 0$)
- *Expected runtime*: $\frac{1}{p}\left(= \lim_{n\to\infty} \sum_{i=1}^{n} 1 \times (1-p)^{i-1} \times p\right)$
  - Example: a third of the elements have key *k* ⇒ expected number of iterations = 3

If we cannot guarantee that the list contains any elements with key *k* …

```
findKey(L,k,d):
|  Input  list L, key k, maximum #attempts d
|  Output some element in L with key k
|
|  repeat
|  |  if d=0 then
|  |     return failure
|  |  end if
|  |  randomly select e∈L
|  |  d=d-1
|  until key(e)=k
|  return e
```

Analysis:

- *p* … ratio of elements in *L* with key *k*
- *d* … maximum number of attempts
- *Probability of success*: 1 - (1 - *p*)^d
- *Expected runtime*: O(1) if *d* is a constant

Randomised Algorithms

Non-randomised Quicksort

Reminder: *Quicksort* applies divide and conquer to sorting:

- Divide
  - pick a *pivot* element
  - move all elements smaller than the *pivot* to its left
  - move all elements greater than the *pivot* to its right
- Conquer
  - sort the elements on the left
  - sort the elements on the right

Divide …

```
partition(array,low,high):
|  Input  array, index range low..high
|  Output selects array[low] as pivot element
|         moves all smaller elements between low+1..high to its left
|         moves all larger elements between low+1..high to its right
|         returns new position of pivot element
|
|  pivot_item=array[low], left=low+1, right=high
|  while left<right do
|  |  left  = find index of leftmost element > pivot_item
|  |  right = find index of rightmost element <= pivot_item
|  |  if left<right then
|  |     swap array[left] and array[right]
|  |  end if
|  end while
|  array[low]=array[right]
|  array[right]=pivot_item
|  return right
```

… and Conquer!

```
Quicksort(array,low,high):
|  Input  array, index range low..high
|  Output array[low..high] sorted
|
|  if high > low then
```

```
|   |   pivot = partition(array,low,high)
|   |   Quicksort(array,low,pivot-1)
|   |   Quicksort(array,pivot+1,high)
|   end if
```

```
3  6  5  2  4  1

3  1  5  2  4  6

3  1  2  5  4  6

2  1   | 3 |   6  4  5

1  2   | 3 |   6  4  5

1  2   | 3 |   5  4   | 6 |

1  2   | 3 |   4  5   | 6 |
```

Worst-case running time

Worst case for Quicksort occurs when the pivot is the unique minimum or maximum element:

- One of the intervals low..pivot-1 and pivot+1..high is of size *n-1* and the other is of size 0
  ⇒ running time is proportional to $n + n\text{-}1 + \ldots + 2 + 1$
- Hence the worst case for non-randomised Quicksort is $O(n^2)$

```
6  5  4  3  2  1

5  4  3  2  1   |   6

4  3  2  1   |   5   |   6

3  2  1   |   4   |   5   |   6

              ...

1   |   2   |   3   |   4   |   5   |   6
```

```
partition(array,low,high):
|   Input  array, index range low..high
|   Output randomly select a pivot element from array[low..high]
|          moves all smaller elements between low..high to its left
|          moves all larger elements between low..high to its right
|          returns new position of pivot element
|
|   randomly select pivot_item∈array[low..high], left=low, right=high
|   while left<right do
|   |   left  = find index of leftmost element > pivot_item
|   |   right = find index of rightmost element <= pivot_item
|   |   if left<right then
|   |      swap array[left] and array[right]
|   |   end if
|   end while
|   array[right]=pivot_item
|   return right
```

Analysis:

- Consider a recursive call to partition() on an index range of size *s*
  - *Good call*:
    both low..pivot-1 and pivot+1..high shorter than ¾·*s*
  - *Bad call*:

one of low..pivot-1 or pivot+1..high greater than ¾·*s*

- Probability that a call is good: 0.5
  (because half the possible pivot elements cause a good call)

Example of a bad call:

6 3 7 5 8 2 4 1

6 3 5 2 4 1 | 7 | 8

Example of a good call:

6 3 5 2 4 1 | 7 | 8

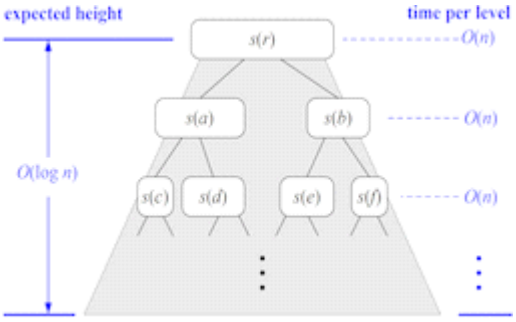
2 1 | 3 | 6 5 4 | 7 | 8

n … size of array

From probability theory we know that the expected number of coin tosses required in order to get k heads is 2·k

- For a recursive call at depth *d* we expect
  - *d*/2 ancestors are good calls
    ⇒ size of input sequence for current call is $\leq (3/4)^{d/2} \cdot n$
- Therefore,
  - the input of a recursive call at depth $2 \cdot \log_{4/3} n$ has expected size 1
    ⇒ the expected recursion depth thus is O(log *n*)
- The total amount of work done at all the nodes of the same depth is O(*n*)

Hence the expected runtime is O(*n*·log *n*)



Randomised Algorithms for NP-Complete Problems

Many NP-complete problems can be tackled by randomised algorithms that

- compute nearly optimal solutions
  - with high probability

Examples:

- travelling salesman
- constraint satisfaction problems, satisfiability
- … and many more

Simulation

Sidetrack: Approximation

Approximation is often used to solve numerical problems by

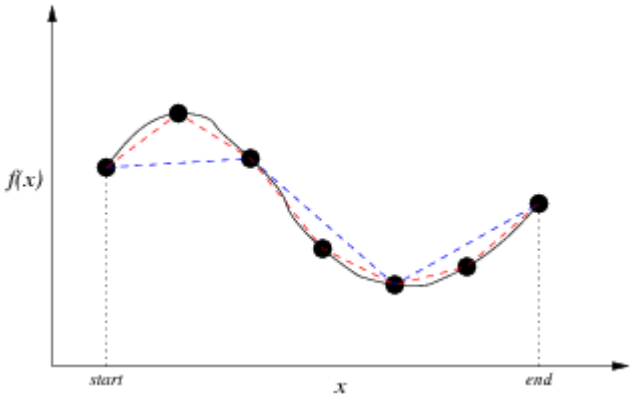
- solving a simpler, but much more easily solved, problem
- where this new problem gives an approximate solution
- and refine the method until it is "accurate enough"

Examples:

- length of a curve determined by a function *f*
- area under a curve for a function *f*
- roots of a function *f*

Example: Length of a Curve

Estimate length: approximate curve as sequence of straight lines.




```
curveLength(f,start,end):
|  Input  function f, start and end point
|  Output curve length between f(start) and f(end)
|
|  length=0, δ=(end-start)/StepSize
|  for each x∈[start+δ,start+2δ,..,end] do
|     length = length + sqrt(δ² + (f(x)-f(x-δ))²)
|  end for
|  return length
```

Trade-offs in this method:

- large step size …
  - less steps, less computation (faster), lower accuracy
- small step size …
  - more steps, more computation (slower), higher accuracy

However, too many steps may lead to higher rounding error.
Each *f* has an optimal step size …

- but this is difficult to determine in advance

Example: length = curveLength(0,π,*sin*);
Convergence when using more and more steps

```
steps =       0, length = 0.000000
steps =      10, length = 3.815283
steps =     100, length = 3.820149
steps =    1000, length = 3.820197
steps =   10000, length = 3.819753
steps =  100000, length = 3.820198
steps = 1000000, length = 3.820198
```

Actual answer is 3.820197789...

In some problem scenarios

- it is difficult to devise an analytical solution
- so build a software *model* and run *experiments*

Examples: weather forecasting, traffic flow, queueing, games
Such systems typically require random number generation

- distributions: uniform, numerical, normal, exponential

Accuracy of results depends on accuracy of model.

Example: Gambling Game

Consider the following game:

- you bet $1 and roll two dice (6-sided)
- if total is between 8 and 11, you get $2 back
- if total is 12, you get $6 back
- otherwise, you lose your money

Is this game worth playing?

Test: start with $5 and play until you have $0 or $20.
In fact, this example is reasonably easy to solve analytically.

We can get a reasonable approximation by simulation

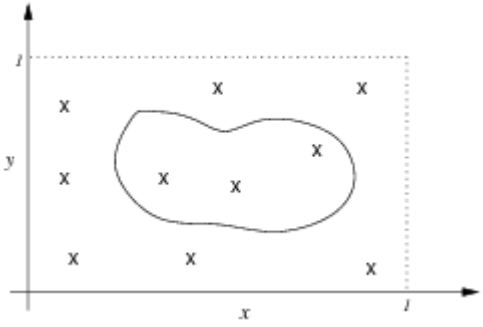
- set our initial *balance* to $5
- generate two random numbers in range 1..6 (dice)
- adjust *balance* by payout or loss
- repeat above until *balance* $\leq \$0$ *or balance* $\geq \$20$
- run a very large number of trials like the above
- collect statistics on the outcome

```
gameSimulation:
|  Output likelihood of ending with a balance ≥$20
|
|  nwins=0
|  for a large number of Trials do
|  |  balance=$5
|  |  while balance>$0 ∧ balance<$20 do
|  |  |  balance=balance-$1
|  |  |  die1=random number∈[1..6], die2=random number∈[1..6]
|  |  |  if 7≤die1+die2≤11 then
|  |  |     balance=balance+$2
|  |  |  else if die1+die2=12 then
|  |  |     balance=balance+$6
|  |  |  end if
|  |  end while
|  |  if balance≥$20 then
|  |     nwins=nwins+1
|  |  end if
|  end for
|  return nwins/Trials
```

Example: Area inside a curve

Scenario:

- have a closed curve defined by a complex function
- have a function to compute "X is inside/outside curve?"



Simulation approach to determining the area:

- determine a region completely enclosing curve
- generate very many random points in this region
- for each point *x*, compute *inside(x)*
- count number of insides and outsides
- areaWithinCurve = totalArea * insides/(insides+outsides)

I.e. we approximate the area within the curve by using the ratio of points inside the curve against those outside
Also known as *Monte Carlo estimation*

Summary

- Analysis of randomised algorithms
  - *probability of success*

    - *expected runtime*
- Randomised quicksort
- Approximation and simulation