11/10/21

LO1

CS 120 Set and Set Operators

Discrete maths: Deals with objects that are separated and distinct

We use this as a foundation and reasoning about computation

Set: A collection of objects / elements where there are no duplicates and a set is unordered.

Sets are named with upper case letters
Members / Elements lower case

$A = \{1, 2, 3, 4, 5, 6, 7\}$
$A = \{7, 6, 5, 4, 3, 2, 1\}$

These are the same as order doesn't matter

$\in$ = belongs to

$1 \in A$ 1 belongs to set A

$\notin$ = doesn't belong

$8 \notin A$ 8 doesn't belong to set A

A set is well-defined when you can decide whether any object is an element

e.g. a set of integers (positive) less than 10 is well-defined.

There can be finite or infinite sets
Writing all members is called enumerating

To enumerate large / infinite sets we can build it

e.g $A = \{1, 2, 3, 4 \ldots 3 \text{ to } \infty$

Every x such that [x is an integer and greater than zero]

$A = \{ x \mid x \text{ is an integer and } x > 0 \}$

$A = \{ x \mid P(x) \}$ P = property P

Properties can also be used for finite sets

$A = \{1, 2, 3, 4, 5, 6, 7\}$

Every x such [x greater than 0, less than 8]

$A = \{ x \mid x \text{ is an integer and } 0 < x < 8 \}$

$A = \{ x \mid P(x) \}$

N = set of natural numbers
Z = set of integers
R = set of real numbers

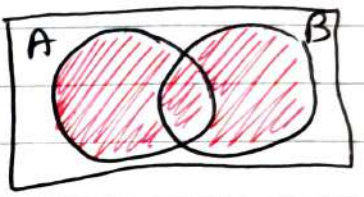
You can then specify type

$\{ x \in \mathbb{Z} \mid x \geq 3 \}$

All x that are integers greater or equal to 3

Venn diagrams can be used to visualise logical representations / relations between sets

Union (OR) ($\cup$)



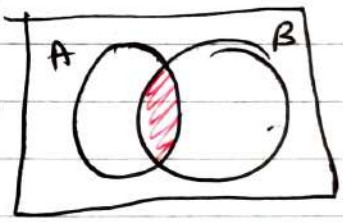
$A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$

$A = \{1, 2, 3\}$

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

$A \cup B = \{1, 2, 3, 4, 5\}$

Intersection (AND) ($\cap$)



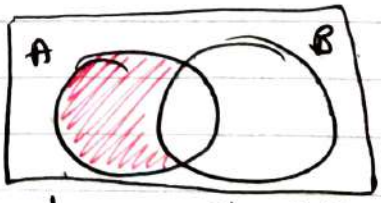
$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$

$A = \{1, 2, 3\}$

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

$A \cap B = \{3\}$

Difference ($-$) ($\setminus$)



$A - B = \{ x \mid x \in A \text{ and } x \notin B \}$

$A = \{1, 2, 3\}$

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

$A - B = \{1, 2\}$

Compound Operations

Union, Intersection, Difference are all equal in order of precedence

Therefore brackets must be used to show order of operations

$(A \cap B) - C$

$A = \{a1, a2, x, y\}$ $B = \{b1, x, y\}$,
$C = \{y\}$,
$A \cap B = \{x, y\}$
$(A \cap B) - C = \{x\}$

Cartesian Product

Ordered pair = pair of objects with an order associated to them

Convention: ordered pair x and y
$\langle x, y \rangle$

Equality: If $a = c$ and $b = d$
$\langle a, b \rangle$ and $\langle c, d \rangle$ are equal

Cartesian Product ($\times$)
$A \times B$ is the set of all ordered pairs $\langle a, b \rangle$ where $a \in A$ and $b \in B$

$A \times B = \{\langle a, b \rangle \mid a \in A \text{ and } b \in B\}$

e.g $A = \{1, 2, 3\}$ $B = \{a, b\}$

$A \times B = \{ \langle a, b \rangle \mid a \in A \text{ and } b \in B \}$

$A \times B = \{ \langle 1, a \rangle, \langle 1, b \rangle, \langle 2, a \rangle, \langle 2, b \rangle, \langle 3, a \rangle, \langle 3, b \rangle \}$

A Cartesian product is a set of all ordered pairs with the first element from set A and second element from set B

14/10/21

LO2

CSS 120 LO2

An ordered pair is an ordered set of a pair of objects

e.g $\langle 1,2 \rangle$ or $\langle x,y \rangle$

Ordered pairs are only equal $\langle a,b \rangle \langle c,d \rangle$ If $a=c$ and $b=d$

Cartesian Product (X):

is the product of two sets $A \times B$ as a set of ordered pairs

$A \times B = \{ \langle a,b \rangle \mid a \in A \text{ and } b \in B \}$

$A = \{1,2,3\}$ $B = \{a,b\}$

A \ B	a	b
1	$\langle 1,a \rangle$	$\langle 1,b \rangle$
2	$\langle 2,a \rangle$	$\langle 2,b \rangle$
3	$\langle 3,a \rangle$	$\langle 3,b \rangle$

$A \times B = \{ \langle 1,a \rangle, \langle 1,b \rangle, \langle 2,a \rangle, \langle 2,b \rangle, \langle 3,a \rangle, \langle 3,b \rangle \}$

The Set Operaters are Union ($\cup$), Intersection ($\cap$), Difference ($-$), Cartesian Product ($\times$)

Types of Sets:

Empty set - Contains no objects

$A = \{\}$

$A = \varnothing$

Disjoint sets - Two sets are disjoint if they have no elements in common

$A \cap B = \{\}$

$A \cap B = \varnothing$

Two empty sets are disjoint as their intersection has no elements

Equal Sets - Equal if they have the same elements

$A = \{a, b, c, d, e\}$ $B = \{b, a, c, d, e\}$

$\therefore A = B$

Not Equal Sets - Not equal if they do not have identical elements

$A \neq B$

Sets of Sets - Sets can also contain other sets.

$A = \{a, \{b, c\}\}$

$B = \{\{a\}\}$

$C = \{\varnothing\}$ The set containing the empty set is not equal to the empty set

Cardinality of Sets:

The cardinality is the number of elements in a set

$A = \{1, 2, 3, 4, 5\}$ $|A| = 5$

$|\emptyset| = 0$

$|\{\}| = 0$

$|\{\emptyset\}| = 1$ Which is strange to remember

$B = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

$|B| = 4$

This is because the set is counted as an element

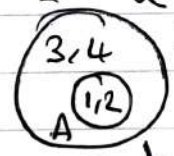
Singleton Set = Set with one element

Subset: A is a subset of set B if every element of set A is also in set B

$A \subseteq B$

A is a subset of B

$\subseteq$ = a subset of ...



Improper:

Not a subset $\not\subseteq$ if there is at least one distinct element in A

Proper subset: Is a subset of set A that is not equal to A

e.g

Written $\subset$

$A \subset B$

Superset - The opposite of a subset

$B \supseteq A$

written $\supseteq$

B is a superset of A

Proper superset is when there is also an object in B that is not in A.

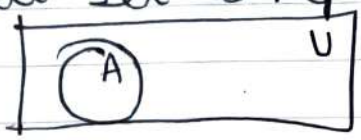
18/10/21

L03

CSS120 L03

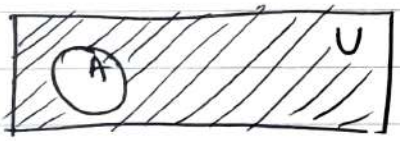
Universal set - Set of all objects under consideration (all possible elements relevant to a problem)
denoted as U

Universal set is represented by a rectangle



Complement set - The difference between the universal set and a given set

comp(A) = U - A
can also be shown as A'
Formally:
comp(A) = $\{x \in U \mid x \notin A\}$



/// = A'

Binary Relations -

You can create associations between elements and sets.
Ordered pairs can be used to show related objects.
Remember that order matters

This can be related using the Cartesian Product

Binary relation R from set A to set B
$\langle a, b \rangle \quad a \in A, b \in B$
This can be written $\langle a, b \rangle \in R$ or
$a \; R \; b$
R = in relation to

$R \subseteq A \times B$
R is a subset of $A \times B$

If $A = B$ $A \; R \; B$ is also called relation on a.

e.g List ordered pairs $A = \{0, 1, 2, 3\}$
$\langle a, b \rangle \in R$ if $b - a = 1$ $B = \{0, 1, 2, 3, 4\}$

$b - a = 1$
$b = a + 1$
$\langle a, a+1 \rangle$
$\langle 0, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle$

n-ary relations

Ordered n-tuple

- is a set of n objects with an order associated with them.

written $<x1, x2, x3 \ldots xn>$
n sets and n elements in the n-tuple

$x1 \in A1$,
$x2 \in A2$
$xn \in An$

n-tuples can be equal if each elements is equal and equivalent in order

Cartesian Product of n-sets

- The set of all possible ordered n-tuples written $A1 \times A2 \times \ldots \times An$

e.g $S = \{John, Jim, Helen, Mary\}$
$C = \{course1, course2, course3\}$
$M = \{65, 41, 55, 72, 63\}$

<John, course1, 65>
<Jim, course2, 41>
<Mary, course3, 55>
<Helen, course2, 72>
<Helen, course3, 63>

$T \subseteq S \times C \times M$

e.g a set of 4, a set of 3, a set of 5
= 60 ordered pairs

Binary relation, Ternary relation, n-ary relation

For an n-ary relation

$R \subseteq A1 \times A2 \times A3 \times \ldots An$
(subset of cartesian product $A1 \times A2 \ldots An$)

e.g Let $A = \{0, 1, 2, 3, 4\}$
R on $A \times A \times A$
$\langle a, b, c \rangle$ such $a < b < c$
List the ordered 3-tuples in relation R

= $\langle 0,1,2 \rangle$, $\langle 0,1,3 \rangle$, $\langle 0,1,4 \rangle$,
$\langle 0,2,3 \rangle$, $\langle 0,2,4 \rangle$, $\langle 0,3,4 \rangle$,
$\langle 1,2,3 \rangle$, $\langle 1,2,4 \rangle$, $\langle 1,3,4 \rangle$
, $\langle 2,3,4 \rangle$

You may see many questions like this

Equality of Relations

$R1 \subseteq A1 \times A2$
$R2 \subseteq B1 \times B2$

$R1 = R2$ if $A1 = B1$, $A2 = B2$
and also $R1 = R2$ as a set

n-ary relation $R1 \subseteq A1 \times \dots An$
m-ary relation $R2 \subseteq B1 \times \dots Bm$

$R1 = R2$ if $m = n$, $Ai = Bi$ for each i,
$R1 = R2$ as a set of ordered tuples

28/11/2022

SCC 120 LO4

Linear Search -

```
boolean isInArray(int theArray[], int iSearch){
  int N = theArray.length;
  for(int i=0; i<N; i++)
     if(theArray[i] == iSearch)
        return true;
  return false;
}
```

If element iSearch is in the Array $T(N) = 3/2N + 7/2$

If not $T(N) = 3N + 4$.

31/1/2022

SCC 120 L25

The N^{th} partial sum = $\frac{N(N+1)}{2}$

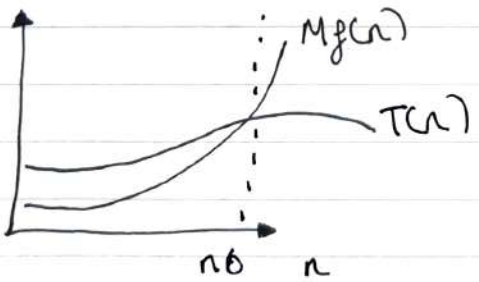
e.g $1+2+3+\dots(N-1) = \frac{(N-1)(N)}{2}$

The growth of functions -
Let A $T(n) = 5000n$ and B $T(n) = 1.1^n$ for inputs of n elements.
This means B cannot be used for large inputs whilst A is still feasible. So the importance is the growth of the time complexity functions.

In order :
$1 \rightarrow \log n \rightarrow n \rightarrow n\log n \rightarrow n^2 \rightarrow n^2 \log n \rightarrow n^3 \rightarrow n^c \rightarrow c^n \rightarrow n!$

The growth of functions is usually described with big-O notation.

Big O Notation -
Let $T(n)$ and $f(n)$ be two positive functions.
$T(n) \in O(f(n))$ and $T(n)$ has order of $f(n)$.
If there are two positive constants M and n_0 such $T(n) \leq M \times f(n)$ for all $n \geq n_0$.



The idea is to establish an upper boundary for the growth of the function $T(n)$ for large n.

e.g $T(n) = 3n+4$ $f(n) = n$

Show $T(n) = O(f(n))$

For $n \geq 1$ $T(n) = 3n+4 \leq 3n+4n$

so $T(n) = 3n+4 \leq 7n$

Therefore for $M = 7$ and $n_0 = 1$ $T(n) \leq 7n$ for $n \geq 1$

$T(n) \in O(n)$

4/12/2022

SCC 120 L26 Big O Notation + L27

The idea behind Big O Notation is to establish an upper boundary for the growth of the function T(n) for large n.

In practice we always use the smallest simple function f(n) for which T(n) is O(f(n))

Case 1: Constant
e.g growth rate = O(1) for best and worst case.

Case 2: Linear (average of n integers)
e.g best case and worst case growth rate = O(N)

Case 3: Linear (minimum of n integers)
e.g best case and worst case growth rate = O(N)

Case 4: Logarithmic
e.g best and worst case growth rate = O(logN)

Case 5: Quadratic, Cubic
e.g $O(N^2)$ and $O(N^3)$ respectively.

Linear search best case is O(1) and worst case is O(N). Best case means first element is found.
Further examples found in Week 13 L2.

SCC 120 L28 (Big Ω and Big Θ Notations) + L29

CP1 Chp 2

Key Maths (Renew)

$$\sum_{i=a}^{b} 1 = b - a + 1 \qquad \sum_{i=a}^{b} M = M \sum_{i=a}^{b} 1$$

$$\sum_{i=1}^{b} i = \tfrac{1}{2} b(b+1) \qquad \sum_{i=a}^{b} Ai = A \sum_{i=a}^{b} i$$

What is the worst case time complexity (Big O Notation) of the following code?

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

3rd line could be written as

```
for (j = i+1; j <= N; j++) {
```

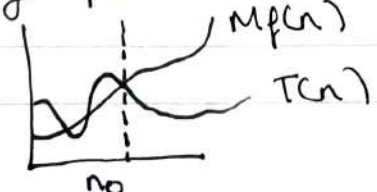
This means

$$T(N) = \sum_{i=0}^{N-1} \sum_{j=i+1}^{N} 1 = \sum_{i=0}^{N-1} N - \sum_{i=0}^{N-1} i = N^2 - \tfrac{1}{2}N(N-1)$$

$$= \tfrac{1}{2}N^2 + \tfrac{1}{2}N$$

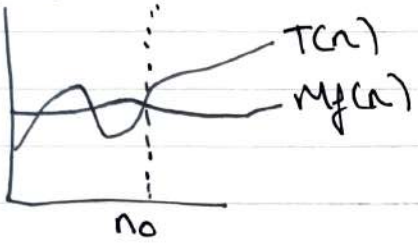
The Big O Notation -

* Let $T(n)$ and $f(n)$ be two positive functions from the integers or real numbers to real numbers.
* $T(n)$ is $O(f(n))$ if even as n becomes large, $T(n)$'s growth is bounded from above by $f(n)$ meaning it grows no faster than $f(n)$.
* $T(n) \in O(f(n))$ if $T(n) \leq M \times f(n)$ for all $n \geq n_0$ for positive constants M and n_0.



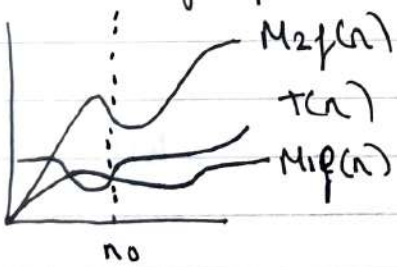
The Big Ω Notation -

* Let T(n) and f(n) be two positive functions from integers or real numbers to real numbers
* T(n) is Ω(f(n)) if even as n becomes large, T(n)'s growth is bounded from below by f(n) meaning it grows no slower than f(n).
* T(n) ∈ Ω(f(n)) if $T(n) \geq M \times f(n)$ for all $n \geq n_0$ for positive constants M and n_0.



The Big Θ Notation -

* Let T(n) and f(n) be two positive functions from integers or real numbers to real numbers.
* T(n) is Θ(f(n)) if as n becomes large, T(n)'s growth is bounded from above and below by f(n) meaning it grows no faster or slower than f(n)
* T(n) ∈ Θ(f(n)) if $M_1 f(n) \leq T(n) \leq M_2 f(n)$ for all $n \geq n_0$ for positive constants.



They are used as fast approaches to count the number of times a for loop is executed.

Example -

$T(n) = 3n+4$ $f(n) = n$ show $T(n)$ is $\Theta(f(n))$

We need to prove $T(n) \in O(n)$ and $T(n) \in \Omega(n)$

Proved in L26 $T(n) \in O(n)$

For $n \geq 1$ $T(n) = 3n+4 \leq 3n+4n$

$T(n) = 3n+4 \leq 7n$

for $M = 7$ and $n_0 = 1$ $T(n) \leq 7n$ for all $n \geq 1$ so $T(n) \in O(n)$

For $n \geq 0$ $T(n) = 3n+4 \geq 2n+0$

$T(n) = 3n+4 \geq 2n$

for $M = 2$ and $n_0 = 0$ $T(n) \geq 2n$ for all $n \geq 0$

$T(n) \in \Omega(n)$. □

Linear Search (Big O, Big Ω and Big Θ)

Case	$T(n)$	Θ
Worst	Linear of n	$\Theta(n)$, $O(n)$, $O(n^2)$, $\Omega(n)$, $\Omega(\log n)$
Average	Linear of n	$\Theta(n)$ but $O(n)$, $O(n^2)$, $\Omega(n)$, $\Omega(\log n)$
Best	Constant	$\Theta(1)$ but $O(n)$, $O(n^2)$, $O(\log n)$

General Rule:

1) Time complexity of a loop is considered as $\Theta(\log n)$ if the loop variables is divided / multiplied by a constant amount

2) Time complexity of a loop is considered as $\Theta(n)$ if the loop variables is incremented / decremented by a constant amount.

1) e.g `for(int i = 1; i <= n; i *= c){` `*` or `/`

2) e.g `for(int i = 1; i <= n; i += c){` `+` or `-`

SCC 120 L30 (Sentinel and Binary Search)

When a linear search is performed on an array of size N then in the worst case a total of (N+1) comparisons are made for the index of the element to be compared so that the index is not out of bounds of the array.

Sentinel Search -

Sentinel Search is a type of Linear search where the number of comparisons is reduced as compared to the linear search.

1) check if iSearch is not at the end of the array. Then replace the number at the end of the array with iSearch.
2) Perform Linear Search

```
boolean isInSentinel (int[] theArray, int iSearch) {
  int N = theArray.length;
  if (theArray[N-1] == iSearch)
     return true;
  theArray[N-1] = iSearch;
  for (int i=0; [ ]; i++) {
     if (theArray[i] == iSearch)
        break;
  }
  return i < (N-1);
}
```

Worst Case Linear = 3N + 4

Worst Case Sentinel = 2N + 5

Binary Search -

This works on sorted arrays by eliminating half of the array from consideration.

mid = $\frac{hi+lo}{2}$ by hi and lo being the values at the front and end of the array.

```
boolean isInBinary(int[] theArray, int iSearch){
  int lo = 0;
  int hi = theArray.length - 1;
  int mid = 0;
  while (hi >= lo) {
    mid = (lo + hi) / 2;    // rounds to highest integer
    if (theArray[mid] == iSearch)
      return true;
    else if (theArray[mid] < iSearch)
      lo = mid + 1;
    else
      hi = mid - 1;
  }
  return false;
}
```

Computing A-Level.

SCC 120 L31 (STRIKE)

Week 16 - 20 covering Abstract Data Types (ADT)

Difference between ADT and data structure is an ADT is defined in terms of it's behavior for the user of the data in terms of values and operations. It doesn't specify implementation details. A data structure is the concrete representation of data from POV of the implementer

Sets -

We have covered these in SCC120 and Maths modules. Used as a familiar way to study difference between abstract and implementation.

API:

```
void add(item)
void remove(item)
bool contains(item)
```

Semantics:

* Every item in the set has to be unique, for this to work items stored in the set must have an equality check.
* The set's contents are unordered

Examples:

A files system is a nested hierarchy of sets.
A table in a relational database is a set of tuples.
A hash table uses set semantics for key/value pairs.

Examples in Lecture

void add(item) checks if item in set, if not adds item to set.

Set Complexity -

The Set ADT has a simple abstraction but the Set implementation is expensive.
Adding a new item requires searching the whole set first.
Same for removing an item and checking the set.
It has O(N) complexity, the larger the set, the longer each operation takes. LINEAR Increase.

Lists -

Workhorse of most programming tasks for storing data.
Assume a list is ordered, we retrieve data in an ordered way at which it was added to the list.
API:

```
void add(item)
void remove(item)
item getFirst()
item getNext()
item getIndex(int)
```

Semantics:

Items can appear multiple times.
Includes an iterator state machine via getFirst, getNext to scan over list contents; retrieval order should match addition order.

List Complexity

We have to decide in advance list size and not be wasteful.
Adding items can be quick if we store list size.
Removing items is O(N)
getFirst, getNext, getIndex all O(1)

Arrays vs Linked Lists -

Simplest way to store and manipulate data is with an array. This is allocated with a contiguous block of memory. They are simple and efficient although allocating / increasing array size is expensive.

The more dynamic data storage is, the less efficient arrays are

A linked list separately allocates memory for each cell in a data structure. We need memory for cell size plus reference to next cell (address).

24/2/2022

SCC 120 L32

ADTs continued

Hash Tables -

Stores a collection of unique keys and values
This concept is used for data storage and access needs.
API:

```
void put(key, value)
value get(key)
bool contains(key)
```

Semantics:

Works like a set for keys except each key has an associated value. This can be of any type and therefore makes this a versatile data structure.

Very common abstraction for data handling.

Theory -

We have a fixed number of 0-indexed buckets which can each store any number of key/value pairs. A key is hashed to an index value of a bucket where the key/value are stored/retrieved.

This mapping is done by a hash function converting key to bucket address. This function must be deterministic (always produces same answer).

The hash function is the main factor of runtime efficiency.

Number of buckets must be fixed but number of items in each bucket is dynamic.

An array represents set of buckets.

Linked List stores keys and values in that bucket

* The more uniform the distribution of keys to buckets the faster lookup of keys are. Worst case: Linear O(N).

Queues -

A queue is first in, first out FIFO. This means it will use two pointers.

API:

```
void add(item)
item remove()
item peek()
int size()
```

Semantics:

Item returned by remove / peek is the first added. This is an ordered list.

Examples:

Commonly used in handling streaming data (packets etc) or in representing real-life queue systems.

Priority Queues -

Discrete Priorities -

We have a fixed number of priority classes e.g "severe", "moderate", "low".

Continuous Priorities -

We assign any number to an item to give it a priority relative to others in the queue.

Starvation = when low-priority items never get completed.

API:

```
void add(item, priority)
item remove()
item peek()
void promote()
```

Semantics:

Item returned by remove or peek is the first item of the highest priority level that has items.

The priority queue is like a set of ordered queues. Used for scheduling. Disadvantage = Starvation

24/2/2022

SCC 120 L32

ADTs continued

Hash Tables -

Stores a collection of unique keys and values
This concept is used for data storage and access needs.
API:

```
void put(key, value)
value get(key)
bool contains(key)
```

Semantics:

Works like a set for keys except each key has an associated value. This can be of any type and therefore makes this a versatile data structure.

Very common abstraction for data handling.

Theory -

We have a fixed number of 0-indexed buckets which can each store any number of key/value pairs. A key is hashed to an index value of a bucket where the key/value are stored/retrieved.

This mapping is done by a hash function converting key to bucket address. This function must be deterministic (always produces same answer).

The hash function is the main factor of runtime efficiency.

Number of buckets must be fixed but number of items in each bucket is dynamic.

An array represents set of buckets.

Linked List stores keys and values in that bucket

* The more uniform the distribution of keys to buckets the faster lookup of keys are. Worst case: Linear O(N).

Queues -

A queue is first in, first out FIFO. This means it will use two pointers.

API:

```
void add(item)
item remove()
item peek()
int size()
```

Semantics:

Item returned by remove / peek is the first added. This is an ordered list.

Examples:

Commonly used in handling streaming data (packets etc) or in representing real-life queue systems.

Priority Queues -

Discrete Priorities -

We have a fixed number of priority classes e.g "severe", "moderate", "low".

Continuous Priorities -

We assign any number to an item to give it a priority relative to others in the queue.

Starvation = when low-priority items never get completed.

API:

```
void add(item, priority)
item remove()
item peek()
void promote()
```

Semantics:

Item returned by remove or peek is the first item of the highest priority level that has items.

The priority queue is like a set of ordered queues. Used for scheduling. Disadvantage = Starvation

28/12/2022

SCC120 L33 STRIKE

Stacks -

Last in, First Out ordered list of items.

API:

void push(item)

item pop()

item peek()

Semantics:

Item returned by pop or peek is the most recent item that was pushed. The stack therefore needs to maintain an ordered list. Push = add Pop = remove

Commonly used in representing the underlying execution of a program and in parsing structured data. Peek = return. Memory can expand and shrink and both adding + removing items is O(1)

Function call stacks -

When you run a program in which one function calls another the compiler arranges these calls in a stack

Reversing a list -

We can easily reverse any list using a stack

3/12/2022

SCC 120 L34

Recursion -

A recursive function is one which calls upon itself to execute the same logic multiple times. The order of executing logic depends whether it's before/after the self-call.

Iterative: Memory usage is controlled explicitly so a "stack overflow" is less likely. Sometimes executes more quickly as no overhead from stack frame creation / destruction. Easier to read / understand / debug.

Recursive: Naturally recursive functions are much more concise Languages which support tail recursion can eliminate some of the extra performance cost and stack overflows.

7/12/2022

SCC 120 L35

Graphs -

A graph is made from a set of nodes and edges.
Nodes (Vertices), Edges (Arcs).
A simple graph is one which cannot have self-loops (a node cannot have an edge pointing to itself).
Both nodes and edges can have values associated with them. A graph can be undirected or directed.
Undirected = can be traversed in both directions.
A sub-graph is a graph formed from a node subset and edge subset.
A spanning sub-graph is a sub-graph using all the nodes but not all the edges of the graph.

A complementary graph is any graph with the same nodes derived by creating a fully-connected graph for these nodes and then removing all edges of the other graph.
Graph density $0 \leq d \leq 1$ measures how heavily connected nodes are.
A graph is strongly connected if we can reach any node from any other.

Examples / Use cases -

The internet is organised as a graph with a core of high-performance routers and increasingly cheaper routers at the edge. Routing packets uses a graph.
Transit systems (Underground)
Social Media Follower / Friend system.
Artificial Neural Networks

API:

```
void add-node(n)
void rem-node(n)
void add-edge(n, m)
void rem-edge(n, m)
bool adjacent(n, m)
item[] get-neighbours(n)
```

Semantics:

Nodes are identified by values (names). We shouldn't have nodes of the same value.

Adding Node = O(1) Removing and Adding an edge is O(N). Adjacent is O(N)

A graph can be represented as a two-dimensional array called an adjacency matrix, two cells are always modified when in an undirected graph has an add / removed edge.

01/3/2022

SCC 120 L36

Traversals -

We can traverse a graph in two-main ways: depth-first and breadth-first. In both variants the challenge is how to deal with loops in the graph, which could cause a traversal algorithm to run indefinitely.
We will look at these from directed graphs but principles apply to undirected graphs.

Depth-first traversal.

We start from any node in the graph and try to visit every other node once. To do this we need to record which nodes we've already visited and need a way to order our routes taken from any given node where there are multiple edges available
Algorithm:

```
void traverse(List L, Node N){
    visit(L,N)      //records visit to N, adding N to L.
    for(int i=0; i<N.edges.length; i++){
        if(!N.edges[i].node.visited)
            traverse(L, N.edges[i].node)
        }
}
```

Breadth-first traversal -

We start from any node in the graph and try to visit every other node once. We still need to record which nodes we've visited and a way to order routes from nodes with multiple edges available. The difference is that we follow each edge of the current node by only "one hop" before then expanding our distance to travel two hops away etc.

once we've exhausted all of the links of our current node in the graph we move to the node that was first visited from this one and repeat the process (for one hop).

Algorithm -

```
void traverse(List L, Node N){
    Queue qu = new Queue()
    visit(L, N)
    qu.add(N)
    while(qu.size() > 0){
        Node X = qu.remove()
        for(int i = 0; i < X.edges.length; i++){
            if(!X.edges[i].node.visited)
                visit(X.edges[i].node)
                qu.add(X.edges[i].node)
            }
        }
    }
```

Shortest Path -

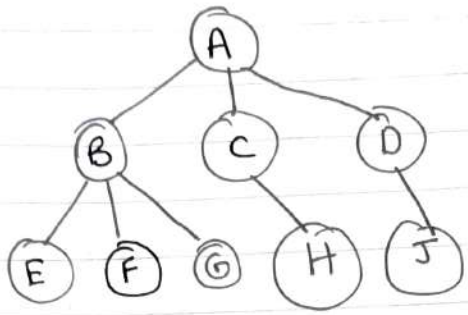
A very useful thing to do with graphs that have edge values is find the shortest path from one node to another. This is calculated by finding the set of edges between the two nodes which have the lowest overall sum value.

13/2022

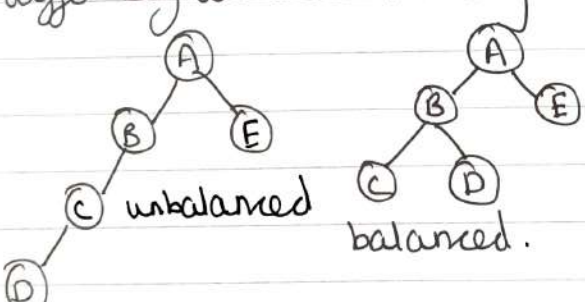
SCC 120 L37

Trees -

A Tree is a connected acyclic (with no loops) graph.



A = root node. B, C, D = internal nodes, E, F, G, H, J = leaf nodes We refer as these having parent and child relationships within the tree. The height of a tree refers to how many levels it has when the root node is level 0. A k-ary tree is one which imposes a maximum number of children at each node in the tree e.g binary trees. For any tree we calculate number of edges by subtracting 1 from the number of nodes |edges| == |nodes| - 1. A sub-tree is formed by taking any node along with all nodes and edges below it. A balanced binary tree is a binary tree which for any node the heights of it's child sub-trees differ by at most 1. e.g



Example uses:

Trees are a common design choice in engineering work as they arrange data and organisations in a hierarchical way which is what a tree structure is e.g the hierarchy of employees at a company and seeing who manages who. A directory file structure on a computer is a tree (tree of sets with unique file/folder names). Computer document formats like XML. A compiler takes syntax code and converts it into an abstract syntax tree.

The Tree ADT -

API:

void add-node(n,m)
void rem-node(n)
void move-node(n,m)
item[] get-children(n)

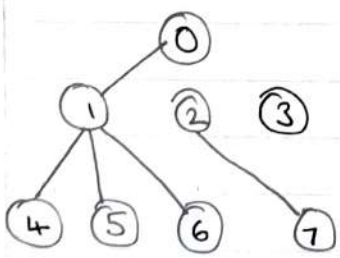
Semantics:

When we add a node(n) we also specify the parent node (m) of which this new node will be a child. The API has no way to control edges (compared to a graph) because the parent/child relationships implied by the tree concept allow all edge calculation to be automated.

A tree is just a restricted kind of graph and so can be represented in an adjacency matrix in the same way as any other graph. eg for 8 nodes:

nodes	0 1 2 3 ... 7 node has an edge to ...
0	
1	
2	
3	
...	
7	

This can then be flattened to form a parent vector e.g



[0,1,2,3,4,5,6,7]
INF 0 0 0 1 1 1 2

22/3/2022

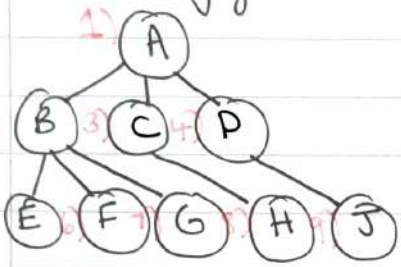
SCC 120 L38

Traversals -

Compared to graphs traversals are easier because there are no loops so we don't need to keep track of which nodes we've already visited.

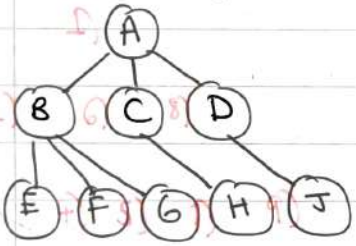
Breadth-First -

Here we traverse the tree from left to right at each level starting from the root

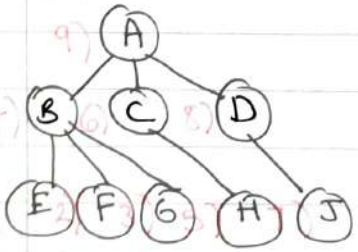


Depth-First (pre-order) -

In this traversal we walk through the tree top to bottom, visiting each node in the same order



Depth-First (post-order) -



Self-Balancing -

Balanced trees provide the property that reaching any leaf node takes the same number of hops.
This is useful for building an index system with a predictable time lookup for any element.
A Binary Tree is an example of a self-balancing tree which is particularly effective for indexing large volumes of data such as a file system and which can grow with new index values as our data grows.

Binary Tree -

We start by defining a maximum number of values that can be stored at any node. When we hit the limit for a node we split the node into children up to the same maximum number. This implies how the depth of our tree evolves. By refactoring the tree depending on the actual values we can provide a much better bound on search time based on the current depth of the tree.
We can further control this search time by deciding on how many values can be stored in each node, which directly correlates with tree depth.

22/3/2022

SCC 120 L39

The efficiency of many algorithms is highly dependent upon the distribution of data in the unsorted list. A new algorithm is recognised as being useful if it has interesting properties relative to particular data distributions or to how it uses memory or CPU Resources.

$O(n)$ = linear $O(n \log n)$ = logarithmic $O(n^2)$ = Quadratic.

Insertion Sort -

Iterates through each element of an array / list and locates where that element should be placed in a new, sorted version of the array. The algorithm is usually done in-place where the sorted region uses the same array as the original data growing as the algorithm progresses (meaning we don't need any extra memory to sort the data).

Properties:

Memory Efficient because it performs sorting in-place
For already-sorted data efficiency = $O(n)$
For reverse-sorted data efficiency = $O(n^2)$
Average-case efficiency = $O(n^2)$.

Selection Sort -

Also uses a concept of sorted and unsorted regions. Finds the smallest remaining element in the unsorted region and places it at the end of the sorted region; this is done repeatedly until there are no more elements remaining in the unsorted region. Similar complexity to Insertion Sort however in practice insertion sort is often faster because it doesn't scan the unsorted array every time to find a minimum value but rather scans only as far as is needed to find an item's correct place.

Best Case : $O(n^2)$ Average Case : $O(n^2)$ Worst Case : $O(n^2)$
In-Place : Yes.

Merge Sort -

Recursively split the array in half until we only have single cells then merge the arrays back together sorting them as we do so. The merge phase takes two sorted arrays and creates a third sorted array. This is done by iterating through each of the two smaller areas at the same time, tracking which item is "next" in each one and moving the smaller of these two items into the merged array. The merge sort works back up from the single-cell arrays which are already sorted because they only contain one item.

Best Case : $O(n \log n)$ Average Case : $O(n \log n)$ Worst Case : $O(n \log n)$
In-Place : No.

28/4/2022

SCC 120 L42 Predicate Logic Cont

Compound Formulae -

"Wizard of Oz is not a SCC 120 student" $\sim S(w)$
"Jay and Kay are SCC 120 students" $S(j) \wedge S(k)$
"Either Jay or Kay is a SCC 120 student" $S(j) \oplus S(k)$
"If Jay is a SCC 120 student then Kay is also" $S(j) \rightarrow S(k)$

Quantifiers -

Quantifiers relate to variables and have two types Universal Quantifier : everything is such that. Existential Quantifier : At least one thing is such that.

UNIVERSAL $\forall x P(x)$ means for all values of variable x in the Universe of Discourse, the formula $P(x)$ is True. e.g for every x $P(x)$ is true

EXISTENTIAL $\exists x P(x)$ means for at least one value of variable x in the Universe of Discourse the formula $P(x)$ is True.
e.g $P(x_1) \vee P(x_2) \vee (P x_3) \dots P(x_n)$

English	Predicate Logic
All P are Q	$\forall x (P(x) \rightarrow Q(x))$
No P are Q	$\forall x (P(x) \rightarrow \sim Q(x))$
Some P are Q	$\exists x (P(x) \wedge Q(x))$
Some P are not Q	$\exists x (P(x) \wedge \sim Q(x))$

$\forall x B(x)$ The negation of this is $\sim \forall x (Bx)$ or $\exists x \sim B(x)$
and the same the other way around for $\exists x B(x)$

De Morgan's Laws for Quantifiers -

$\sim \forall x P(x) = \exists x \sim P(x)$ If not everything is, there exists something that isn't.
$\sim \exists x P(x) = \forall x \sim P(x)$ If something doesn't exist then everything isn't.

Connectives Precedence -

1) Quantifiers : $\forall$, $\exists$
2) $\sim$
3) $\wedge$
4) $\vee$
5) $\rightarrow$
6) $\leftrightarrow$

e.g $\forall x P(x) \vee Q(x) \equiv (\forall x P(x)) \vee Q(x)$

Rules of Inference -

Inference rules for propositional logic : modus ponens, modus tollens, addition, simplification, hypothetical syllogism, disjunctive syllogism, disjunctive syllogism, absorption.

Specific Inference rules for quantified formulae -

Universal Instantiation
Universal Generalisation
Existential Instantiation
Existential Generalisation

Universal Instantiation

$\forall x P(x)$

$\therefore P(c)$

Every man is mortal
$\therefore$ Any specific man is mortal

Universal Generalisation

$P(a)$ for any arbitrary a

$\therefore \forall x P(x)$

Any arbitrary man is mortal
$\therefore$ Every man is mortal

Existential Instantiation

$\exists x P(x)$

$\therefore P(a)$ for some element a

There is someone who is a mortal called a
$\therefore$ a is mortal

Existential Generalisation

$P(c)$ for some element c

$\therefore \exists x P(x)$

c is mortal
$\therefore$ There is someone mortal

Nested Quantifiers, interpretation and satisfiable formulae

Nested Quantifiers: Order

e.g $\forall x \exists y P(x,y)$ The order of quantifiers doesn't matter if the quantifiers are of the same type e.g

$\forall x \forall y P(x,y) \rightarrow C(y,x)$

$\forall y \forall x P(x,y) \rightarrow C(y,x)$ equivalent formula

The order matters if quantifiers are of different types e.g

Assume $L(x,y)$ = x loves y

$\forall x \exists y L(x,y)$ = everybody loves somebody

$\exists y \forall x L(x,y)$ = there is someone who is loved by everyone

An n-place formula is satisfiable if there is at least an n-tuple which satisfies it.

4/5/2022

SCC120 L44

No L43 Due to STRIKE

Time Complexity of Recursive Algorithms -

In order to calculate time complexity of a recursive function we need to define its time complexity function $T(n)$.

This is usually $T_1 + n$ (recursion)

Example - Back Substitution

$T(n-1) = T(n-2) + 1$

$T(n-2) = T(n-3) + 1$

$T(n-3) = T(n-4) + 1$ etc.

Recursion Trees -

A Recursion tree is useful for visualising what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call

Further Examples on Moodle

Master Theorem -

Let $T(n)$ be a monotonically increasing function that satisfies

$T(n) = aT(\frac{n}{b}) + f(n)$

$T(1) = c$

where $a \geq 1$ $b \geq 2$ $c > 0$ and $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ Then

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

You can't use the Master Theorem if

- T_n is not monotone
- $f(n)$ is not a polynomial
- b cannot be expressed as a constant

Further Examples on Moodle

9/5/2022

SCC 120 L45 Linear and Binary Searching

Data Storage and Retrieval -

Storage	Retrieval
Fast: Just append latest data to the end of the store (e.g a heap)	Slow: Linear scan of store required to find item.
Slow: Maintain data in sorted order. Will need to find position in store where data should be placed. Depending on data structure, may need to make room (arrays vs linked list).	Fast: Perform binary search.

Algorithmic Efficiency -

Primary Efficiency:
- The relationship between the amount of data (or data size) and running time of an algorithm
- Time complexity and big-O class.

Secondary Efficiency:
- Reflects the time taken by algorithms to process a single data item.

Efficiency Cases -

How to determine the time complexity (big O class) of an algorithm
1) Identify the worst case
2) Choose a representative operation (RO) in the innermost loop.
3) Estimate how often it will be done as a function of N.

Running Times -

Worst case $W(N) = a * b + c$

primary efficiency (iterations at innermost level) b O(b)
secondary efficiency (how long each iteration takes) a
one-off operations c

Linear Searching -

For an unsorted linear array A containing N integers, find index of first occurrence of integer X. note X may not be present in A.

Need to scan through array elements checking for X and for the end of the array.

OP1 : Check for end of array
OP2 : Check for occurrence of X
OP3 : Increment to the next array element

Complexity is O(N).

Binary Searching -

Can only be used on sorted arrays but has better primary efficiency than linear searching

OP1: Find midpoint
OP2: Compare item to midpoint
OP3: Look in the half which item belongs to.

} Increment

Complexity is O(log2N)

12/5/2022

SCC 120 L46

Indexed Retrieval and Hashing -

An attribute of each object can be used as a Key for storage and retrieval. The Key is a unique identifier for each object. Using keys allows a set of objects / records to be stored in sorted order so binary search can be used in retrieval

Example :

→ some "A" records some "B" records, some "C" records.

A new array can be constructed to index the 1st element of each section.

index[0] holds index of first "A" record
index[1] holds index of first "B" record
" "[2] " " "C" " "

If there are no Keys then index[n] = -1

Indexed Retrieval Linear Seach Efficiency -

Worst Case - O(N) when All keys start with the same letter
Average Case - Quite fast if lot's of small sections

Indexed Retrieval Binary Search Efficiency -

Worst Case - O(log2N) when everything in one section
Average Case - Could be a lot faster than simple binary search

Dealing with collisions using chains Efficiency -

Worst Case : Every object on the same chain O(N)
Average Case : Much faster (same improvement as using arrays)
Best Case : No collisions, retrieval O(1)
Big Advantage : chains are dynamic so extra objects can be inserted easily.

Hashing =

A Hashing Function converts a key (often string) to an integer
Integer is used for indexing a storage array. The same
function is used for storing and retrieving data.
Index integer isn't restricted from 0 - 25 (like letters) so
data can be spread along a longer array. Collisions can
be dealt with by chaining

Designing a hashing function -

You want a wide range of index values and a nice
even distribution to minimise collisions. So use
multiplication and division as well as addition.

16/5/2022

SCC 120 L47 Advanced Sorting

Quick Sort:

The algorithm works by picking a pivot point somewhere in the array and moving the data so that all items less than the pivot are before it and all items greater are after it.
The algorithm then recursively does the same thing on the two halves choosing a new pivot for each of them until there is no more data to sort.

Quick sort has an average complexity case of $O(n \log n)$ same as merge sort. Worst case complexity is $O(n^2)$ worse than merge sort. Quick sort also works in place without using any extra memory unlike merge sort making it a more desirable algorithm.

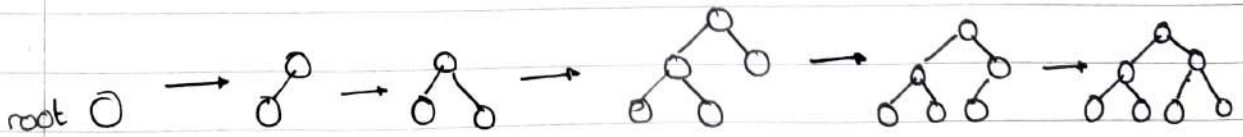
Tree Sort:

Inserts all items into a binary tree using less/greater than comparisons during the insertion to decide which branch to insert into. Once all items are inserted it then uses an in-order traversal to retrieve them resulting in a sorted list.

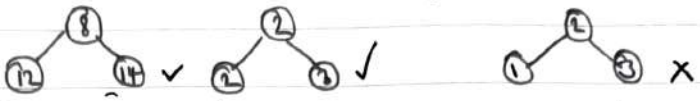
Heap Sort:

The algorithm for this combines the tree concept with a representation of a tree inside an array (this is why the algorithm sorts in-pla

Definition: A left-complete binary tree is a tree where every level is full except potentially for the bottom one which is filled from left to right. Always fills left → right.



Definition: A min-heap is a left-complete binary tree in which no node has a value greater than any of its children.

e.g 

Heap sort maintains a min-heap as items get added and removed. When we insert a node we place it at the next available position which is the right most position of the lowest level. bubbling up is a process to re-establish the min-heap property

G < C so G and C can then switch positions

We retrieve the sorted list of items from the min-heap by removing the root node (lowest value) When doing this replace the root node with the last value in the tree. From there bubble up / trickle down until correct order

Summary =

	Insertion	Selection	Merge	Quick	Tree	Heap
Best	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Average	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Worst	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$
In-Place	Yes	Yes	NO	Yes	NO	Yes

1/10/21

L04

CSS 120 L02

Operations on Binary relations

$R_1 \cup R_2$ is the set of all ordered pairs $\langle a, b \rangle$ that are in R_1 or R_2 or both

$R_1 \cap R_2$ is the set of all ordered pairs $\langle a, b \rangle$ common to R_1 and R_2

$R_1 - R_2$ is the set of all ordered pairs $\langle a, b \rangle$ in R_1 but not in R_2

$R_1 \subseteq R_2$
means R_1 is a subrelation of R_2

Empty relation is a relation with no elements
$\emptyset$
This is a subrelation of any other relation

You can represent a relation using a table with each column being a set.

Digraph - Directed graph consisting of points (vertices and nodes) and arrows (arcs) which connect points to other points.

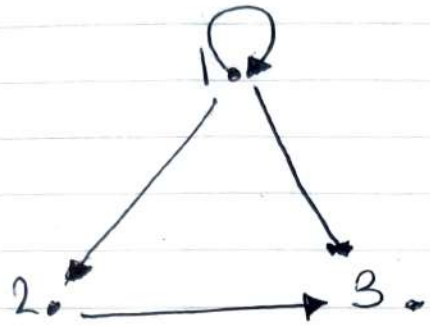
$G = \langle P, A \rangle$
P = set of points
A = set of ordered pairs (arcs)

Digraph

e.g $P = \{1, 2, 3\}$

$A = \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$

$G1 = \langle P, A\rangle$



$\langle p1, p2\rangle$ is an arc from $p1 \rightarrow p2$
if $\langle p1, p2\rangle$ is in R

The purpose is to give a visual representation of relations

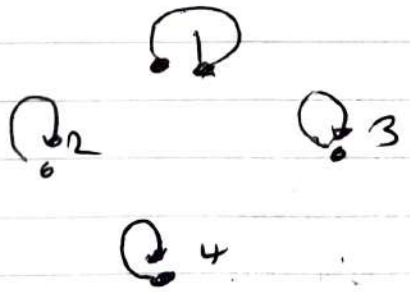
Any relation on A
$R \subseteq (A \times A)$

Examples:

Equal (=)

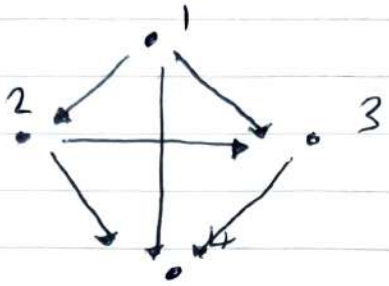
$A \times A = \{\langle 1,1\rangle, \langle 1,2\rangle \ldots \langle 4,4\rangle\}$

$R = \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle, \langle 4,4\rangle\}$



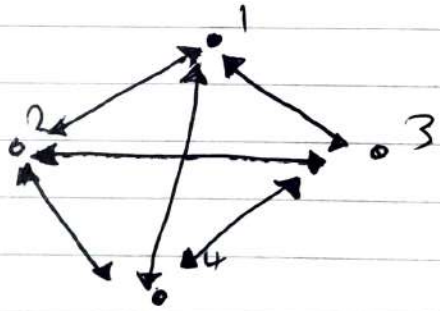
Less than (<)

This time $R = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 3,4\rangle\}$



Example Different (≠)

$R = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,1\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$



Draw two lines not ⟷ in future

Property of Relations

- Symmetry $R \subseteq A \times A$ is symmetric if $\langle a,b\rangle \in R$ then $\langle b,a\rangle \in R$ for every a and b
- Equality Equal elements $\langle a,a\rangle, \langle b,b\rangle, \langle c,c\rangle, \langle d,d\rangle$

25/10/21

LO5

SCC 120 L01

Property of Relations

Symmetry $R \subseteq A \times A$ $\langle a,b \rangle \in R$ then $\langle b,a \rangle \in R$ for every a and b

Equality Equal elements $\langle a,a \rangle, \langle b,b \rangle \ldots$ $\langle a,b \rangle$ such $a=b$

Transitivity $R \subseteq A \times A$ is transitive if $\langle a,b \rangle \in R$ and $\langle b,c \rangle \in R$ then $\langle a,c \rangle \in R$ (carrying over relations)

e.g is taller than. a taller than b, b taller than c therefore a must be taller than c.
($<,>$)

Reflexivity if $\langle a,a \rangle \in R$ for every element a of A

e.g $B = \{5,6,7\}$
$R = \{\langle 5,5 \rangle, \langle 6,6 \rangle, \langle 7,7 \rangle\}$

Irreflexive if ≠ Reflexivity e.g ($<$)

Equivalence - If reflexive, symmetric, transitive

Functions

Input → Function → Output

e.g $f(x) = 2x + 3$

It can be based on a table, formula or graph

Each Input must have a unique output

A relation can be a function. A function is a specific type of binary relation (when it is one-to many etc) ~~can't have~~.

26/10/21

L06

CSS 120 L06

$f: A \to B$ defines a function

A function is a relation if
- for each element a in A there is an element b in B such that $\langle a, b \rangle$ is in the relation
- that element is unique. if $\langle a, b \rangle$ and $\langle a, c \rangle$ are in the relation then $b = c$

Domain = {A}
Co-domain = all possible outputs in this case {B}
Range = set of values that come out of a function (subset of co-domain)

If $\langle a, b \rangle \in f$, then b is denoted by $f(a)$ or $f(a) = b$

a is the pre-image (input) under f
b is the image (output)

Sum of Functions -

Let $f: \mathbb{R} \to \mathbb{R}$
$g: \mathbb{R} \to \mathbb{R}$

$(f + g)(x) = f(x) + g(x)$
e.g $f(x) = 3x + 1$ $g(x) = x^2$ (Intersect Domains)
$= x^2 + 3x + 1$

Difference of Functions

$(f-g)(x) = f(x) - g(x)$

e.g $f(x) = 3x+1 \quad g(x) = x^2$ (Intersection of Domains)

$(f-g)(x) = -x^2+3x+1$

$(g-f)(x) = x^2-3x-1$

Product of Functions

$(f * g)(x) = f(x) * g(x)$

e.g $f(x) = 3x+1 \quad g(x) = x^2$ (Intersection of Domains)

$(f*g)(x) = 3x^3+x^2$

Division of Functions

$(f/g)(x) = f(x) / g(x)$ (Intersection but $x \neq 0$)

e.g $f(x) = 3x+1 \quad g(x) = x^2$

$(f/g)(x) = \frac{3x+1}{x^2} \quad x^2 \neq 0$

Composition

Let $g: A \to B$ Notation: $f \circ g$

$f: B \to C$

$f \circ g: A \to C \quad f(g(x))$

Domain of $f \circ g$ = set of all elements x such that x is in the domain of g and $g(x)$ in domain of f

LO7

SCC 120 LO7 Function Types

Inverse Function

f^{-1} is a function that reverses f.
e.g $f: A \to B$
$f^{-1}: B \to A$
$f(x) = y$ $f^{-1}(y) = x$
Not every function has an inverse function. This is because the inverse function must also have unique x

Surjective Function

$f: A \to B$ is surjective if for each element y of B, there is at least one element x in A that $f(x) = y$
The range of f equals co-domain of f

Injective function

$f: A \to B$ is injective or one-to-one if no y is the image of two distinct x.
This means wherever:
$f(x_1) = f(x_2)$ then $x_1 = x_2$ (unique x)

Bijective function

When a function is both injective and surjective it is bijective. Every bijective function has an inverse function surjective ensures each, injective ensures unique

Propositional Logic

Truth / False, A proposition is either true or false

e.g

A = "It is raining" True = T or 1

B = "I feel sleepy" false = F or 0

Atomic proposition, doesn't depend on the truth or falsity of other propositions

Compound proposition, propositions constructed from Atomic + connectives

These can be shown by truth tables

4/11/2021

L08

SCC120 L08

Fundamental Connectives

P	Q	P ∩ Q	AND
F	F	F	
F	T	F	
T	F	F	
T	T	T	

P	Q	P ∨ Q	OR
F	F	F	
F	T	T	
T	F	T	
T	T	T	

P	Q	P ⊕ Q	XOR
F	F	F	
F	T	T	
T	F	T	
T	T	F	

P	~P	NOT
T	F	
F	T	

P	Q	P => Q	CONDITIONAL
F	F	T	
F	T	T	
T	F	F	
T	T	T	

P	Q	P <=> BI-CONDITIONAL
F	F	T
F	T	F
T	F	F
T	T	T

Operater precedence :

~	NOT
^	AND
v	OR
=>	CONDITIONAL
<=>	BI-CONDITIONAL

8/11/2021

L09

SCC 120 L09

Propositional Logic - Recap L08

Argument - Sequence of propositions that end with a conclusion.
Premises - Basis on which we establish the conclusion.
Conclusion - The claim that we are trying to establish as true.

Written:
Premise 1
Premise 2
Premise n
∴ Conclusion

Rules of Inference:

* Highlight the logical reasoning behind a valid argument
* A series of steps in which each intermediary and the final conclusion are justified by the premises.

If left = true then Q = true

$[(P \Rightarrow Q) \wedge P] \Rightarrow Q$ modus ponens

$[(P \Rightarrow Q) \wedge \sim Q] \Rightarrow P$ modus tollens

$P \Rightarrow (P \vee Q)$ addition

$(P \wedge Q) \Rightarrow P$ simplification

→ means a premise.
$[(P \rightarrow Q) \wedge \sim Q] \Rightarrow P$ means

$$\frac{(P \rightarrow Q) \quad \sim Q}{\therefore P}$$

it also means ⇒ from MATH 111

→ means if P then Q. This is a premise

$[(P \rightarrow Q) \wedge (Q \rightarrow R)] \Rightarrow (P \Rightarrow R)$ hypothetical syllogism

$[(P \vee Q) \wedge \sim P] \Rightarrow Q$ disjunctive syllogism

$(P \rightarrow Q) \Rightarrow P$ absorption

11/11/2021

L10

SCC 120 L10

Categories of propositions:

Tautologies - propositions which are always true, regardless of the truth values in truth tables

Contradictions - propositions which are always false regardless of truth values in truth tables

Contingencies - propositions that are neither tautologies nor contradictions

Equivalent - Two propositions are logically equivalent if they have exactly the same truth value under all circumstances
$P \equiv Q$

Rules of Replacement:

(Rules for manipulating propositions by applying them to parts and replacing such parts with logically equivalent expressions)

$\leftrightarrow$ = bi-implication

T = true
F = false
$\rightarrow$ = implies

$\vee$ = or
$\wedge$ = and
$\sim$ = not

common sense

1) Commutative Law $P \vee Q \leftrightarrow Q \vee P$
$P \wedge Q \leftrightarrow Q \wedge P$ Commutation

2) Associative Law $(P \vee Q) \vee R \leftrightarrow P \vee (Q \vee R)$
$(P \wedge Q) \wedge R \leftrightarrow P \wedge (Q \wedge R)$

Like expanding brackets

Distributive Law $P \wedge (Q \vee R) \leftrightarrow (P \wedge Q) \vee (P \wedge R)$
$P \vee (Q \wedge R) \leftrightarrow (P \vee Q) \wedge (P \vee R)$

4) De Morgan's Law $\sim P \wedge \sim Q \leftrightarrow \sim(P \vee Q)$
Double Negatives $\sim P \vee \sim Q \leftrightarrow \sim(P \wedge Q)$

5) Absorption Law $P \vee (P \wedge Q) \leftrightarrow P$
$P \wedge (P \vee Q) \leftrightarrow P$

6) Identity Law $P \wedge T \leftrightarrow P$
$P \vee F \leftrightarrow P$
Removes function (and ; and (or ;

7) Idempotence Law $P \wedge P \leftrightarrow P$
$P \vee P \leftrightarrow P$ Obvious

8) Negation Law $P \vee \sim P$ = Tautology TRUE
$P \wedge \sim P$ = contradiction FALSE

9) Double Negation Law $P \leftrightarrow \sim(\sim P)$ Obvious

10) Implication Law $P \rightarrow Q \leftrightarrow \sim P \vee Q$ Look at truth table

11) Contraposition Law $P \rightarrow Q \leftrightarrow \sim Q \rightarrow \sim P$ MATH 111 Proof

12) Equivalence Law $P \leftrightarrow Q \leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$
Definition of bi-implication

(Data Structures and Abstract Data Types)

SCC 120 Week 6 (L11) Unit 0: Intro

Basic Concepts

* Facilities and functionality of an abstract data type
* The Data structure employed by the ADT (Abstract Data Type)
* The Implementation of the data structure

Example of an ADT -

A queue (ADT) is defined by the following

Structure:
- Structured as an ordered collection of items which are added at one end (rear) and removed from the other (front)
- Maintain FIFO property (First in, First out)

Operations:
- Queue() creates a new empty queue
- enqueue(item) adds "item" to the rear
- dequeue() removes the first item and returns it
- isEmpty() tests whether Queue() is empty and returns a Boolean value
- size() returns the number of items in queue()

Why is it called Abstract -

* The definition of ADT only mentions what operations are to be performed but not how they will be implemented
* It does not specify how data will be organised in memory and what algorithms will implement the operations
* It is called abstract as it gives an implementation independent view.

Data structure = Physical Representation of the structure of the data being stored.

ADT = Data Structure and procedures / functions which manipulate that data structure

Encapsulation = Bundling of data, along with the methods that operate on that data, into a single unit.

Support for ADT's and Encapsulation =

* Most languages allow us to introduce data structures as user-defined types.
* Some languages allow us to encapsulate the procedures that operate on the data structure.
* If the encapsulation is "strong" then details of how the data structure is stored are hidden from the user and they will have an abstract (conceptual) idea of what the ADT does rather than how it does it.
* The only way to interact with a variable of an ADT is through an interface (set of procedures that operate on the data structure)

Encapsulation is a highly desirable property for Software Engineering

C -

C doesn't offer encapsulation so you have to
1) Declare a new type to model ADT
2) Provide a set of procedures / functions to operate on ADT

Java -

Java does offer encapsulation so an ADT can be defined as a class.
A class contains both the elements of the data structure and the set of procedures / functions (methods) that operate on the data structure.
The elements can be declared as private making them hidden ("strong" encapsulation)

Unit 1 : Simple Arrays

Declare an array in C -

```
int counts [26];
```

- `int` ← data type
- `counts` ← variable name
- `26` ← size (no. elements)

Indexes are used to refer to specific elements of an array. In C it is zero-indexed.

By default in C the type of value used for an index is an int.

Upper and Lower Bounds -

Upper Bound (upb): Index of the last element
Lower Bound (Lwb): Index of the first element

Linear (1-Dimensional) Arrays -

To define the shape you need -
The Type, Variable Name, Size of Array

[3, 8, 7, 2, 1] For example

Array scanning (initialising all elements) can be done with a for loop when debugging to show values every iteration.

BE CAREFUL not to violate bounds of an array index as you will get run-time or compilation errors.

Other Languages -

In other languages we can define upb and lwb.
e.g Pascal

```
ageCounts : array [18 ... 65] of integer;
```

In other languages discrete values can be used as the index type.
e.g Pascal

```
counts : array ['a' .. 'z'] of integer;
```

SCC 120 ~~tests~~ L12

Unit 2 : Strings -

ASCII code encodes 128 specified characters into seven-bit integers
95 printable characters -
Digits, Lowercase, Uppercase, Punctuation
33 non-printing control codes -
(For teletype machines) mostly obsolete except carriage return, line feed and tab code.

Unicode has space for 1,114,112 characters

UTF-8 used with multi-byte encoding for I/O that can represent every unicode char.

Strings are stored in a one-dimensional array of characters.

When coding printf("Hello") this is a string literal and the allocated space given in memory is exactly 5 characters.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	'H'	'e'	'l'	'l'	'o'	','	' '	'W'	'o'	'r'	'l'	'd'	'!'	'\0'

\0 = null byte (byte sized value of zero)

All strings in C are assumed to end in a null byte.

18/11/2021

C String Library -

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

* The null byte is refered to as a terminater
* All strings are character arrays. Not all characte arrays are strings.

Declaration
```
char text[50];
chartext [] = "This is a string"
```

Copy
```
strcpy (text, "hello Mum");
```

Concatenation
```
strcat (name, "Johnson");
```
(result placed in first parameter)

Length
```
unsignedsize = strlen(text);
```
strlen("Hello") = 5

Comparisons
```
int res = strcmp(s1, s2);
```
1) Returns negative if $s_1 < s_2$
2) Returns zero if $s_1 = s_2$
3) Returns positive if $s_1 > s_2$

```
int res = strncmp(s1, s2, n);
```
Compares only first n digits

Scanning by length -

```
char cString[100] = "hello";
int count = strlen(cString);
for (int i=0; i<count; i++)
    printf("%c", cString[i]);
```

scanning with a terminater.

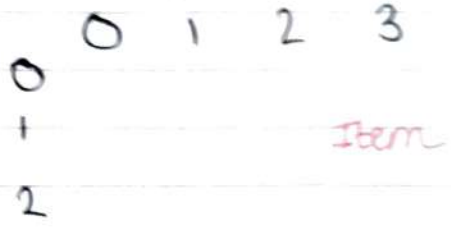
Change for loop to
```
while (cString[index] != '\0')
```

SCC 120 L13

Unit 3: Two Dimensional Arrays

myArray[1][3]

row column



Examining elements of a 2D-Array

```
void printStats() {
  for (int r=0; r< some value ; r++)      ROWS
    for (int c=0; c < some value2 ; c++)  COLUMNS
      if (stats[r][c] != 0)
        printf("%d, %d = %d\n", r, c, stats[r][c
}
```

This prints all non-zero elements in a 2D array

Arrays Pros -

- Single name represents multiple data items of same type
- Random access very fast at accessing stored values.

Arrays Cons -

- Fixed size
- Insertions and deletions are costly

Manifest constants -

```
#define NUMBER_OF_TEAMS 20
```

Constant not a variable

Only one value then needs changing

25/11/2021

SCC 120 L14

Unit 4: Multi-Dimensional Arrays -

Previously 2D arrays are thought of as rows and columns. When thinking about multi-dimensional arrays it is useful to think of the indices as acting like co-ordinates in a graph.

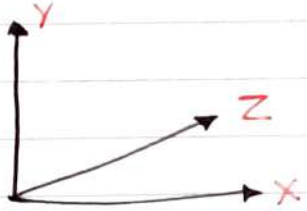
```
int rating [3][4]
```

no. of elements x-axis

no. of elements y-axis

3D Arrays -

```
enum colour {orange, green, blue, red, white};
enum colour box[5][4][3];
```
declared



Unit 5: Memory and Pointers -

Decimal to Hex -

10 - A, 11 - B, 12 - C, 13 - D, 14 - E, 15 - F,

Binary to Hex -

1) Use Nibbles

A Byte is 8-bits long

Computer memory is usually byte-addressable. It can be thought of as an array of bytes with each memory "cell" having a unique address

Think of this like a character array where the address is the index of the element (memory cell).

24/11/2021

SCC 120 L15

Unit 5: Memory and Pointers CONT -

Words -

* Word size can differ from machine to machine but is usually assumed to be 32 bits.
* Integer values are usually word size.

e.g Memory in terms of "words"

Address	Memory
40 30 04	Ø
40 30 08	6c 6c 65 48
40 30 0c	68 74 20 6f
40 30 10	2c 65 72 65
40 30 14	6c 6f 66 20

In HEX

[32 bits wide]

Notice because 4 bytes it goes up by 4.

04 → 08 → 12 → 16 → 20 → 24

04 → 08 → 0c as c is 12 in Hex and so on.

0c → 10 → 14 → 18 → 1c

5.12 Variables -

A variable has three aspects:
Name, Value, Address in memory.

5.18 Pointers -

some pointer code

```
char val; char *addr;

void go() {
  val = 3;
  addr = &val;
  printf("val = %d, addr = %x\n", val, addr);
  *addr = *addr + 2;
  printf("val = %d, addr = %x\n", val, addr);
```

This program adds 2 to the value of val.

addr = &val; *addr = *addr + 2;

&val means address of val

addr is a pointer to a char.

addr = address

*addr = retrieve value in address

7/27/2021

SCC 120 L16

Unit 5: Memory and Pointers CONT -

Levels of Indirection 5.32 -

Fetch, Evaluate, Store when changing values of variables

A level of indirection is when you fetch another address and therefore need to fetch the value within that next address. The more fetching the more levels of indirection.

Notation:

* A variable is indicated as a pointer with an asterisk (a type modifier)
  * ptr fetches value at ptr
  & ptr fetches address of ptr

Unit 6: Records -

6.2 Entities -

An entity consists of values of different types. Traditionally this is referred to as a record.

A record is a compound item. It contains components of various types. Each component is called a field. Each field is identified using a name and not an index.

Example : Students Record

Fields	Students	Types
name		string
age		int
gender		char
subject		string

Records in C 6.5 -

type definition structure — name of the structure

```
typedef struct student {
    char name [size];
    int age;
    char gender;
    char subject [size];
} Student
```

← name of the type

6.7 "malloc" and "sizeof" -

The malloc function (memory allocation) dynamically allocates memory from an area within the program's space called the heap.

```
Student* pt = malloc (sizeof (Student));
```

Malloc allocates the space needed, returning the base address of the area of memory allocated

6.8 The arrow operator and referencing -

The arrow (->). This code below creates a new Student record by using the previous "newStudent" function.

```
Student* stu = newStudent();
strcpy(stu->name, "James");
stu->age = 18;
stu->gender = 'M';
strcpy(stu->subject, "Maths");
```

6.9 The dot operator:

This is used when accessing the fields.

Instead of stu->age we can do stu.age = 18; for example

6.10 It is possible to then have an array of records.

Unit 7: The Vector ADT -

A vector is a dynamic data structure meaning unlike an array in C it can grow or shrink as required.

List of procedures (loosely based on Java vector class).

```
append(v, datum)
insert(v, position, datum)
removeElementAt(v, position)
size(v)
get(v, position)
```

7.3 The Vector ADT -

Assume we have a vector of integers

Append -

void append (Vector* v, int elem)

If elem is 23

```
        0   1   2
Before {34, 12, 65}

        0   1   2   3
After  {34, 12, 65, 23}
```

Insert -

void insert (Vector* v, int index, int elem)

If elem is 23, index is 1

```
        0   1   2
Before {34, 12, 65}

        0   1   2   3
After  {34, 23, 12, 65}
```

Get -

int get (Vector* v, int index)

If index = 2 res = get(v, 2)

```
 0   1   2
{34, 12, 65}      res = 65
```

Remove element at

void removeElementAt (Vector* v, int index)

removeElementAt (v, 2)

```
        0   1   2   3
Before {34, 23, 12, 65}

        0   1   2
After  {34, 23, 65}
```

Size -
res = size(v)
size = 3
if 3 elements

SCC 120 L17

Unit 8a: Using a List to implement a Vector: "size", "get" and "append"

A vector is an ADT that supports the following operations:

append(v, datum)
insert(v, position, datum)
removeElementAt(v, position)
size(v)
get(v, position)

To implement this functionality we will use a linked list:

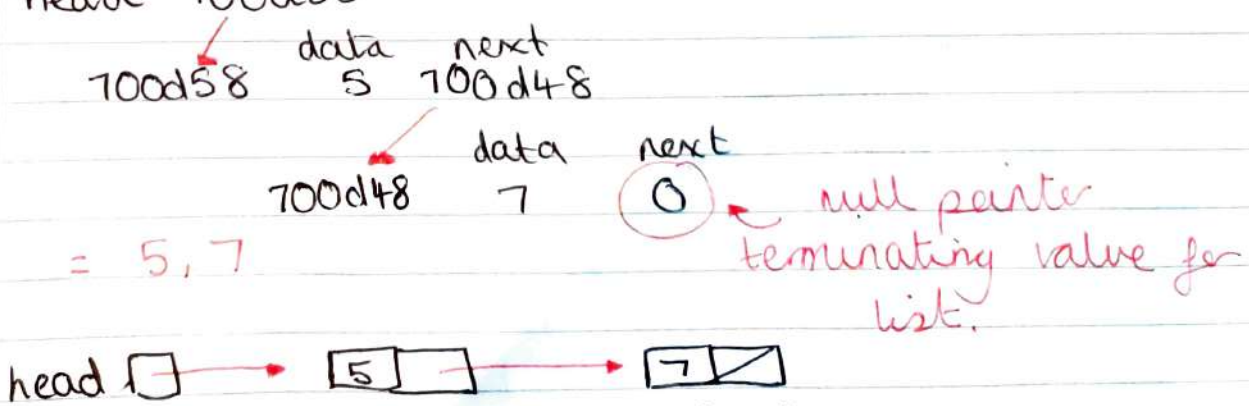
```
typedef struct VectorElem {
    int data;
    struct VectorElem* next;
} VectorElem;

typedef struct Vector {
    VectorElem* head;
} Vector;
```

recursive declaration

head / start of list

Representing a list



most simplified abstract version.

The "size" function :

```
int size (Vector* v) {
  VectorElem* pt = v -> head;
  int count = 0;
  while (pt != NULL) {
    count++;
    pt = pt -> next;
  };
  return count;
}
```

Scans through the list, incrementing the count for each element we encounter until end of list.

pt -> next : next address

pt -> data : data item

The "get" routine

```
int get (Vector* v, int position) {
  if ((position < 0) || (position >= size(v))) {
    return 0;
  };
  VectorElem* pt = v -> head;
  for (int i = 0; i < position; i++) {
    pt = pt -> next;
  };
  return pt -> data
}
```

The "append" routine

```
void append (Vector* v, int datum) {
  if (v -> head == NULL) {
    v -> head = newVectorElem (datum, NULL);
    return; }
  VectorElem* pt = v -> head;
  VectorElem* newElem = newVectorElem (datum, NULL);
```

```
    while (pt -> next != NULL) pt = pt -> next;
    pt -> next = newElem;
}
```

Creating a new VectorElem -

```
VectorElem* newVectorElem(int d, VectorElem* n){
  VectorElem* pt = malloc(sizeof(VectorElem));
  pt -> data = d;
  pt -> next = n;
  return pt;
};
```

Creating the Vector -

```
Vector* newVector(){
  Vector* pt = malloc(sizeof(Vector));
  pt -> head = NULL;
  return pt;
};
```

Initialising Values -

```
Vector* myVector = newVector();
VectorElem* ve = newVectorElem(3, NULL);
ve = newVectorElem(2, ve)
ve = newVectorElem(1, ve);
myVector -> head = ve;
```

SCC 120 L18

Following on from L17 -

Initialising values -

```
Vector* myVector = newVector();
...
ve = newVectorElem(3, NULL);
```

ends linked list

Unit 8c : Using a list to implement a Vector : Insert Element

The Insert routine :

```
void insert(Vector* v, int position, int datum)
```

where we will insert item — item adding

1) Check position is valid (defensive coding)
2) special case (Element added at the start of the list.
3) Add element at position

```
1) if ((position < 0) || (position > size(v)))
       return;
2) if (position == 0) {
       holder = newVectorElem(datum, v->head);
       v->head = holder;
       return;
   }
3) for(int i=0; i< position-1; i++) {
       pt = pt->next;
   };
   holder = pt->next
   pt->next = newVectorElem(datum, holder)
```

holder = temporary vector element to hold data to add to vector.

More pointer to node before position

point to the rest of the list

point to new element.

Element at position -1 points to our new element at position which in turn points at the rest.

The Remove-Element-At Routine -

void removeElementAt (Vector* v, int position)

1) Check position is valid
2) Special case (Element to be removed is at the start)
3) Remove Element
1) The same as the insert routine
2) VectorElem* toBeFreed = NULL; points at element to be deleted

```
if (position == 0) {
    toBeFreed = v->head;        removing element at 0.
    v->head = v->head->next;
    free (toBeFreed);
    return;
};
```

3) Same as insert routine but in a remove type function (combination of both.

3/12/2021

SCC 120 L19

Step 3) of RemoveElementAt

We need to delete the pointer to the element and make the previous element point to the next element after the element

Example without current previous pointer.

```
VectorElem* current = v->head;
for(int i=0; i<position; i++){
  current = current->next;
};
```

current points to element we want to remove.

PROPER ALGORITHM

```
VectorElem* current = v->head,
VectorElem* previous = NULL;
for(int i=0; i<position; i++){
  previous = current;
  current = current->next;
};
previous->next = current->next;
free(current);
```

Deleting from a doubly linked list 13 <-> 22 <-> 45

Remove all pointers to element. Make 45 point to 13 and 13 to 45.

```
1) (current->previous)->next = current->next;
2) (current->next)->previous = current->previous;
```

Unit 9 : Arrays vs Lists -

Static vs Dynamic

Arrays are static, once declared they have a fixed, unchangeable size.

Lists are dynamic, they can grow or shrink as the amount of data being stored changes.

Direct Access -

When you index an array this is transformed into the address where the element with that index actually lives in memory.

General formula - Address of A[K] = base address of A + (K * size of an element (bytes) in A

e.g char message[] = "hello"

address	memory
1000	message[0]
1001	message[1]

Address of message[4]
= 1000 + (4 * 1) = 1004.

Direct Access vs Scanning -

Arrays are indexed, to access the 4th element you can use index 3 and directly access 4.

Lists are not indexed and you must scan through all elements before 4. This takes time

Inserting a value -

Before you can insert a value you need to make room for it. Shunt all elements following the new element to the right. This takes time for large amounts. It is more efficient to do pointer manipulation

	Array	List
Nature	static	Dynamic
Accessing Element	Direct Access	Scan to position
Inserting Element	Slow	Fast (scanning)
Deleting Element	Slow	Fast (scanning)

Unit 10: The Stack

Stack: Data Structure of ordered items. Last in, First Out. First in Last Out.

Push = Append Pop = Remove

Popping when empty = Stack Underflow
Pushing when full = Stack Overflow

16/12/2021

SCC 120 L20

Declaring a stack -

```
int stack [100];
int sp = 100;
```

Pushing -

```
void push(int value){
  sp = sp-1;
  if (sp < 0) return;
  stack [sp] = value;
}
```

This is a stack that goes down. (In memory addresses)

Popping -

```
int pop() {
  if (sp >= 100) return -1;
  int val = stack [sp];
  sp = sp + 1;
  return val;
}
```

L20 Video

There are large programs for creating a stack in c and doing the above.

Stack.h

Unit 11 - Regions of Memory -

When a program is loaded into memory we can think of it as having two sections, Data and Instructions.

Variables -

Global, Local, Pointer variables

Local Variables -

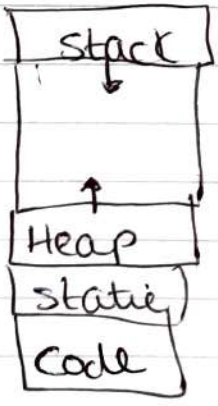
* Created when a procedure / function is called
* Allocated from an area of memory called the Stack
* The Stack is also the mechanism used to support procedure / function calls and return values from a function. It is the same as the Stack ADT (Unit 10)

Pointer Variables -

These are dynamic data from the Heap

Stack (local variables)
Heap (dynamic data)
Global variables (static data)

C:



Stack grows down, Heap grows up

A Stack Frame is an area that stores returns, local variables etc. When a procedure / function is called a new stack frame is created and pushed on the stack. When exiting a function / procedure it is popped

Unit 12: Memory

2D Array Memory Layout

Row-major-order (used by C)

	0	1	2
0	a	b	c
1	d	e	f

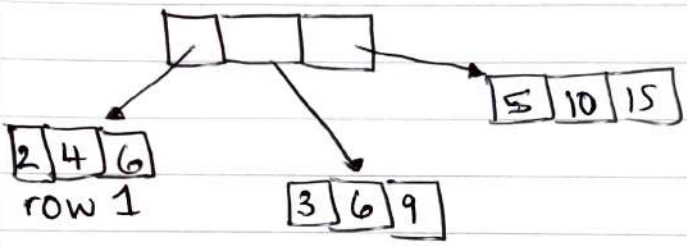
	Base Address
(0,0)	a
(0,1)	b
(0,2)	c
(1,0)	d
(1,1)	e
(1,2)	f

row 0: a, b, c

row 1: d, e, f

There is an alternative linked mapping (used by Java)

Linked Mapping (row)



2	4	6	8
3	6	9	12
5	10	15	20

Also when outputting values of memory arrays you don't need to include boundary values.

Casting (changing type)

For an array int-message:

```
char* char-message = (char*) int-message;
printf("%s\n", char-message)
```

Integer array becomes string of characters

The memory size of a record must always be divisible by 4 so it is stored in a fixed number of words.

7/1/2022

SCC 120 L21 (Introduction to Algorithms)

An algorithm has an input, method and output. They can be represented as code, pseudocode, flowcharts etc.

Algorithms require resources during computation to solve problems.

Space: How much memory is required to solve the problem

Time: How many steps (relative to input size) to solve a problem.

Performance analysis = Actual running time on a computer.

Space Complexity = Amount of memory required by an algorithm to run to completion.

Space Complexity

Fixed Part: Size required to store data / variables, independent of the size of the problem.

Variable Part: Space needed by variables, whose size is dependent on the size of the problem.

Time complexity is usually the most crucial way to evaluate algorithms due to memory advancements.

SCC 120 L22 (Introduction to Operation Counting)

In order to compare two algorithms the same hardware and software must be used.

Operation Counting = Count how many operations a piece of code will execute when ran.

Worst-case scenario (most often used) refers to the most operations needed to complete.

Cost of Operations -
The number of CPU instructions needed for each language depends on the compiler of the language and the available CPU Instruction set.

e.g Pascal requires 5 operations for each array access whereas C requires 2.

We ignore these details of the cost of operations and solely analyse the algorithm.

SCC 120 L23

• = operation

```
var M = A[0];
for (var i = 0; i < n; ++i){
  if (A[i] >= M){
    M = A[i];
  }
}
```

Worst case scenario A = [1,2,3,4] so T(n) = 3 + 4n because 3 operations before going around for loop and 4 operations for going around for loop.

Examples of Operation Counting and T(N)
Case 1: Constant
Case 2: Linear Average of N integers
Case 3: Linear - find minimum of N integers
Case 4: Logarithmic Case 5: Quadratic, Cubic
Case 6: Exponential

Case 1 occurs when the best case of program time and worst case are the same and so the time taken by a program is independent of input size. e.g finding average of a list of 5 inputs. If more than 5 inputs it terminates at the 5th.

Case 2 has best case and worst case the same but is dependent on input size. e.g 3N + 6.

Case 3 has different best case and worst case but is still directly proportional to the size of input (like case 2).

Case 4: Logarithmic $b^c = a \quad \log_b a = c$ as input size increases the program time logarithmically proportional. Best case and worst case are the same

Case 5: Quadratic, Cubic Best case and worst case the same e.g $3N^2 + 4N + 3$. Program time is quadratic / cubic to input size. (Almost Exponential)

Case 6: Exponential Time gets very large for even small values of N.