DSA Review 1 – course goals

Introduce ADTs	(set, sequence, tree, graph)
Introduce algorithms	Dijkstra, Floyd, Warshall, Prim, …
Introduce concepts	abstraction, collections, modelling
Introduce definitions	definition➔ implementation
Implement these ideas	Lab exercises
Explore abstraction	ADTs, abstract programming, models
Provide a “**mental toolbox**”	thinking tools
Develop your understanding of computer science	

Data Structures & Algorithms

ADTs	**set, sequence, trees, graphs** – definitions & ops
Set	**unordered** collection of **unique** elements
Bag	**unordered** collection of elements
Sequence	**ordered** collection of elements (**possibly sorted**)
BT	**ordered hierarchical** collection of elements **(LC,RC)**
BST	**sorted** BT
AVL	**balanced** BST **$(\lvert \text{height}(LC) - \text{height}(RC) \rvert) < 2$**
Graph	collection of **vertices Vi** and **edges (Vi, Vj)** • **directed** **(Vi, Vj)** • **undirected** **(Vi, Vj) (Vj, Vi)**
Operations	**add, find, remove, cardinality, display + algorithms**

Recursion – sequence

- **Seq ::= Head Tail | empty; Head ::= element; Tail ::= Seq;**
- De-construction functions **head: seq → el; tail: seq → seq**
- (Re-)construction function **cons: head x tail → seq**
- Operations – cardinality & add **(empty + non-recursive + recursive)**

```
static listref size(listref L) {
 return is_empty(L) ? 0 : 1 + size(tail(L));
}
```

model

(1) Empty case
(2) Non-empty case **head**
(3) Non-empty case **tail** **(rec)**

```
static listref add_val(listref L, valtype v) {
 return is_empty(L)                  ?  create_e(v)
      :  v < get_value(head(L))      ?  cons(create_e(v), L)
      :                                 cons(head(L), add_val(tail(L),v));
}
```

Recursion – BT (Binary Tree)

- **BT ::= LC N RC | empty; N ::= element; LC, RC ::= BT;**
- De-construction functions **N: BT ➔ el; LC: BT ➔ BT; RC: BT ➔ BT**
- (Re-)construction function **cons: LC x N x RC ➔ BT**
- Operations – cardinality & add **(empty + non-recursive + recursive)**

```
static treeref size(treeref T) {
 return is_empty(T) ? 0 : 1 + size(LC(T)) + size(RC(T));
}

static treeref add(treeref T, int v)
{
  return   is_empty(T)              ? create_node(v)
         : v < get_value(node(T))   ? cons(add(LC(T), v), node(T), RC(T))
         : v > get_value(node(T))   ? cons(LC(T), node(T), add(RC(T), v))
         :                            T;
}
```

model

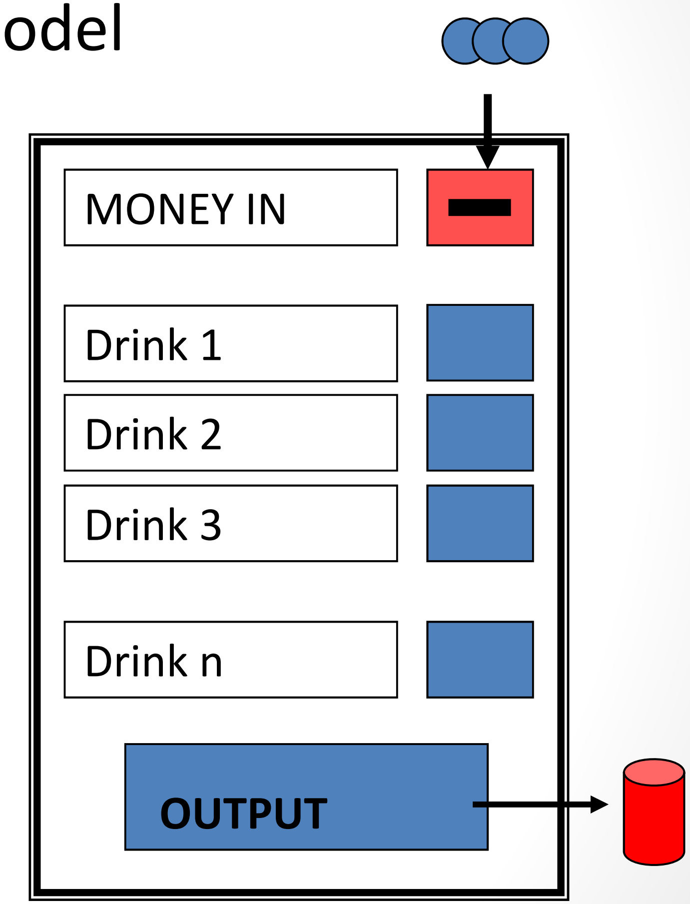
(1) Empty case
(2) Non-empty case **LC (rec)**
(3) Non-empty case **RC (rec)**
(4) Non-empty case **node**

The **ADT** & a Drinks Machine

ADT = Abstract Data Type

- IPO = input / process / output
  - Money in — input
  - Choose drink — process
  - Collect drink — output
- UI (user interface) – **front panel**
- **ADT = a virtual machine**
  - UI – menu based
    - d – display ADT
    - a – add a value
    - f – find a value
    - r – remove a value
    - n – number of elements

- Model



The **ADT** & a Virtual Machine

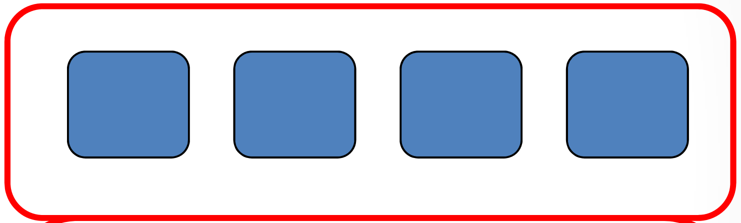
ADT = Abstract Data Type

- UI (user interface) – **menu**
- **ADT = a virtual machine**
- **Menu 🡪 User Dialog**

>enter value:

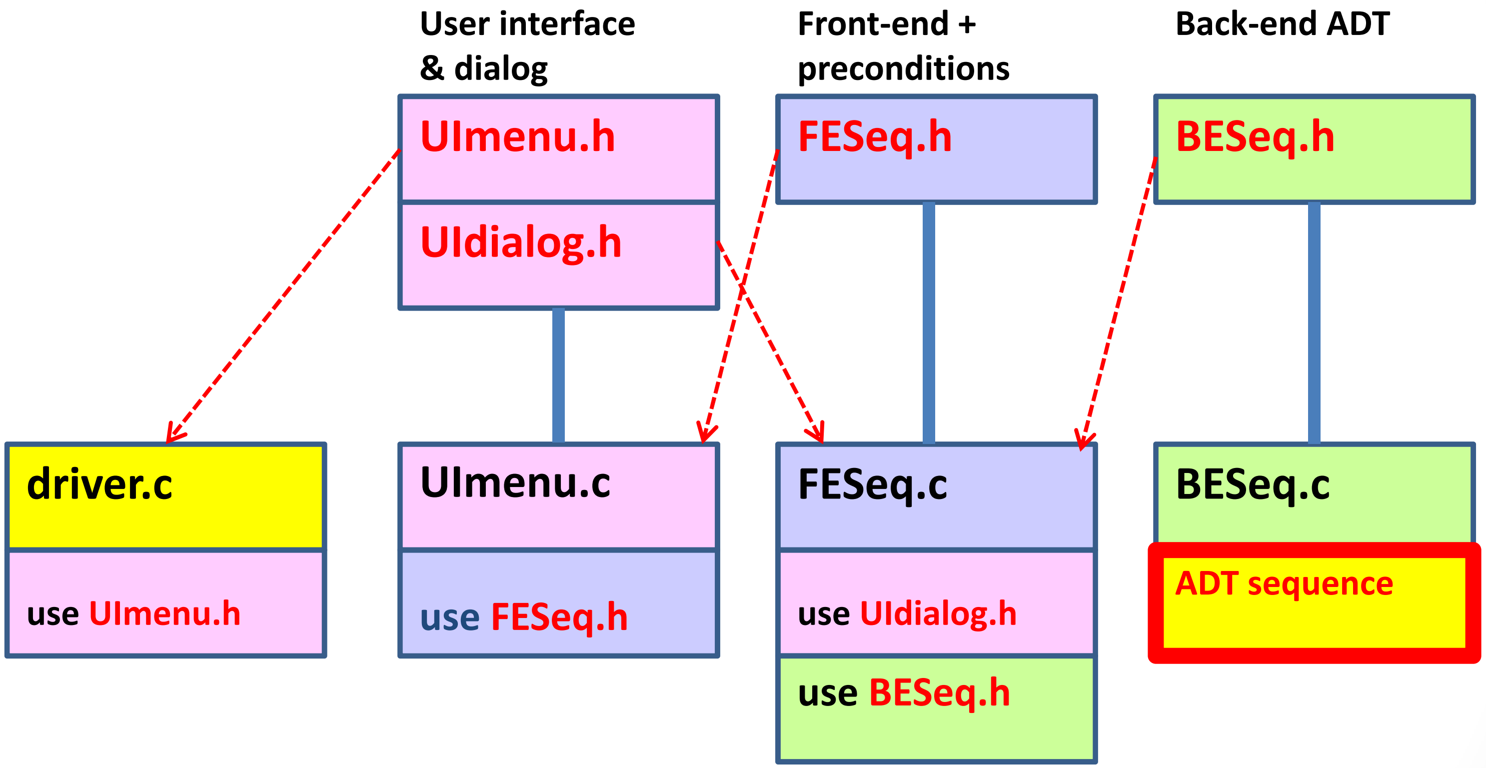
- UI – menu based
  - d – display ADT
  - a – add a value
  - f – find a value
  - r – remove a value
  - n – number of elements (cardinality)

- A sequence
- ADT + operations



D	**display ADT**
A	**add a value**
F	**find a value**
R	**remove a value**
C	**cardinality**

ADTseq – implementation UI/FE/BE



User interface
& dialog

Front-end +
preconditions

Back-end ADT

UImenu.h

UIdialog.h

FESeq.h

BESeq.h

driver.c

use UImenu.h

UImenu.c

use FESeq.h

FESeq.c

use UIdialog.h

use BESeq.h

BESeq.c

ADT sequence

xxx.h == interface
xxx.c == implementation

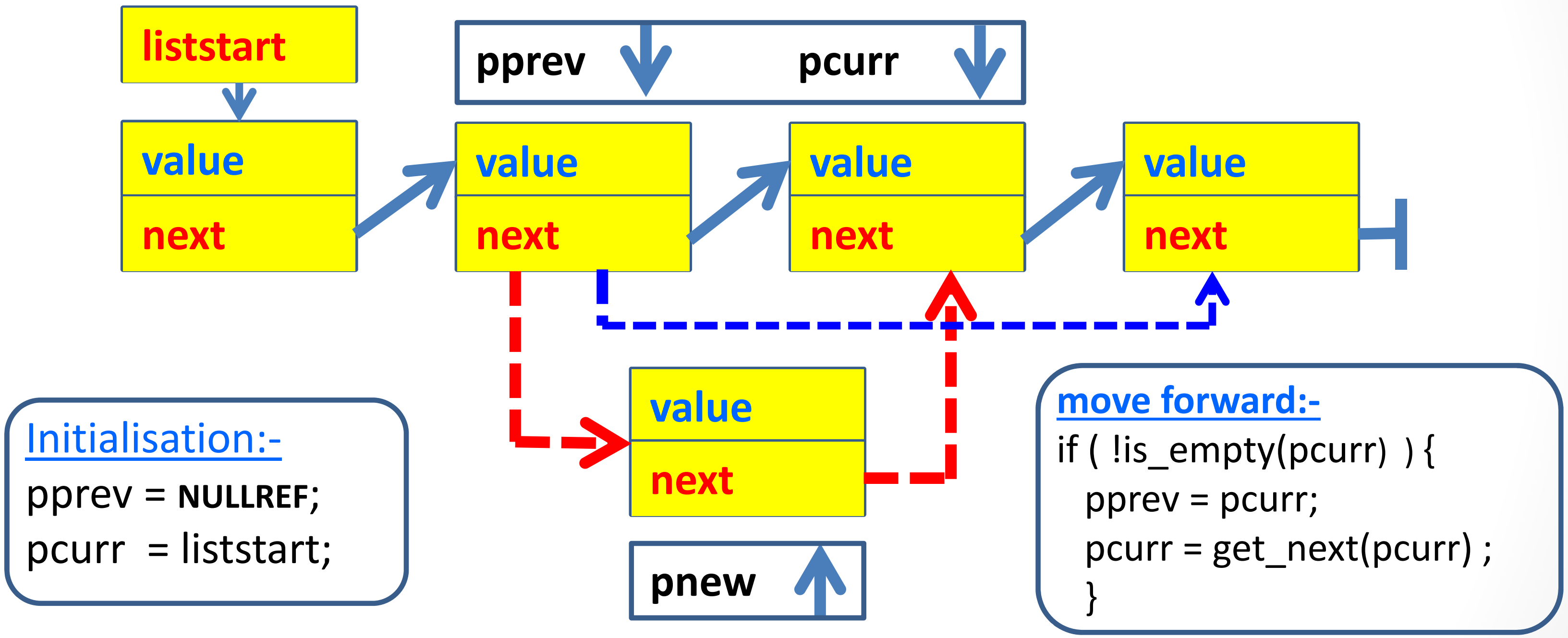
ADTseq – preconditions

size	none	
display	**if size == 0 ➔ empty** else display x x x x x	
find	**if size == 0 ➔ empty** else search ➔ found/not found	
add	none	
remove	**if size == 0 ➔ empty** else search & remove if found	
addpos	**if position not valid ➔ error** else add element at position	**(1..size+1)**
rempos	**if size == 0 ➔ empty** **else if position not valid ➔ error** else remove element at position	 **(1..size)**

The role of pprev, pcurr, pnew

model

```
get_seq_first();
while(!is_empty())  {   process_element();   get_seq_next();   }
```



Initialisation:-

```
pprev = NULLREF;
pcurr  = liststart;
```

move forward:-

```
if ( !is_empty(pcurr)  ) {
   pprev = pcurr;
   pcurr = get_next(pcurr) ;
   }
```

(pprev, pcurr) move as a pair along the list **(used in add/ find /remove)**

pnew is inserted between pprev and pcurr **(used in add)**

Abstraction - Definitions

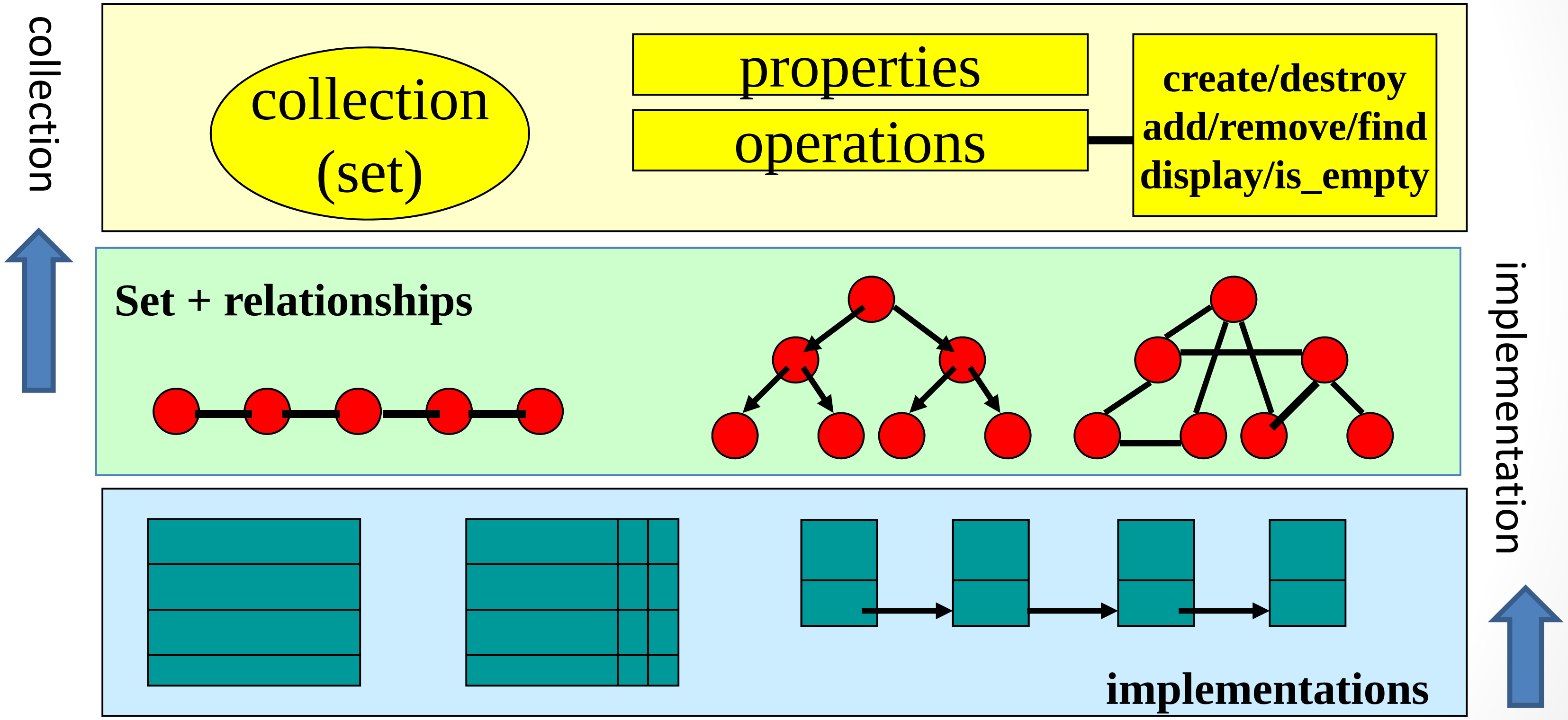
- **Definition 1: Modelling Abstraction**
    - The process of selecting certain properties (attributes) of an entity to model that entity in say a computer program
- **Definition 2: Collection Abstraction**
    - The common properties of and operations on ADTs (set, sequence, tree, graph)
    - is_empty(), add(v), find(v), rem(v), cardinality() (size)
- **Definition 3: ADT (implementation abstraction)**
    - To implement the ADT as an abstract machine
    - i.e. to hide as many of the implementation details as possible

Entities, Collections & Relations (model)

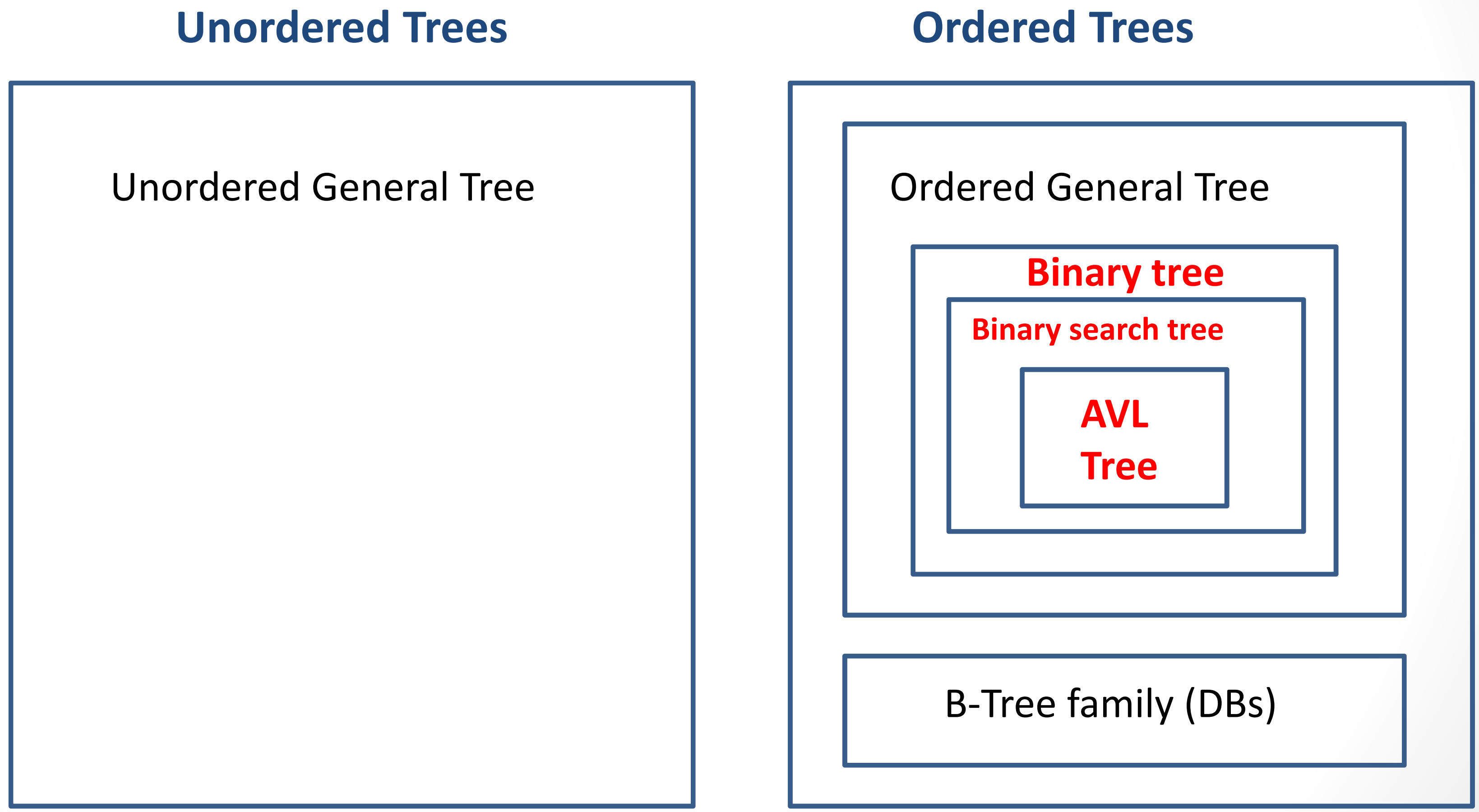
- **Entity / Relationship & Attributes(E/R Model)**
  - abstraction from reality
  - **set of properties** which represent a real life object
  - **student ➔ (name, address, job, personal #, gender)**
- **Collection**
  - a **set** of entities having a common property
  - **all third year students**
- **Relation / relationship**
  - a **property** connecting two entities
  - **mother/daughter, distance between two cities**

Levels of Abstraction

Collection
Implementation



General & Binary Trees



Unordered Trees

Unordered General Tree

Ordered Trees

Ordered General Tree

Binary tree

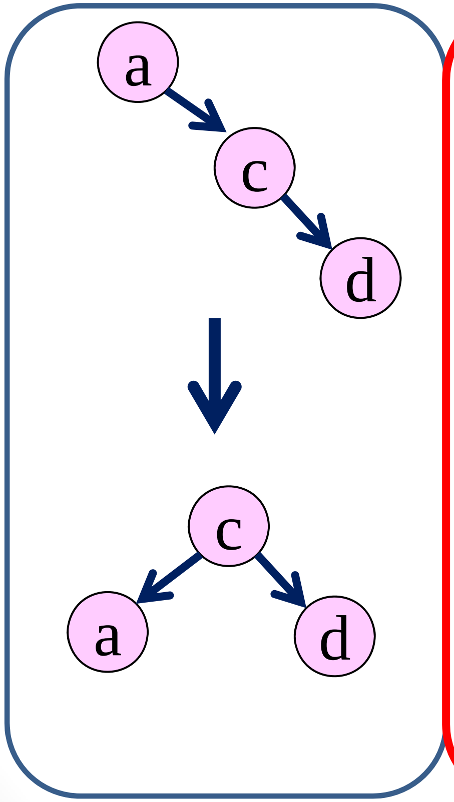
Binary search tree

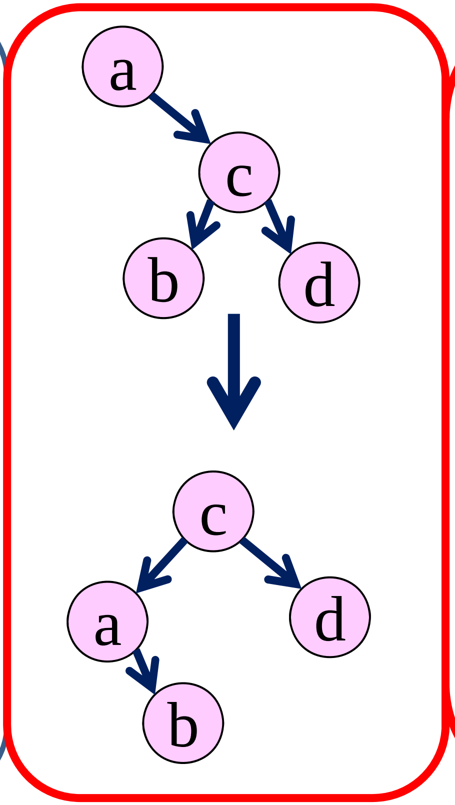
AVL Tree

B-Tree family (DBs)

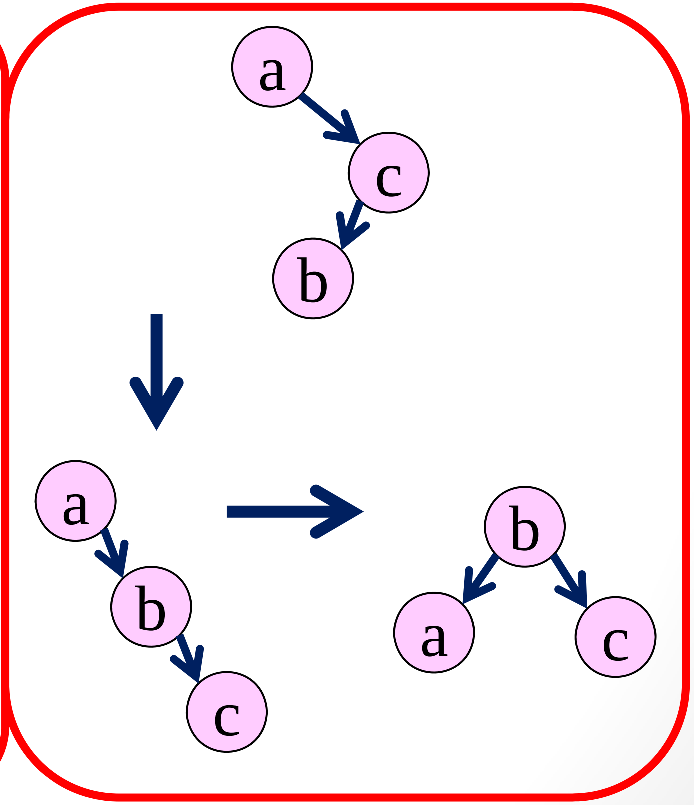
AVL-trees & balancing

- Rotations: SLR, DLR, (mirror images - SRR, DRR)
- SLR(T) (2 cases)





DLR (SRR(RC)+SLR(T))



Rotation (correcting imbalance)

- **Moves the "centre of gravity" from one side of the (sub-)tree to another**
- In order to correct imbalances, there are **4 cases** to consider **(inside/outside).**
- If an imbalance occurs at X, the following may have taken place :
  1. An insertion in the **left sub-tree** of the **left child** of X requires a
     simple right rotation **(SRR – single right rotation)** **outside**
  2. An insertion in the **right sub-tree** of the **left child** of X requires a
     left-right rotation. (**DLR - double left rotation)** **inside**
  3. An insertion in the **left sub-tree** of the **right child** of X requires a
     right-left rotation. (**DRR - double right rotation)** **inside**
  4. An insertion in the **right sub-tree** of the **right child** of X requires a
     simple left rotation **(SLR – single left rotation)** **outside**

  After a rotation the BST-invariant still applies
  - **Value(Left(n)) < Value(n) < Value(Right(n))**
  - **SLR/DLR is mirror image of an SRR/DRR respectively**

The “code”

- **SLR** (+ outside right)

```
RotateLeft(n2)
    n1          = n2.right
    n2.right    = n1.left
    n1.left     = n2
    return n1
    end RotateLeft
```

- **SRR** (+ outside left)

```
RotateRight(n2)
    n1          = n2.left
    n2.left     = n1.right
    n1.right    = n2
    return n1
    end RotateRight
```

- **DLR** (+ inside right)

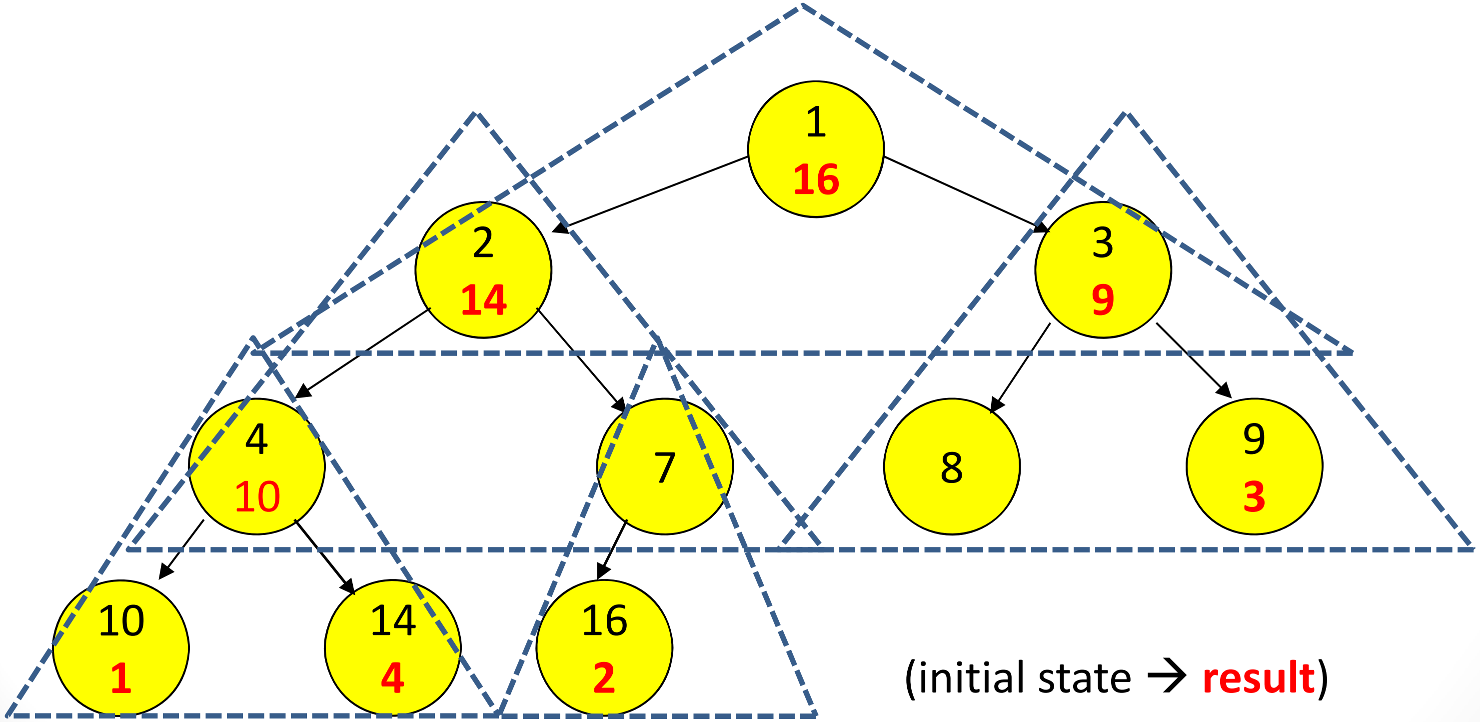
```
RotateDoubleRightLeft(n2)
    n2.right = RotateRight(n2.right)
    return RotateLeft(n2)
end RotateDoubleRightLeft
```

- **DRR** (+ inside left)

```
RotateDoubleLeftRight(n2)
    n2.left = RotateLeft(n2.left)
    return RotateRight(n2)
end RotateDoubleLeftRight
```

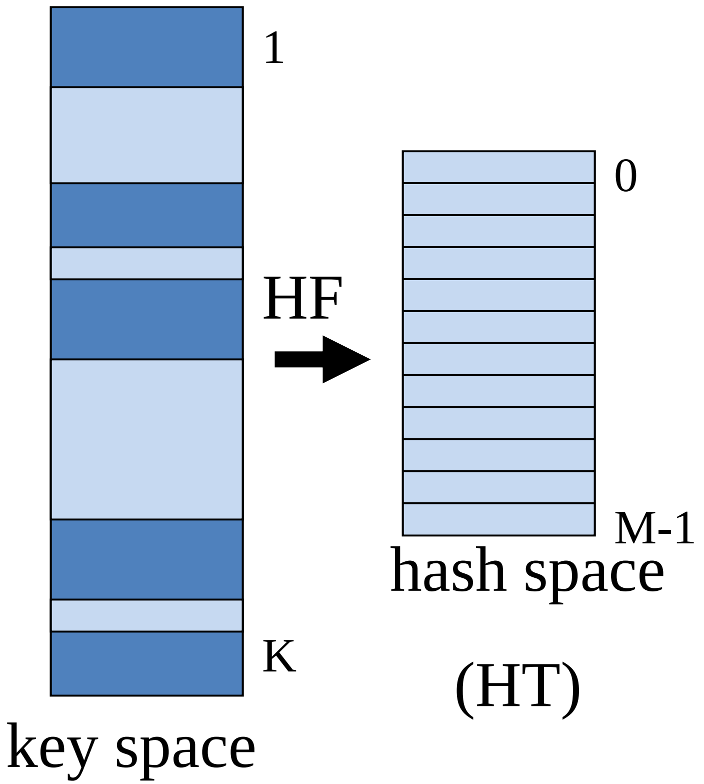
Heap

- **Heapify – parent = max_value(LC,P,RC);   start @ size(H)/2**
- **(16,7,-)→(7,16,-); (10,4,14)→(10,14,4); (8,3,9)→(8,9,3);**
- **(14,2,16)→(14,16,2);   rec (7,2,-)→(2,7,-);**
- **(16,1,9)→(1,16,9);   rec (14,1,7)→(1,14,7); (10,1,4)→(1,10,4);**



(initial state → **result**)

Hashing: Logical Model



- Hashing function, h

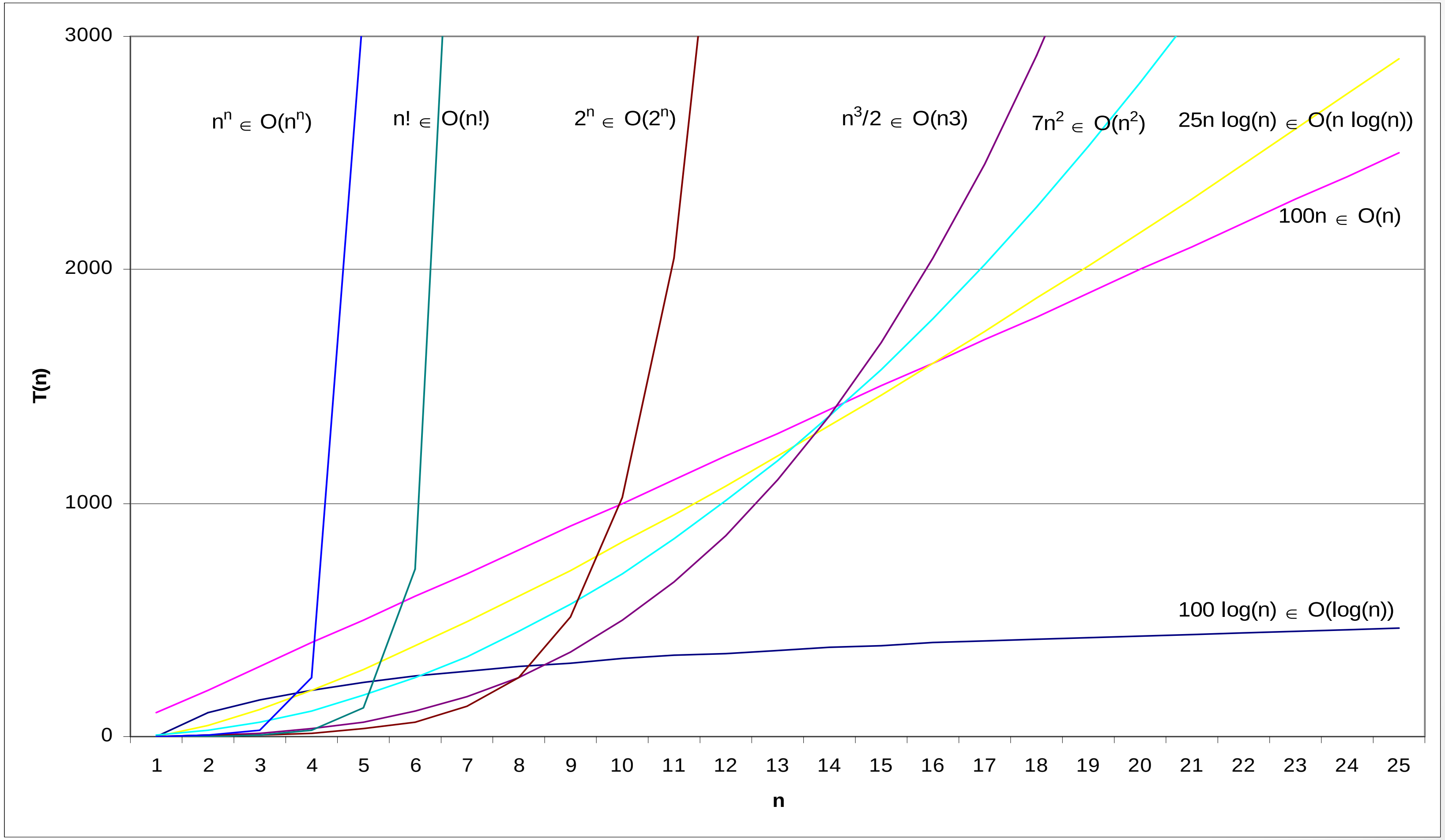
  $0 <= h(key) <= M-1$

- Collision

  $h(key_i) = h(key_j)$ where i != j

- Collision resolution
  - Chaining on collision slot
  - **h(key) + f(i)**
  - **i is the i-th collision**
  - **Linear probing f(i) = i**
  - **Quadratic probing f(i) = i*i**
  - **Double Hashing $f(i) = i*h_2(key)$**

Diagram – Analysis and big-oh **O(x)**



OK – these were the facts!

- … what should we do with these “facts”?

- **The “soft” aspects of the course**
- **Awareness** of the ADTs, operations & use
- **Awareness** of **algorithms** (how to **INTERPRET**)
- **Awareness** of **recursion**
- **ABSTRACTION**
- Improved programming awareness (more efficient)
- (Abstract) **Mental “Toolbox”**
- A **terminology** – easier to **articulate ideas**
- Improved **knowledge** of Computer Science