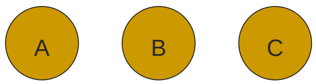
Sequence Overview

- Sequence



- Properties
  - May be empty **else**
  - Every element has a successor (except the last)
  - **Ordered** (position)
- Implementation
  - Arrays
  - Structures & pointers
  - As abstract graphs
    - Adjacency list
    - Adjacency matrix
- See separate document for 7 implementations

Abstract operations - sequence

- create: ¤ ➔ S
- uncreate: S ➔ ¤
- add_el: S x e ➔ S
- rem_el: S x e ➔ S
- find_el: S x e ➔ B
- add_pos: S x e x p ➔ S
- rem_pos: S x p ➔ S
- find_pos: S x p ➔ e
- **cons: S x e ➔ S**
- merge: S_1 x S_2 ➔ S
- concatenate: S_1 x S_2 ➔ S
- sort: S x Rel ➔ S
- is_empty: S ➔ T | F
- is_pos_valid: S x p ➔ B
- cardinality: S ➔ n
- first_e: S ➔ e
- next_e: S ➔ e
- last_e: S ➔ e
- first_p: S ➔ p
- next_p: S x p ➔ p
- last_p: S ➔ p

Preconditions – **sequence S exists**

- **not is_empty: S ➔ B**
  - **remove: S x e ➔ S**
  - **is_member: S x e ➔ B**
  - **first_e: S ➔ e**
  - **last_e: S ➔ e**
  - **first_p: S ➔ p**
  - **last_p: S ➔ p**
  - sort: S x Rel ➔ S
- **not is_empty: S ➔ B and is_member:** S x e ➔ B
  - **find_pos: S x e ➔ p**
- **not is_empty: S ➔ B and is_pos_valid: S x p ➔ B**
  - **insert: S x e x p ➔ S**
  - **find_el: S x p ➔** e
- no precondition
  - **cons: S x e ➔ S**
  - merge: S_1 x S_2 ➔ S
  - concatenate: S_1 x S_2 ➔ S
  - cardinality: S ➔ n

Preconditions – **sequence S exists**

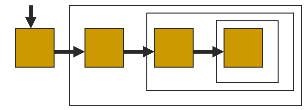
- next_e: S x e ➔ e
  - not is_empty: S ➔ B
    and is_member: S x e ➔ B
    and cardinality: S ➔ n **>** find_pos: S x e ➔ p
- next_p: S x p ➔ p
  - not is_empty: S ➔ B
    and is_pos_valid: S x p ➔ B
    and cardinality: S ➔ n **>** p

Sequence Operations - Complexity

- **create: ¤ → S** **O(1)**
- **uncreate: S → ¤** **O(1)**
- **cons: S x e → S** **O(1)**
- **insert_after: S x e x p → S** **O(n)**
- **is_member: S x e → B** **O(n)**
  (*sorted) **O(logn)**
- **find_pos: S x e → p** **O(n)**
- **find_el: S x p → e** **O(n)/O(1)**
- **merge: S1 x S2 → S** **O(n)**
- **concatenate: S1 x S2 → S** **O(n)**
- **sort: S x Rel → S** **O(n^2)**
  (quicksort) **O(nlogn)**
- **is_empty: S → B** **O(1)**
- **is_pos_valid: S x p → B** **O(n)**
- **cardinality: S → n** **O(n)**
- **first_e: S → e** **O(1)**
- **first_p: S → p** **O(1)**
- **next_e: S x e → e** **O(1)**
- **next_p: S x p → p** **O(1)**
- **last_e: S → e** **O(n)**
- **last_p: S → p** **O(n)**

Sequence: Recursive Definitions

- Sequence view



S ::= H T | ¤
H ::= element
T ::= S

- Quicksort (tree view)

S ::= L p R | ¤
p ::= element
L ::= S
R ::= S

Recursion: sequence view & code

```
List: display_L (List)
{
    if not is_empty (List) {          ← Stop condition
    display_el ( head (List) );       ← Deconstruction H T
    display_L ( tail (List) );        ← Recursive call
    }
    return List;
}
```


Recursion: sequence view & code

```
List: insert (el, List)        /* sorted list assumed */
{   if is_empty (List) return cons(el, List);                          *1*
    if getval(el) < getval(head(List)) return cons(el, List);          *2*
    return cons(head(List),  insert (el, tail(List)));                 *3*
    }
```

1 stop condition → construction
2 insert at head → construction
3 deconstruction (H T) + recursive call + reconstruction

Recursion: Quicksort view

- S ::= L p R | ¤ where L, R ::= S, p = element
- Stop condition |S| = 0 (empty) or 1
- Phase 1 – de<u>cons</u>truction
  - Choose p
  - Construct L (all elements < p)
  - Construct R (all elements > p)
  - Recursive call on L and R
- Phase 2 – re<u>cons</u>truction (from the sorted L & R)
  - **cons**(L, p, R)
- **Exercise**: sketch the (pseudo-)code for the quicksort function

Other aspects

- Hashing
  - Add O(1)
  - Find O(1)
  - Delete O(1)
- Unless collisions have occurred
- Searching
  - Sequential access O(n)
  - Direct access O(1)
- Restricting properties
- Stack
  - Add **at p = first**
  - Rem **at p = first**
- Queue
  - Add **at p = last**
  - Rem **at p = first**
- Other sequence operations may have no meaning **or** a restricted meaning

Language

- Abstract
  - **Implementation independent**
  - **Language independent**
  - ADT = ADS + ops
  - Reference
    - (name, index, pointer)
  - First, next, last
  - Position (in sequence)
- Implementation
  - DT = DS + ops
  - Array
  - Index
  - Structure
  - Pointer
  - Linked list
  - Primitive ops
    - **Built-in knowledge of structure**
  - First = 0 or 1 (or n)

Thought Pitfalls

- Thinking in specific languages (e.g. C/Java)
  - Try to see the abstract model
  - BUT be aware of the implementation correspondences
- It may be possible in the implementation – but is it correct?
  - Position = 1..n ➔ therefore not found = -1 (or null ptr)
  - This turns position into a polymorphic type
    - Position (a legal value in 1..n)
    - Error – which id *NOT* the same ABSTRACT TYPE as position BUT may be implemented with the same <u>implementation type</u> (int)