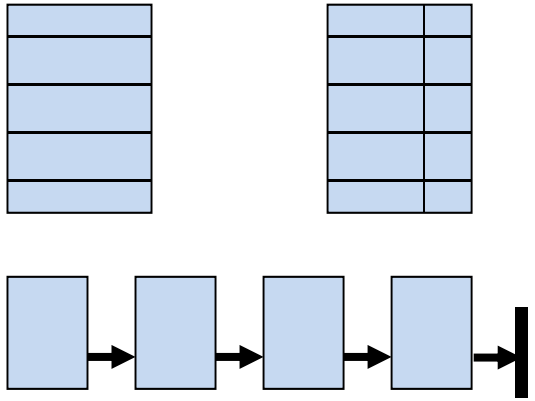
Sequence

- **Properties**
  - Collection (values)
  - Ordered (position)
  - {Sorted by value}
  - Duplicate values
- **Attributes**
  - Value
  - Position

- Visualisations & implementations



Sequence - **importance**

- One of the **basic ADTs**
- **Used to represent Sets & Graphs G=(V,E)**
  - List of lists (adjacency list)
  - Array of arrays (adjacency matrix)
- Good intro to the basic operations on a collection (is_empty, add, remove, find, size)
- Good intro to implementation abstraction (attributes & get/set functions) + **RECURSION**

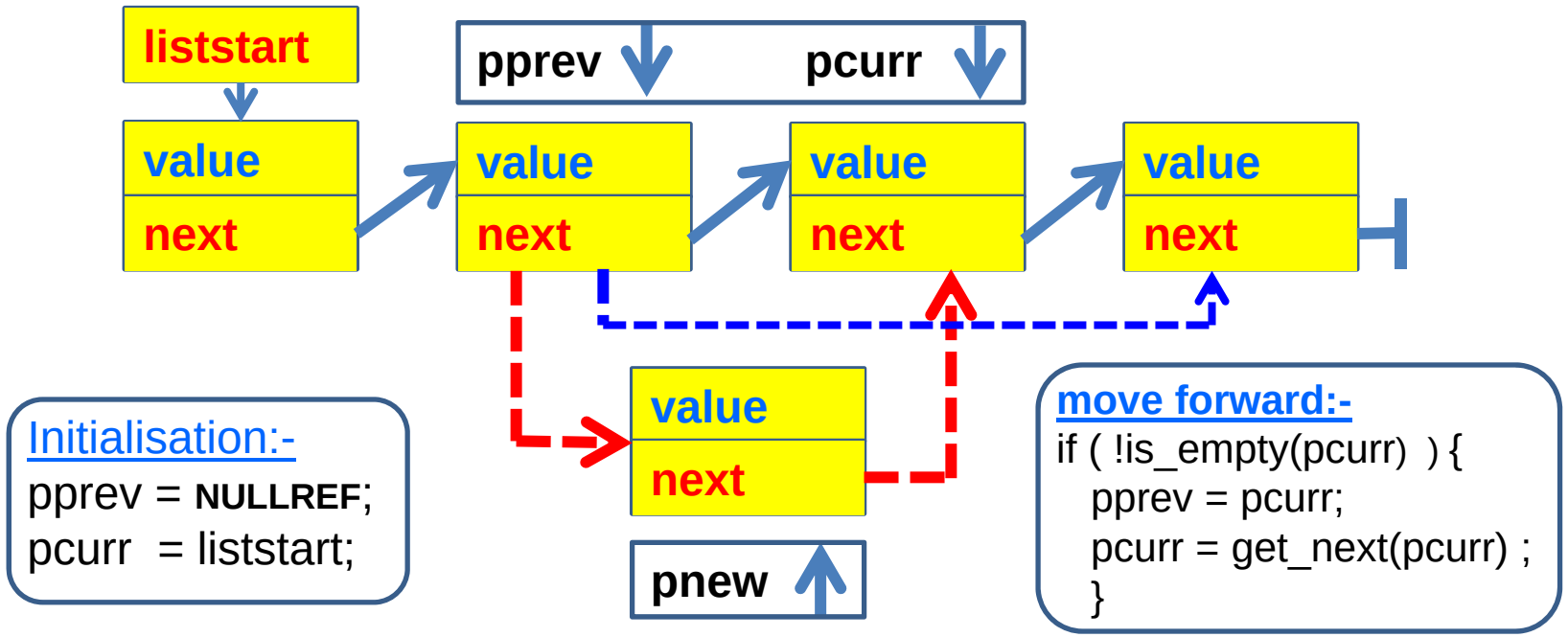
Sequence – **ordered** (position)

- **Position** p must be in range (1..n/n+1)
- **Operations**
  - add_pos(v, p) : **S x v x p → S**
  - rem_pos(p) : **S x p → S**
  - find(v) : **S x v → Boolean**
  - is_empty() : **S → Boolean**
  - size() : **S → integer**

Sequence – **ordered** & **sorted**

- **Operations**
  - add_val(v) **: S x v → S**
  - rem_val(v) **: S x v → S**
  - find(v) **: S x v → Boolean**
  - is_empty() **: S → Boolean**
  - size() **: S → integer**
  - **NB: difference between ORDERED (position) and SORTED (values) (do not confuse these!)**

The role of pprev, pcurr, pnew



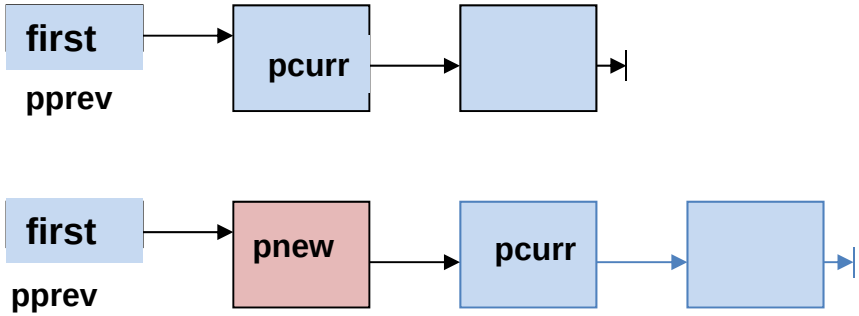
(pprev, pcurr) move as a pair along the list **(used in add/ find /remove)**

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

Sequence – add at position p

p = 1 add at beginning

pnew = element; pprev = null; pcurr = 1

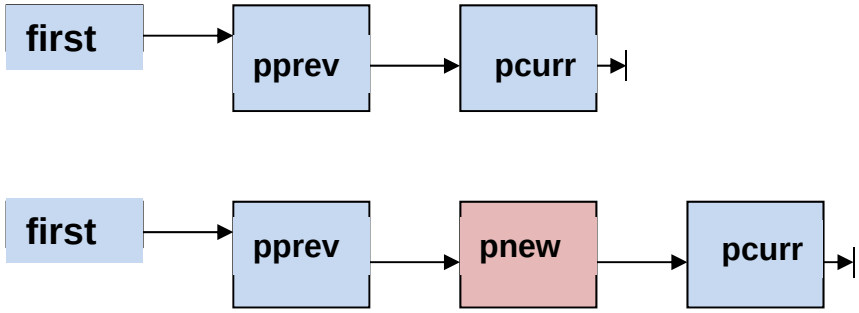


if (is_empty(pprev)) first = pnew; else set_next(pprev, pnew);
set_next(pnew, pcurr);

Sequence – add at position p

p = 2 add in middle

pnew = element; pprev = 1; pcurr = 2

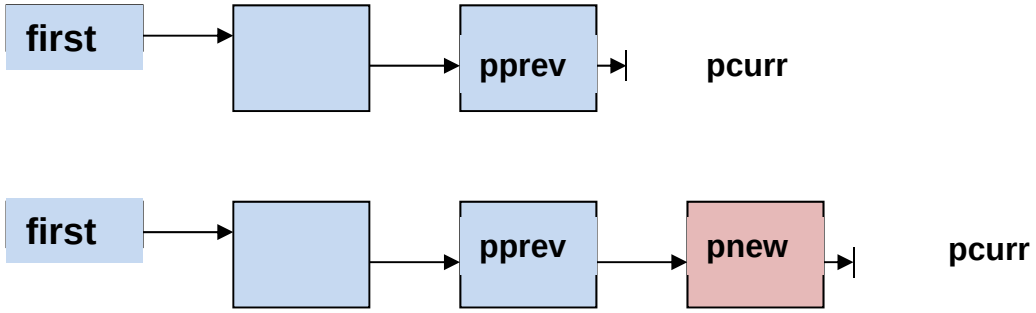


if (is_empty(pprev)) first = pnew; else set_next(pprev, pnew); set_next(pnew, pcurr);

Sequence – add at position p

p = 3 add at end

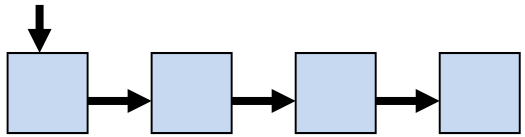
pnew = element; pprev = 2; pcurr = null



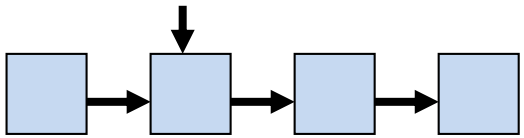
if (is_empty(pprev)) first = pnew; else set_next(pprev, pnew);
set_next(pnew, pcurr);

Sequence: Linked List (implementation)

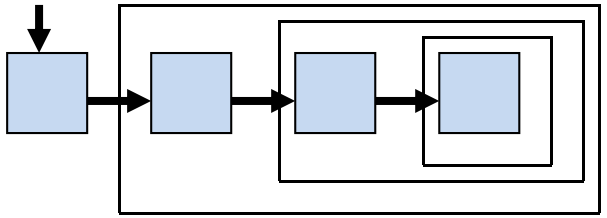
- **Sequential view**



- **pos = 1, 2, 3, 4, ...**
- **first, next(first), next(next(first)), ...**



- **Recursive view**



- **List ::= Head Tail | ¤**

  Head ::= element

  Tail ::= List
- **the “view” is reflected in the program !!!**

Sequence: recursion

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

```
int size(listref L) {
    if is_empty(L) return 0;
    else return 1 + size(tail(L));
    }
```

Sequence: cons

- **Cons**truct a list (add at the **HEAD**)

```
listref cons(listref e, listref L) {
    return set_tail(e, L);
    }
```

- Set the tail of an element (**e**) to a reference to a list (empty/non-empty)

Sequence: add

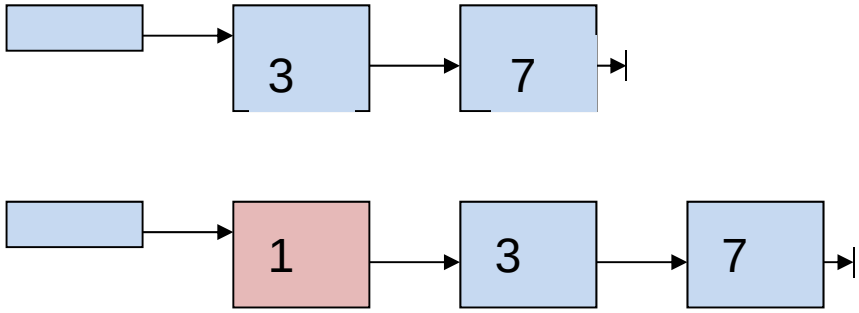
```
listref b_add(int v, listref L)
{
    if (is_empty(L)) return create_e(v);        //1
    else if (v < get_value(head(L)))            //2
        return cons(create_e(v), L);
    else                                        //3
    return cons(head(L), b_add(v, tail(L)));
    }
```

Sequence: add

```
listref b_add(int v, listref L)
{
  return is_empty(L) ? create_e(v)          //1
        : v < get_value(head(L)) ?
          cons(create_e(v), L)              //2
        : cons(head(L), b_add(v, tail(L)));  //3
   }
add at //1 end; //2 beginning; //3 middle
```

Sequence – add

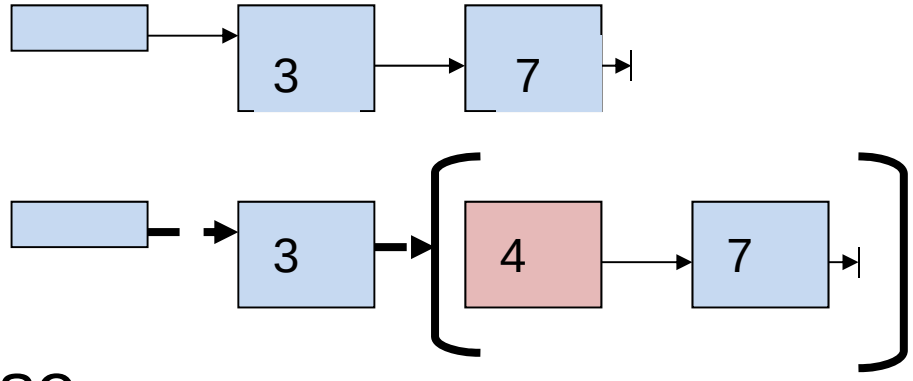
- **add at beginning**




```
else if (v < get_value(head(L)))      //2
        return cons(create_e(v), L);
```

Sequence – add

- **add in middle**

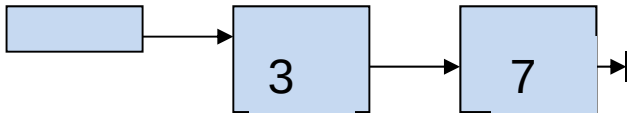


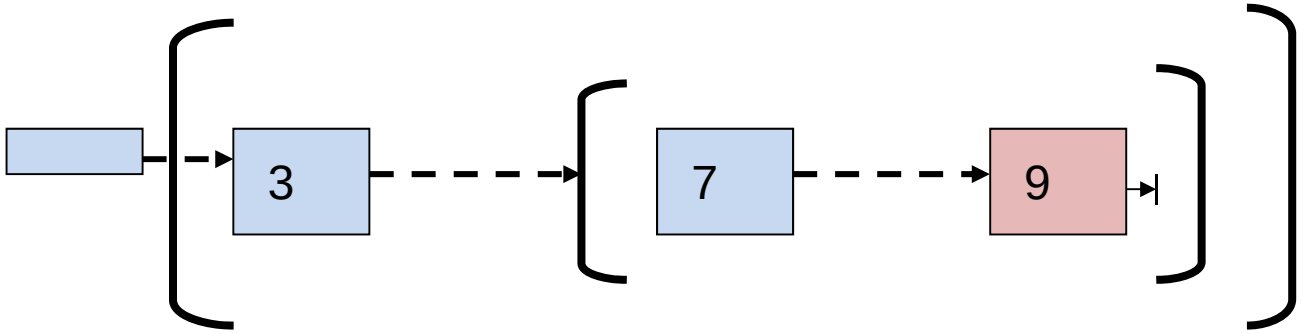
else //3 //2

return **cons**(**head**(**L**), **b_add**(**v**, **tail**(**L**)));

Sequence – add

- **add at end**





if (**is_empty**(**L**)) return **create_e**(**v**); //3 //3 //1