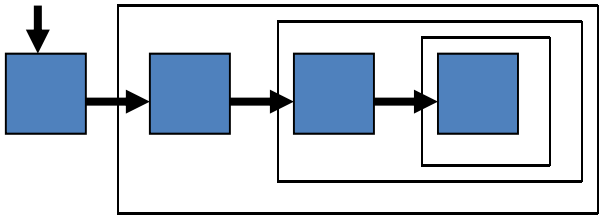
Recursion

- **An entity partially defined** in terms of itself is recursively defined
- **A function which conditionally calls** itself is recursively defined

- Ex: Linked List



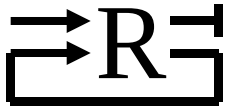
- **List ::= Head Tail | ¤**

  Head ::= element

  Tail ::= List

  - **List is either empty or non-empty**

Recursive Functions

- **Model** 
  - **initial call**
  - **stop condition**
  - **recursive call**

- **Model (pattern) - list**

1. check if **empty** **(stop)**
2. process **head** (non-rec)
3. process **tail** (recursive)

Eg - **display list**

1. empty case
2. head case
3. tail case **(recursive)**

```
List display_L (List L) {
if (!is_empty(L)) {
      display_el (head(L));
      display_L  (tail(L));
      }
return L;
}
```

Recursive / Iterative Comparison

- **Recursive version**

```
List display_L (List L)
{
    if (!is_empty (L)) {
       display_el (head(L));
       display_L ( tail (L) );
       }
    return L;
    }
```

- **Iterative version**

```
List display_L (List L)
{   List ref = first(L);
    while (!is_empty (ref)) {
       display_el (get_val(ref));
       ref = next(ref);
       }
    return L;
    }
```

Ease of Modification

- **Display list**

```
List display_L (List L)
{
    if (!is_empty (L)) {
      display_el (head(L));
      display_L (tail(L));
      }
    return L;
    }
```

- **Display backward**

```
List display_L (List L)
{
    if (!is_empty (L)) {
      display_L (tail(L));
      display_el (head(L));
      }
    return L;
    }
```

Call Sequence: Forward Display

- **Pseudo Code**

```
List display_L (List L)
{
    if (!is_empty (L)) {
      display_el (head(L));
      display_L (tail(L));
      }
    return L;
    }
```

- **List = (A, B, C)**
- 1st call: display_L((**A, B, C**))
  - display_el (**A**)
  - 2nd call: display_L ((**B, C**))
    - display_el (**B**)
    - 3rd call: display_L ((**C**))
      - display_el (**C**)
      - **4th call: display_L (¤)**
        - **STOP / return(¤)**
      - return((**C**))
    - return((**B, C**))
  - return((**A, B, C**))

Call Sequence: Backward Display

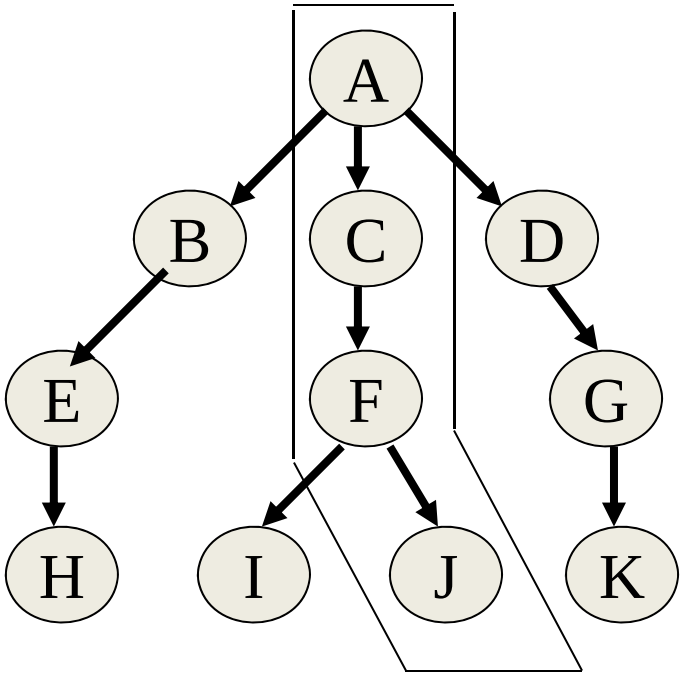
- **Pseudo Code**

```
List display_L (List L)
{
    if (!is_empty (L)) {
      display_L (tail (L));
      display_el (head(L));
      }
    return L;
    }
```

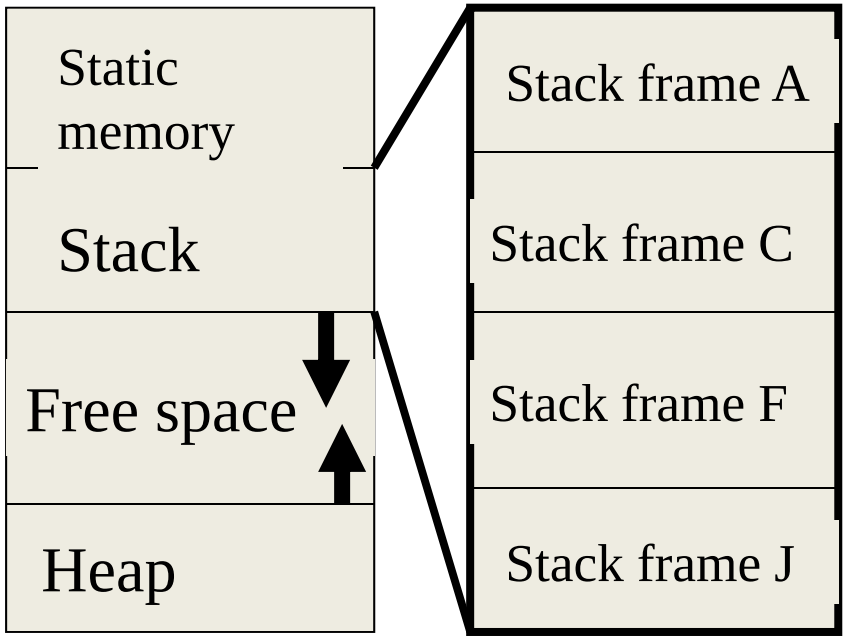
- **List = (A, B, C)**
- 1st call: display_L((**A, B, C**))
  - 2nd call: display_L ((**B, C**))
    - 3rd call: display_L ((**C**))
      - **4th call: display_L (¤)**
        - **STOP / return(¤)**
      - display_el (**C**)
      - return((**C**))
    - display_el (**B**)
    - return((**B, C**))
  - display_el (**A**)
  - return((**A, B, C**))

Calling Sequences

- **Function call sequence**

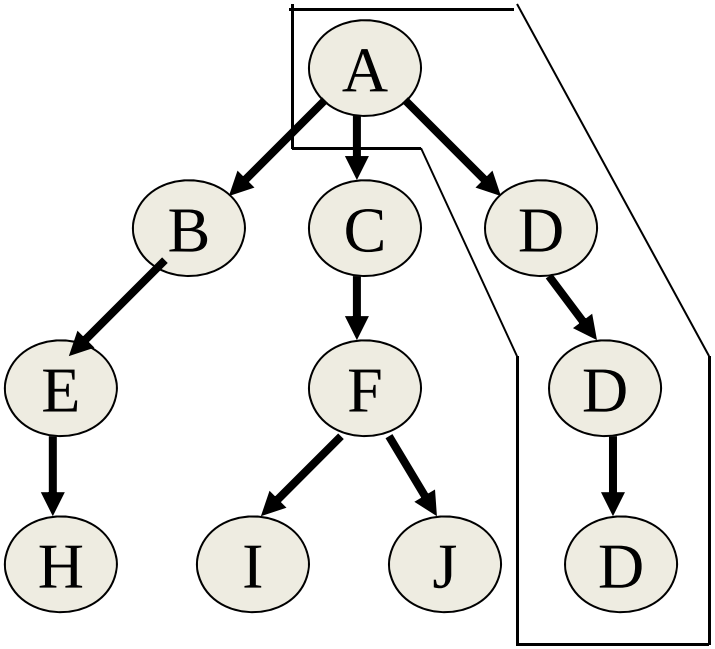


- **Run-time Stack**

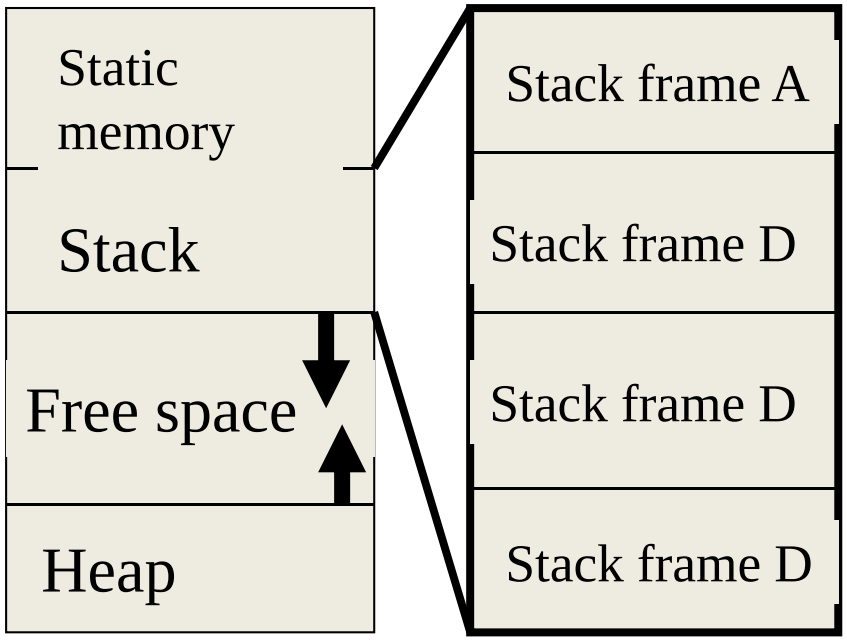


Calling Sequences: Recursion

- **Function call sequence**



- **Run-time Stack**



Linked List functions 1

```
int size (List L)
{    if (is_empty (L))    return 0;
                          return 1 + size(tail(L));
     }
==========================================
List find (valtype v, List L)
{    if (is_empty (L))              return L;   // L is NULL      (F)
     if (v == getval(head(L)))      return L;   // L is not NULL (T)
                                    return find (v, tail(L));
     }
```

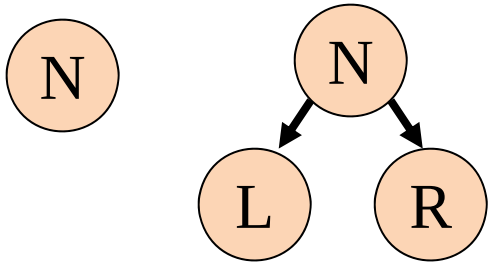
Linked List functions 2

```
List insert (valtype v, List L)          /* sorted list assumed */
{    if (is_empty (L))              return create_el(v);
     if (v < getval(head(L)))    return cons(create_el(v), L);
     return cons(head(L),  insert (v, tail(List)));
     }
==========================================
List delete (valtype v, List L)
{    if (is_empty (L))                    return L;
     if (v == getval(head(L)))            return tail(L);
     return cons(head(L),  delete (el, tail(L)));
     }
```

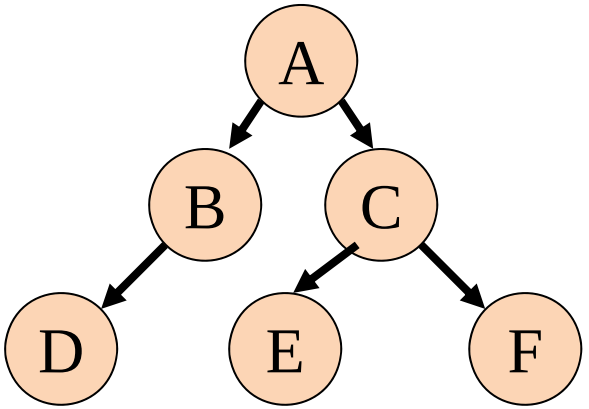
Binary Tree

- **BT Definition**
  - empty
  - Lchild Node Rchild
- **where Lchild and Rchild are BTs**

¤



- **Traversals**
  - Pre-order **N**LR
  - In-order L**N**R
  - Post-order LR**N**



Binary Tree Functions

```
BT: XXX (BT)
{       if not is_empty(BT) { XXX}
        return BT;
        }

XXX = pre_order
                display_node(BT);
                pre-order(left(BT));
                pre-order(right(BT));

XXX = in-order
                in-order(left(BT));
                display_node(BT);
                in-order(right(BT));
```

```
BT: post-order (BT)
{       if not is_empty(BT) {
                post-order(left(BT));
                post-order(right(BT));
                display_node(BT);
                }
        return BT;
        }

Int: size (BT)
{       if is_empty(BT) return 0;
        return 1 + size(left(BT)) +
                        size(right(BT));
        }
```