ECE 220: Computer Systems & Programming

Lecture 24: From C to LC-3 with Linked List

Thomas Moon

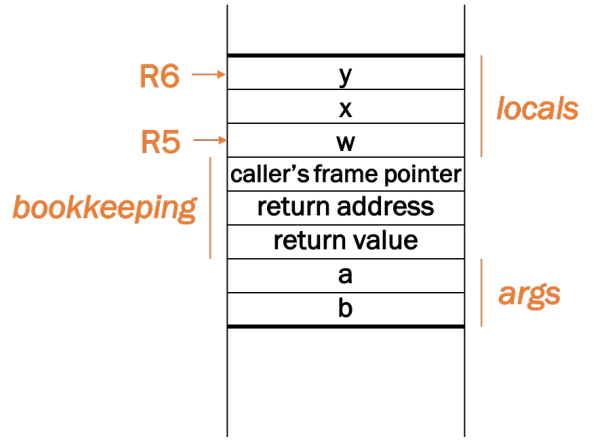
April 18, 2024

Stack Build-up and Tear-down

Caller function	1. Caller setup (push callee's arguments onto stack) 2. Pass control to callee (JSR/JSRR)
Callee function	3. Callee setup (push bookkeeping info and local variables onto stack) 4. Execute function 5. Callee tear-down (update return value, pop local variables, caller's frame pointer and return address from stack) 6. Return to caller (RET)
Caller function	7. Caller tear-down (pop callee's return value and arguments from stack)

```
int Volta(int q, int r)
{
  int k;
  int m;
  ...
  return k;
}
int Watt(int a)
{
  int w;
  ...
  w = Volta(w,10);
  ...
  return w;
}
```

```
Volta
; 3. Callee setup
...
; 4. Exe function
...
; 5. Callee tear-down
...
; 6. Return to caller
RET
```




```
Watt
...
; 1. Caller setup
...
; 2. Pass control to Callee
JSR  Volta
; 7. Caller tear-down
...
```

Caller function	1. Caller setup (push callee's arguments onto stack) 2. Pass control to callee (JSR/JSRR)
Callee function	3. Callee setup (push bookkeeping info and local variables onto stack) 4. Execute function 5. Callee tear-down (update return value, pop local variables, caller's frame pointer and return address from stack) 6. Return to caller (RET)
Caller function	7. Caller tear-down (pop callee's return value and arguments from stack)

Recursive functions in LC3 : Run-time stack

```
int foo(...){
   // base case
   if(...) return something;
    // recursive case
    foo(...);
    foo(...);
    return something;
}
```

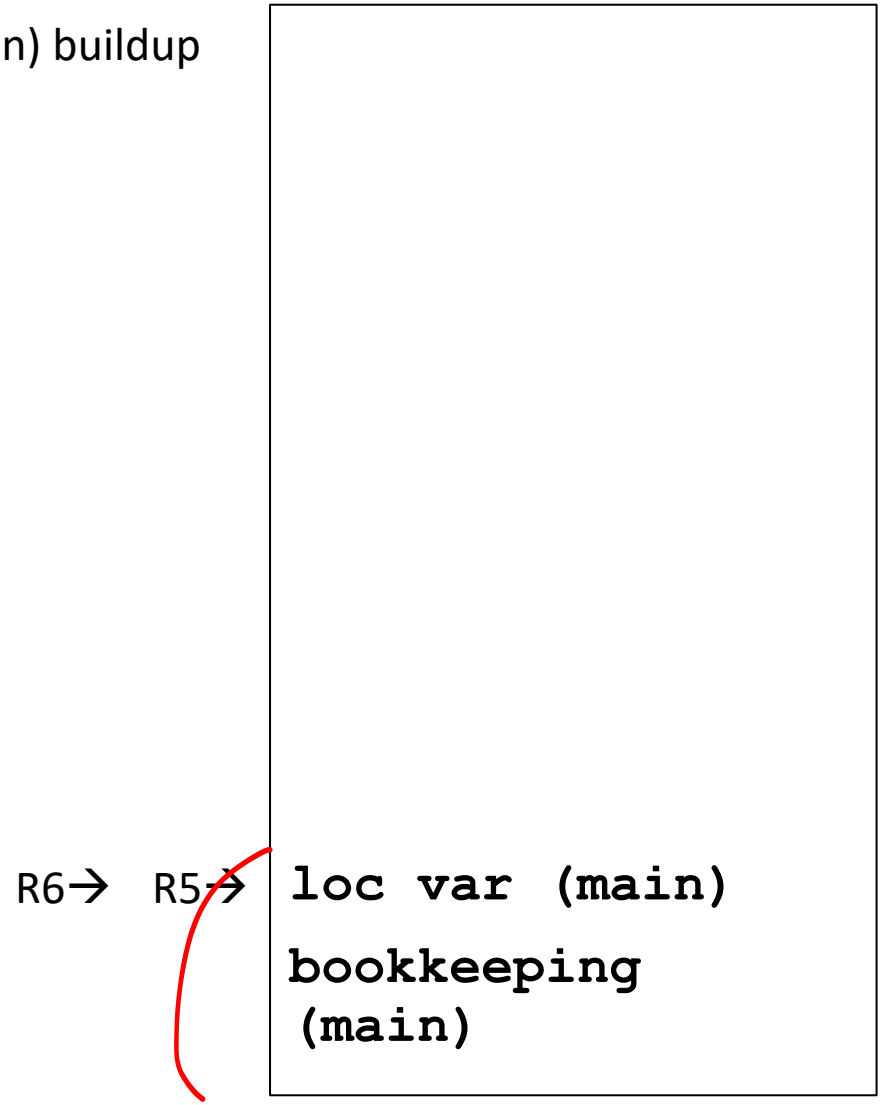
```
FOO
; 1. callee build up
; 2. base cases
; 3-A. caller build up
; 3-B. recursive call
; 3-C. caller tear down
; 4-A. caller build up
; 4-B. recursive call
; 4-C. caller tear down
; 5. callee tear down
RET
```

Example: Foo 2 arguments, 2 local variables

```
FOO
; 1. callee build up
; 2. base cases
; 3-A. caller build up
; 3-B. recursive call
; 3-C. caller tear down
; 4-A. caller build up
; 4-B. recursive call
; 4-C. caller tear down
; 5. callee tear down
RET
```

Before Caller (main) buildup

Runtime stack

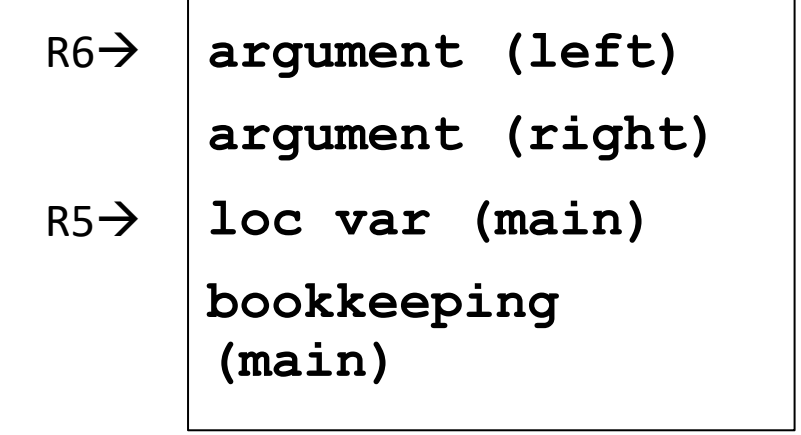


Example: Foo 2 arguments, 2 local variables

```
FOO
; 1. callee build up
; 2. base cases
; 3-A. caller build up
; 3-B. recursive call
; 3-C. caller tear down
; 4-A. caller build up
; 4-B. recursive call
; 4-C. caller tear down
; 5. callee tear down
RET
```

Just after entering FOO

Runtime stack

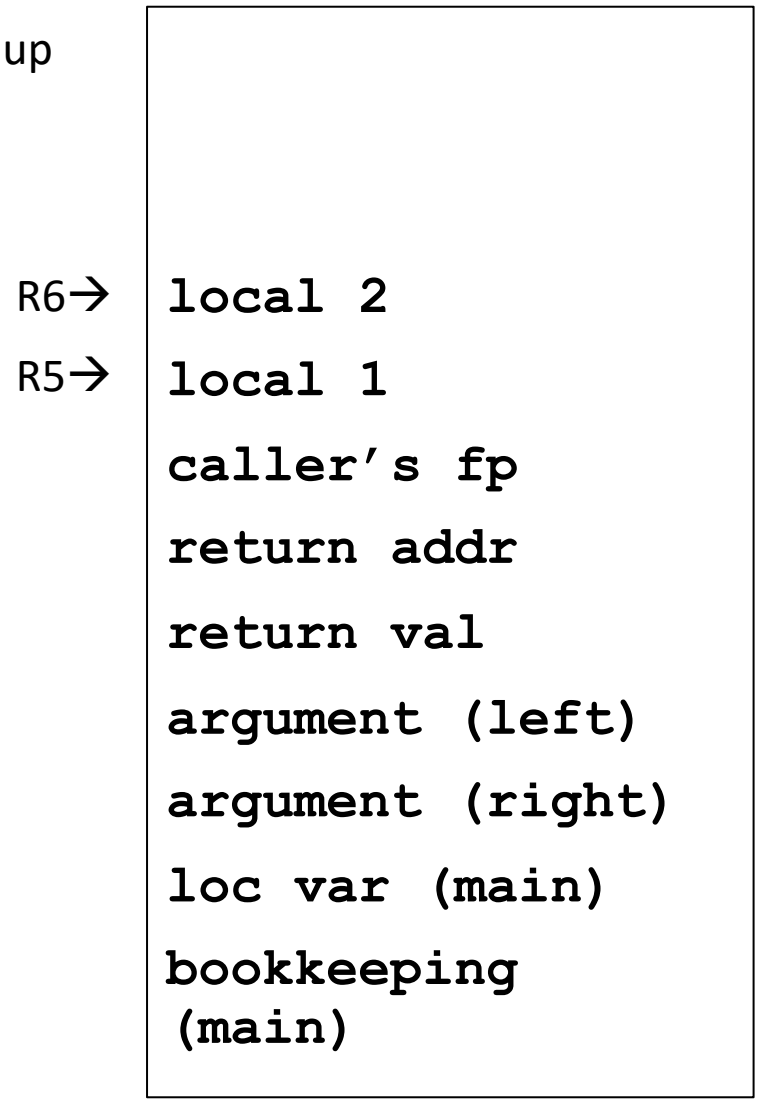


Example: 2 arguments, 2 local variables

```
FOO
; 1. callee build up
; 2. base cases
; 3-A. caller build up
; 3-B. recursive call
; 3-C. caller tear down
; 4-A. caller build up
; 4-B. recursive call
; 4-C. caller tear down
; 5. callee tear down
RET
```

Just after callee build up

Runtime stack

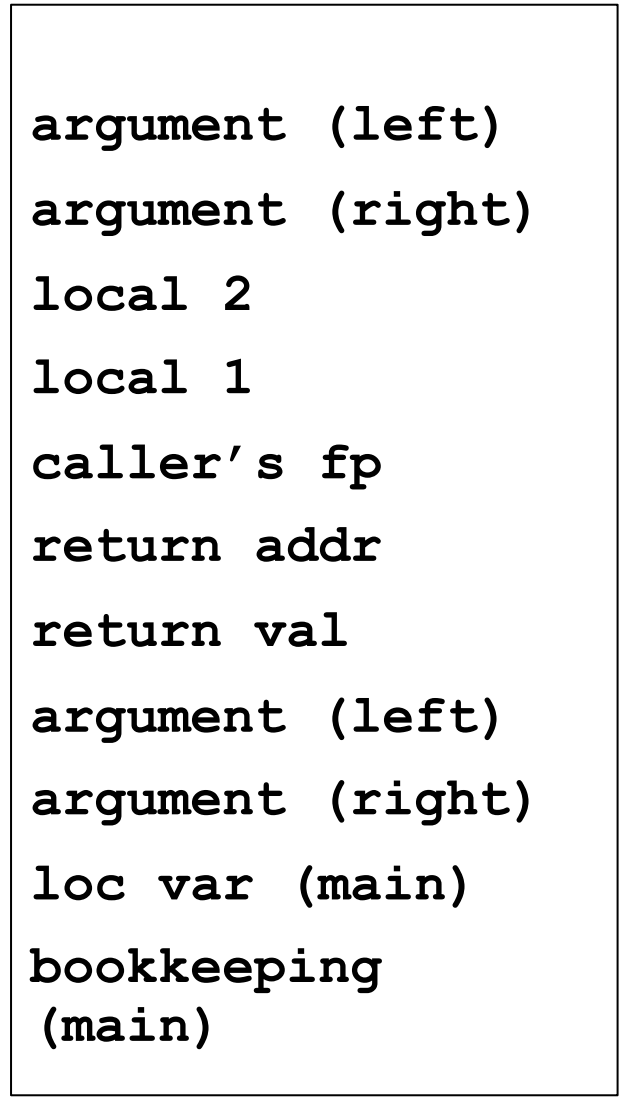


Example: 2 arguments, 2 local variables

```
FOO
; 1. callee build up
; 2. base cases
; 3-A. caller build up
; 3-B. recursive call
; 3-C. caller tear down
; 4-A. caller build up
; 4-B. recursive call
; 4-C. caller tear down
; 5. callee tear down
RET
```

Just after caller build up

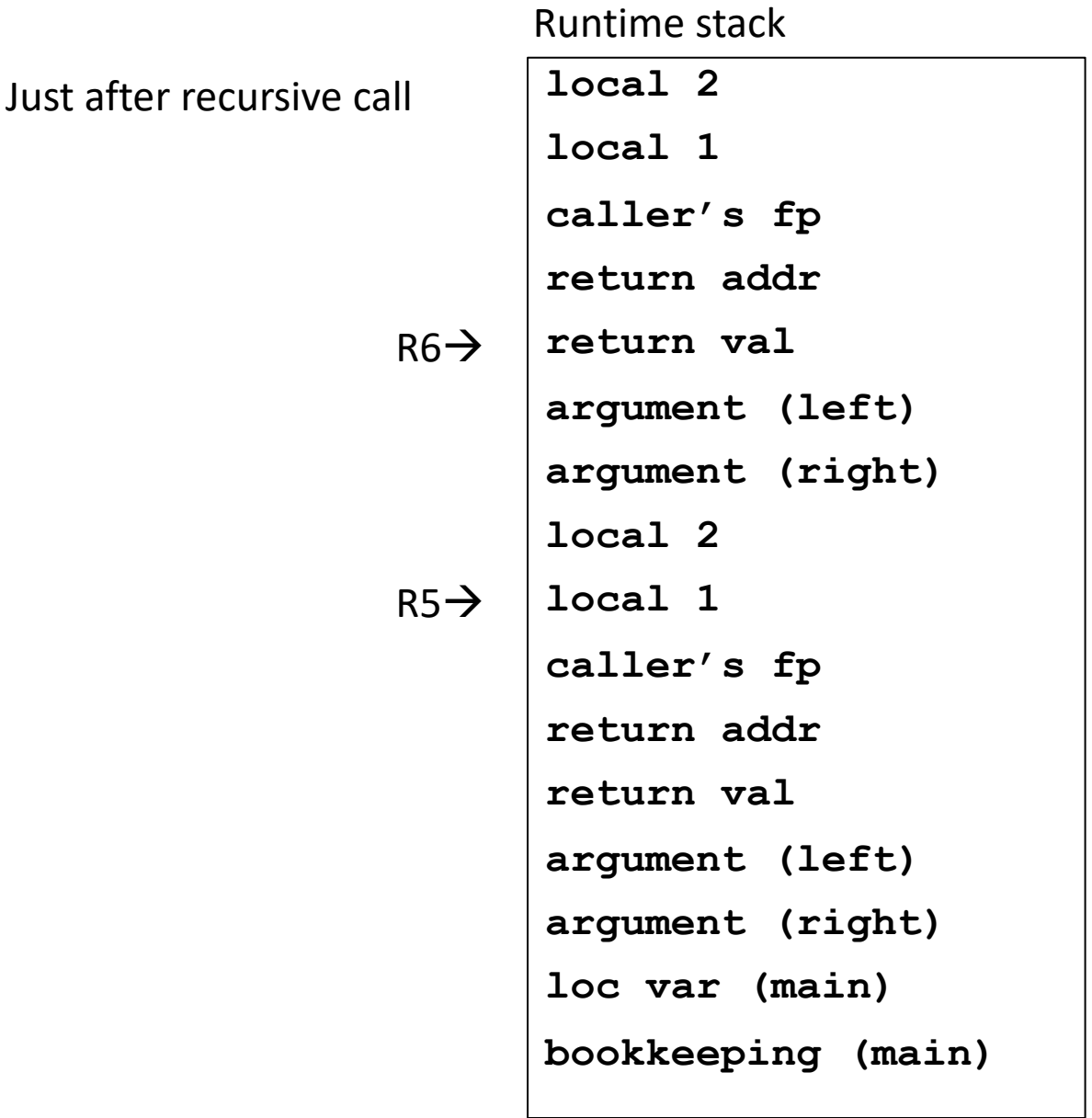
Runtime stack



Example: 2 arguments, 2 local variables

```
FOO
; 1. callee build up
; 2. base cases
; 3-A. caller build up
; 3-B. recursive call
; 3-C. caller tear down
; 4-A. caller build up
; 4-B. recursive call
; 4-C. caller tear down
; 5. callee tear down
RET
```

Just after recursive call



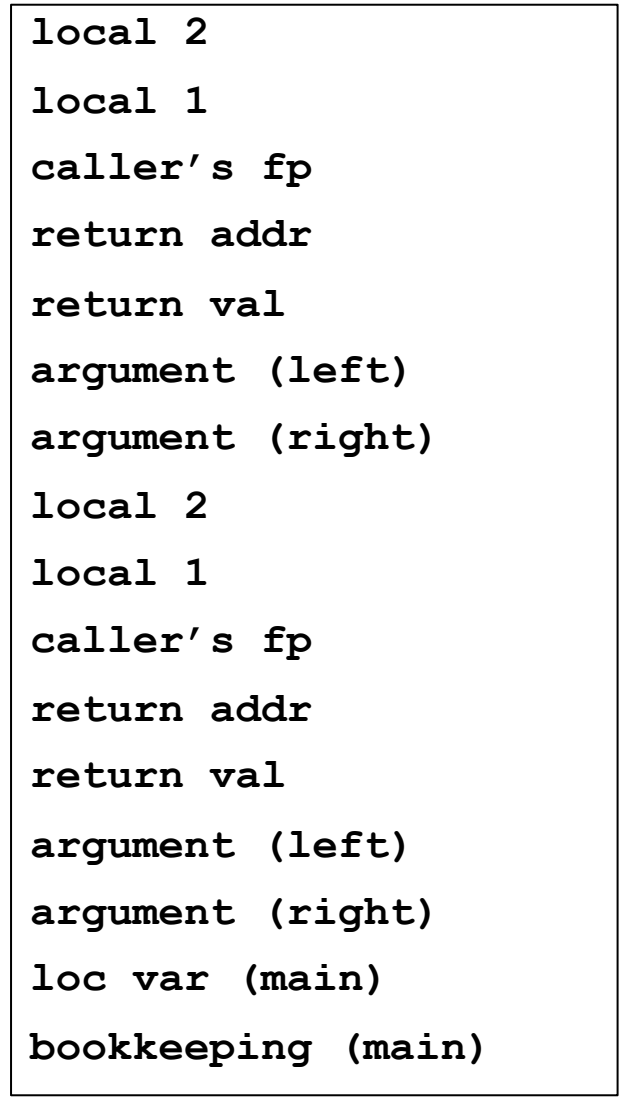
runtime stack should match with the one after callee build up

Example: 2 arguments, 2 local variables

```
FOO
; 1. callee build up
; 2. base cases
; 3-A. caller build up
; 3-B. recursive call
; 3-C. caller tear down
; 4-A. caller build up
; 4-B. recursive call
; 4-C. caller tear down
; 5. callee tear down
RET
```

Just after caller tear down

Runtime stack

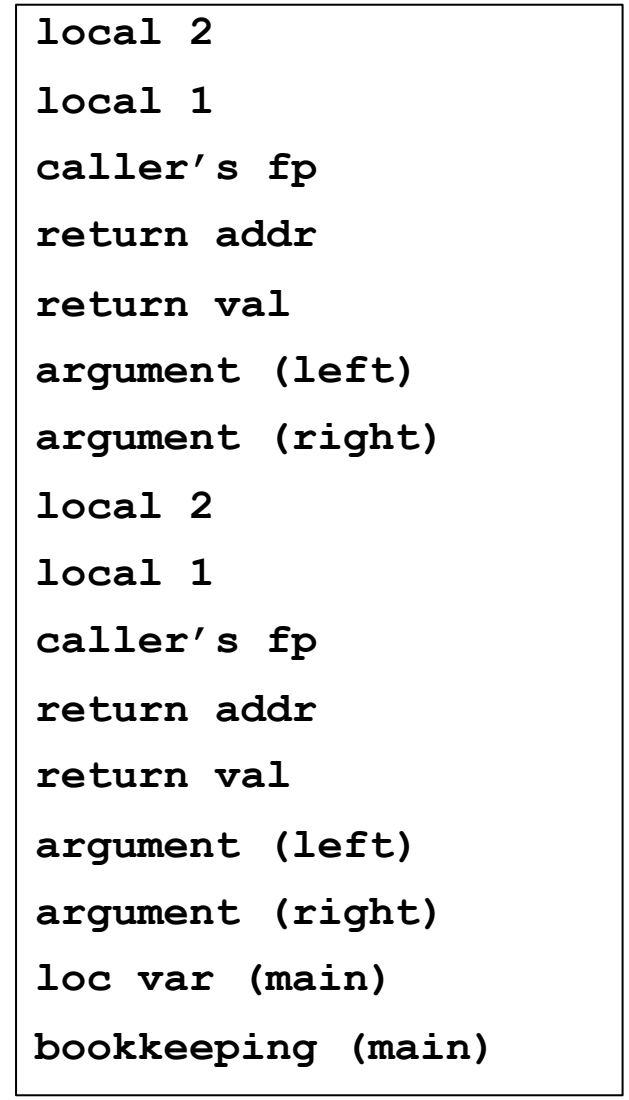


Example: 2 arguments, 2 local variables

```
FOO
; 1. callee build up
; 2. base cases
; 3-A. caller build up
; 3-B. recursive call
; 3-C. caller tear down
; 4-A. caller build up
; 4-B. recursive call
; 4-C. caller tear down
; 5. callee tear down
RET
```

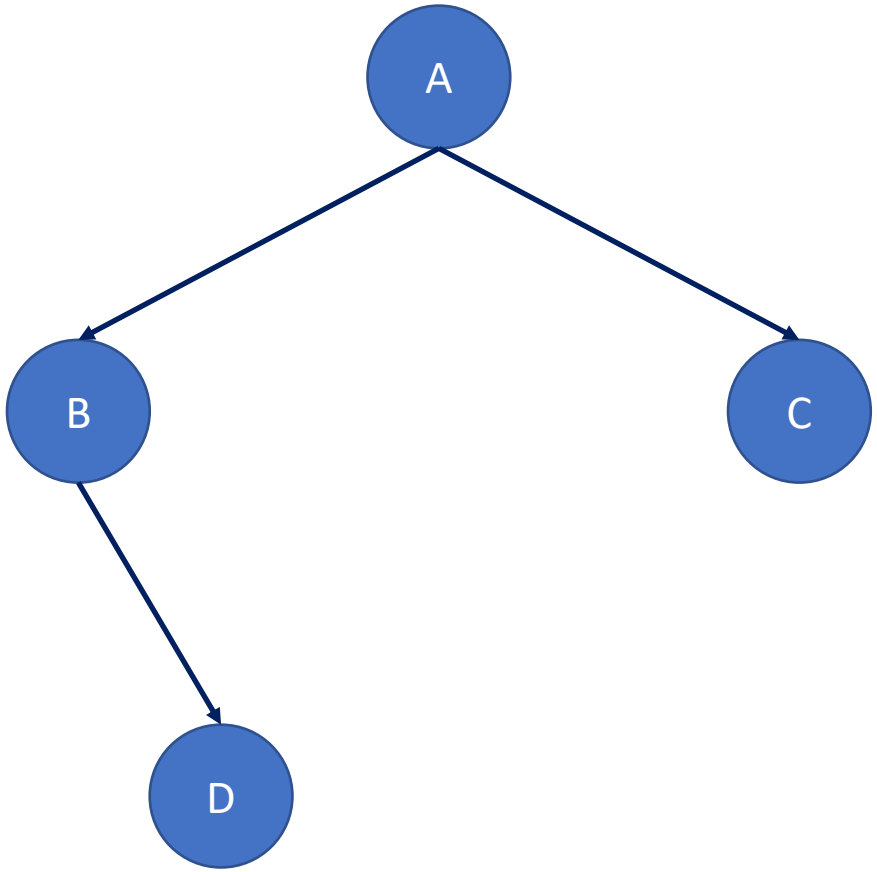
Just after callee tear down

Runtime stack



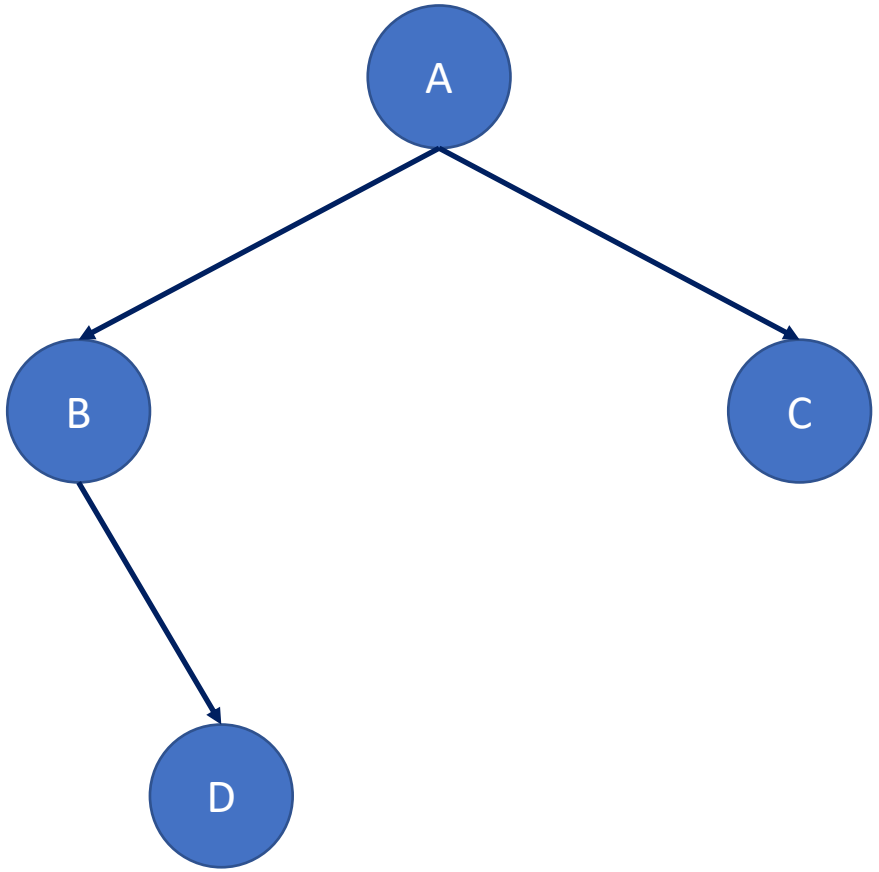
C to LC-3 – Binary Tree Traverse

```
typedef struct nodeTag t_node;
struct nodeTag{
    int data;
    t_node *left;
    t_node *right;
};



void inorder(t_node *node){
    if(node == NULL)
        return;
    else{
        inorder(node->left);
        printf("%d ", node->data);
        inorder(node->right);
    }
}
```

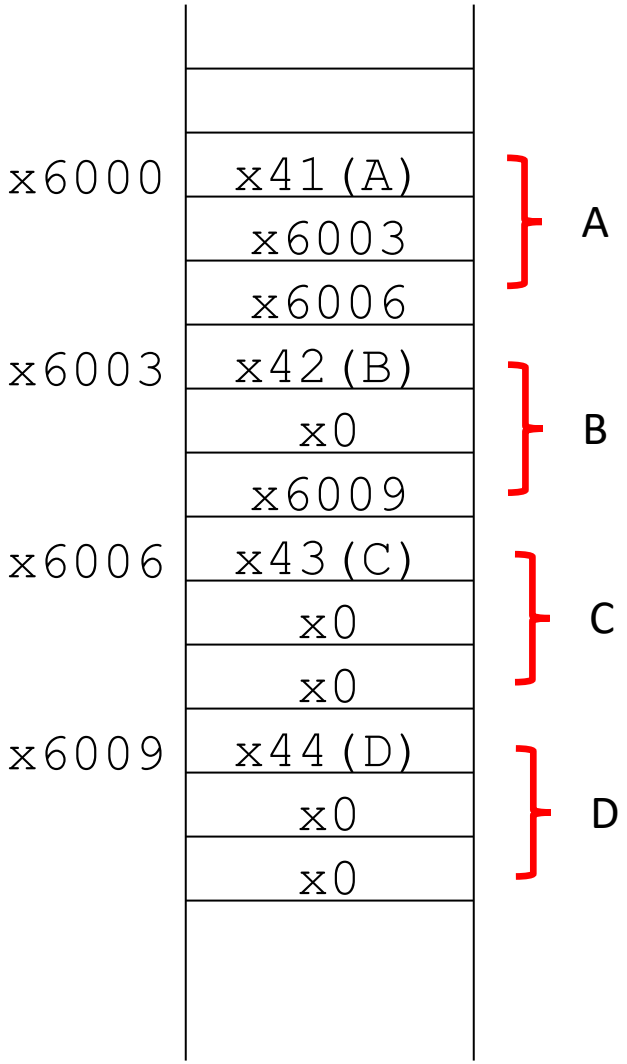


Memory Allocation (Example)

```
typedef struct nodeTag t_node;
struct nodeTag{
    int data;
    t_node *left;
    t_node *right;
};
```



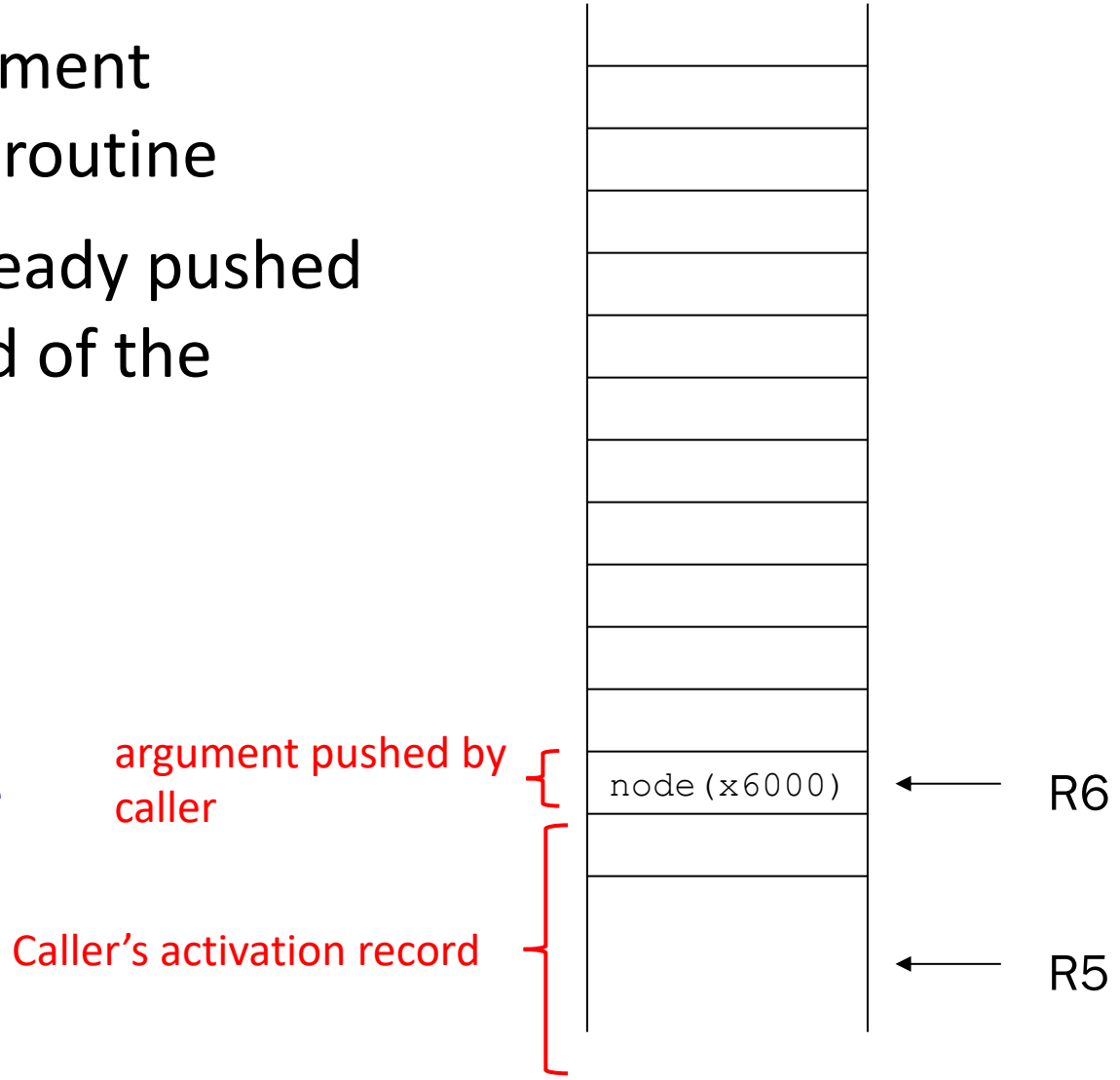
inorder → BDAC



Called Function – Push Arguments

- Run-time stack at the moment of entry in `inorder` subroutine
- Arguments, if any, are already pushed onto the activation record of the called function

```
void inorder(t_node *node)

int main(){
   t_node *root = NULL;
   // tree constructed here
   ...
   inorder(root);
```



C to LC3 Implement

- Part1 – Callee build up

```
INORDER
;;Part 1 - push book keeping info
```

- Part2 – Implement function logic

```
;;Part2
;if (node == NULL) skip to the end (Done)
;else
;inorder(node->left);
;printf("%d", node->data);
;inorder(node->right);
```

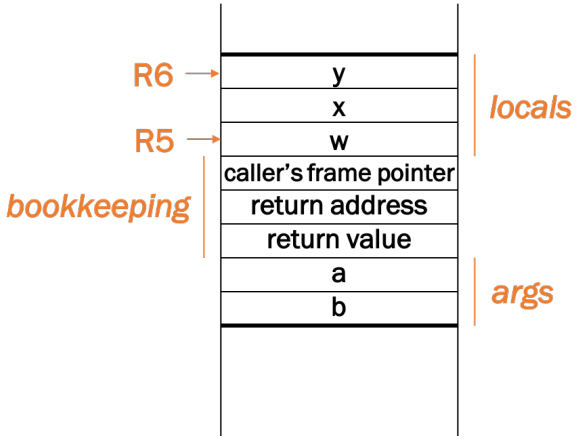
- Part3 – Callee tear down (prepare to return)

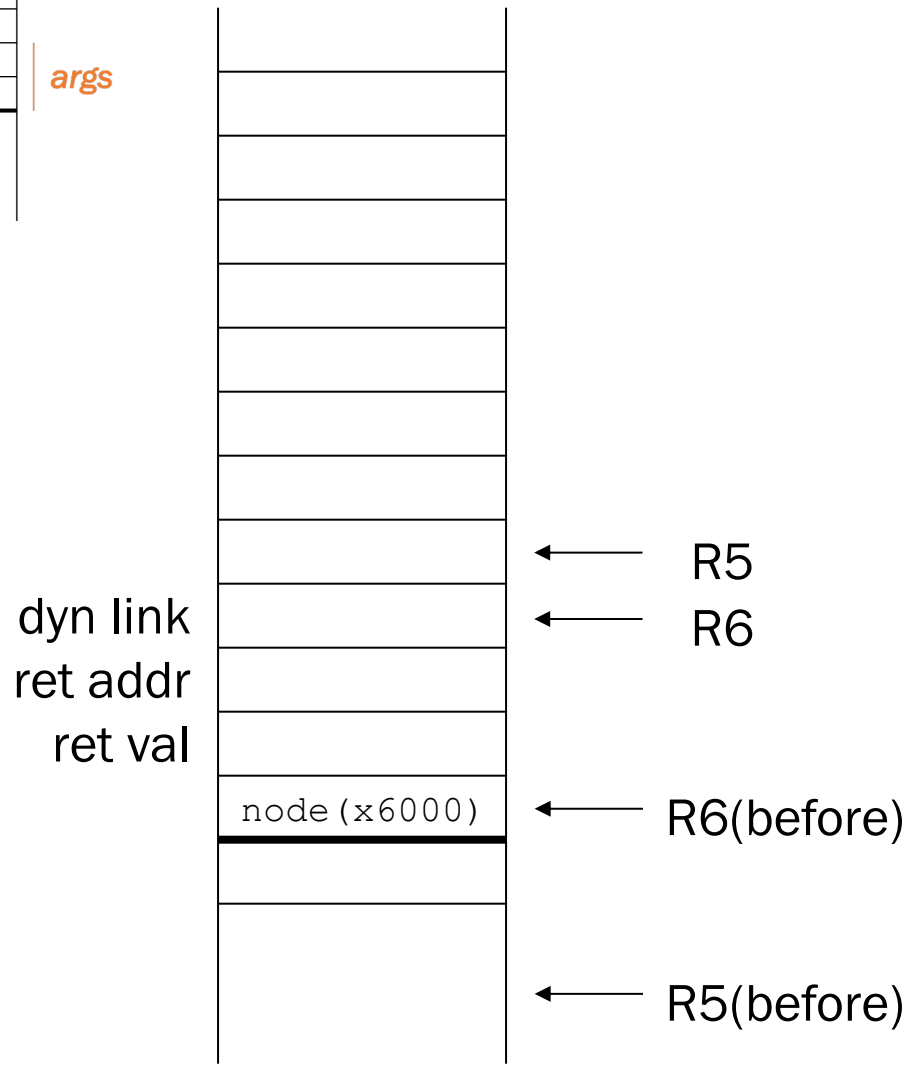
```
;;Part3
RET
```

Implement

- Part1 – Callee build up

```
INORDER
;;Part 1 – push book keeping info
;allocate space for return value

;Push return address to stack

;Store old frame pointer

;Set up new frame pointer
```

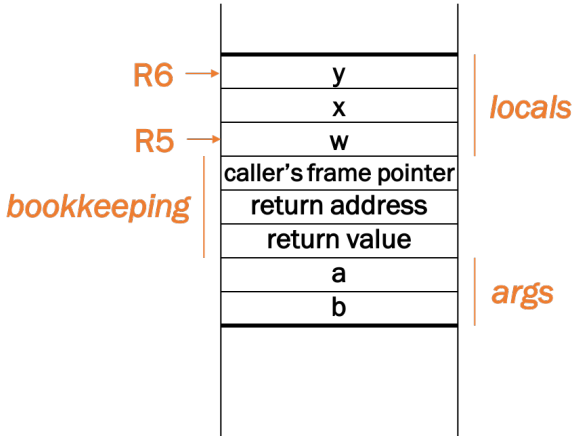


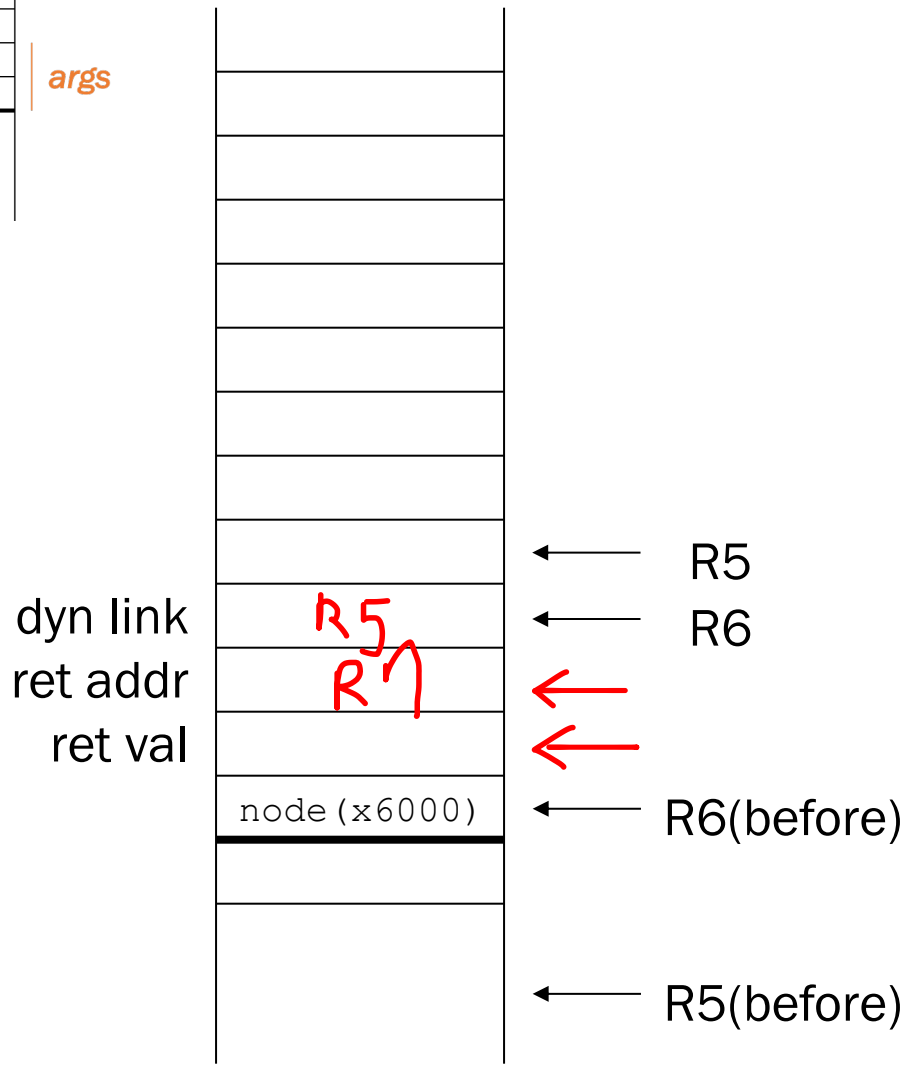


Implement

- Part1 – Callee build up

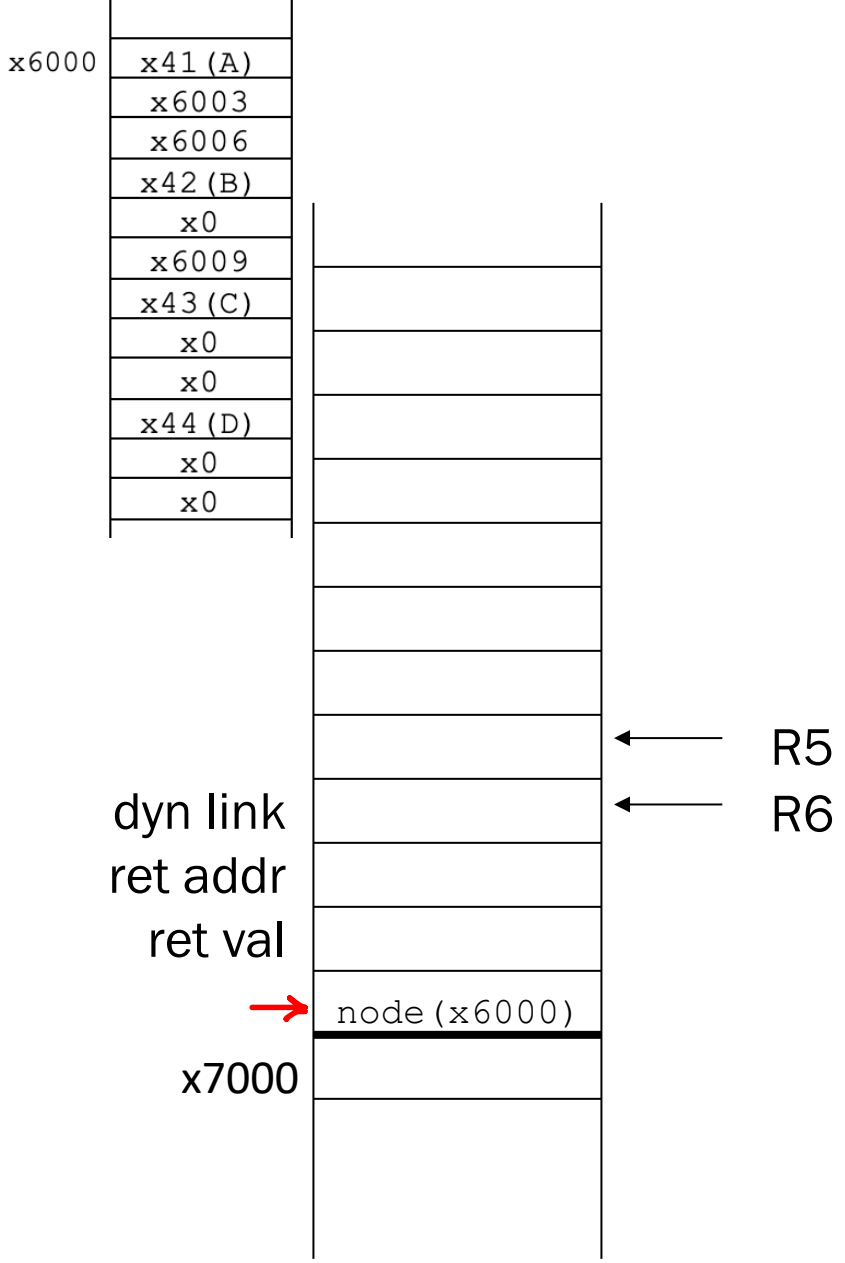
```
INORDER
;;Part 1 – push book keeping info
;allocate space for return value
ADD R6, R6, #-1
;Push return address to stack
ADD R6, R6, #-1
STR R7, R6, #0
;Store old frame pointer
ADD R6, R6, #-1
STR R5, R6, #0
;Set up new frame pointer
ADD R5, R6, #-1
```





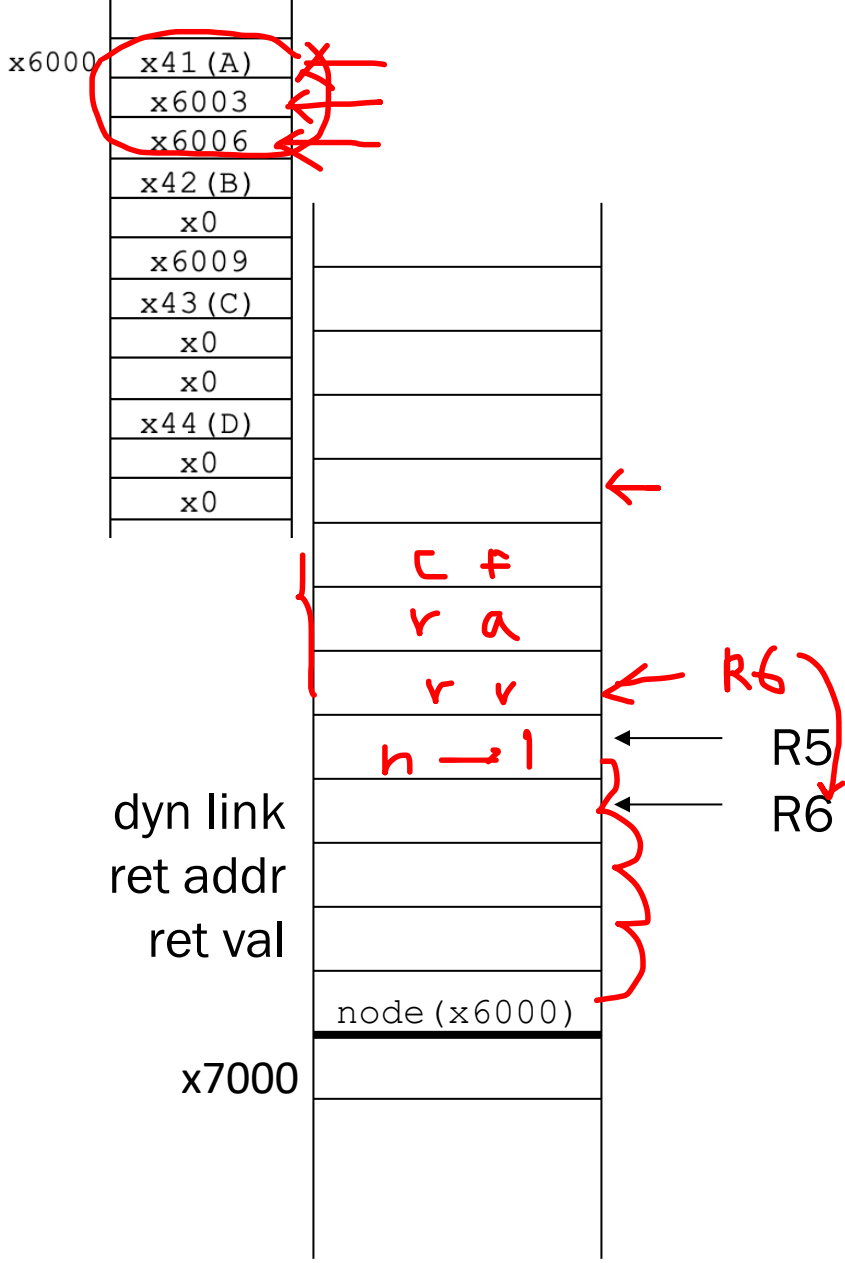
- Part2 – Implement function logic

```
;;Part2
;if (node == NULL) skip to the end (Done)
                                ;load node to R1
BRz DONE
;else
;inorder(node->left);
                                ;load node->left to R2

                                ;push node->left to stack
JSR INORDER
                                ;stack tear down
;printf("%d", node->data);
                                ;reload node

;inorder(node->right);
                                ;load node->right to R2

                                ;push node->right to stack
JSR INORDER
                                ;stack tear down
```



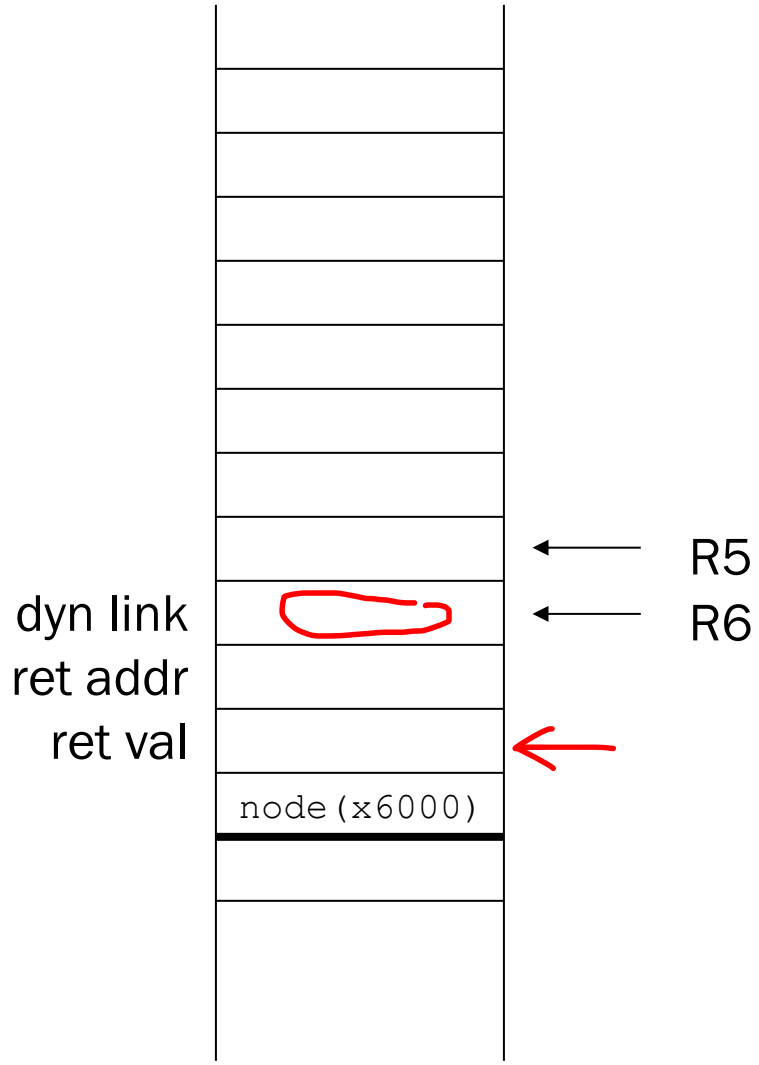
- Part2 – Implement function logic

```
;;Part2
;if (node == NULL) skip to the end (Done)
LDR R1, R5, #4              ;load node to R1
BRz DONE
;else
;inorder(node->left);
LDR R2, R1, #1              ;load node->left to R2
ADD R6, R6, #-1
STR R2, R6, #0              ;push node->left to stack
JSR INORDER
ADD R6, R6, #2              ;stack tear down
;printf("%d", node->data);
LDR R1, R5, #4              ;reload node
LDR R0, R1, #0
OUT
;inorder(node->right);
LDR R2, R1, #2              ;load node->right to R2
ADD R6, R6, #-1
STR R2, R6, #0              ;push node->right to stack
JSR INORDER
ADD R6, R6, #2              ;stack tear down
```



- Part3 – Callee tear down (prepare to return)

```
;;Part3
DONE
_____________________________   ;restore old frame pointer
_____________________________
_____________________________   ;restore return address
_____________________________
RET
```



- Part3 – Callee tear down (prepare to return)

```
;;Part3
DONE
LDR R5, R6, #0          ;restore old frame pointer
ADD R6, R6, #1
LDR R7, R6, #0          ;restore return address
ADD R6, R6, #1
RET
```