STACK

STACK

FROM Adams Drozdek
and
Allen Weiss books

STACK

Example: A stack of plates

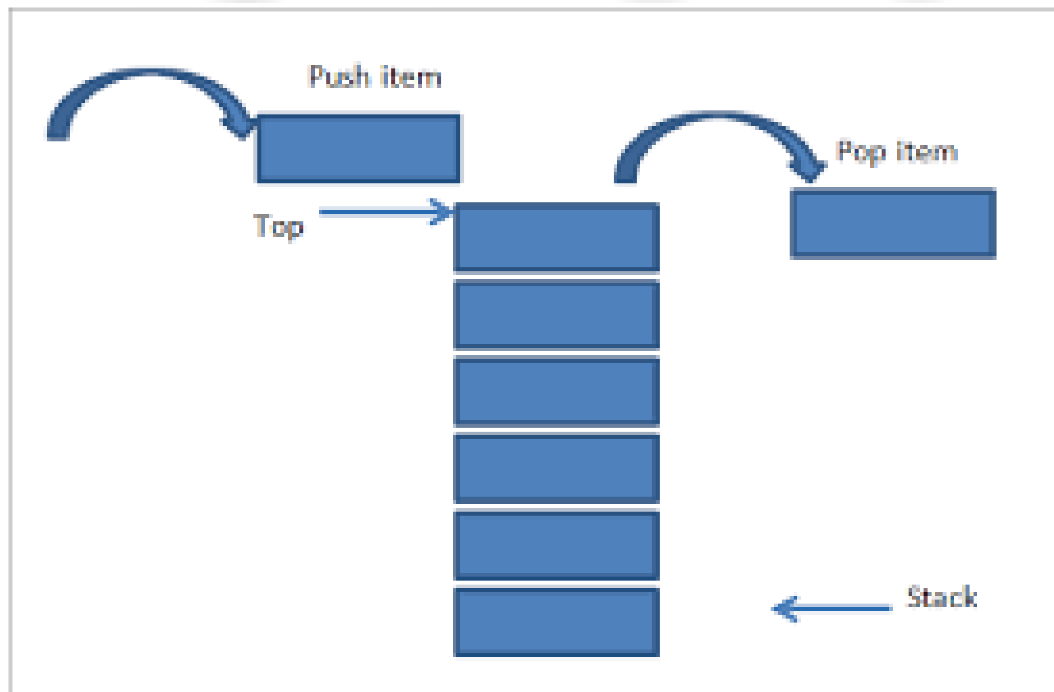
- New plates are put on the top of the stack and
- We take plates from the top.

A stack is ***LIFO structure:***
last in/first out.

Stack

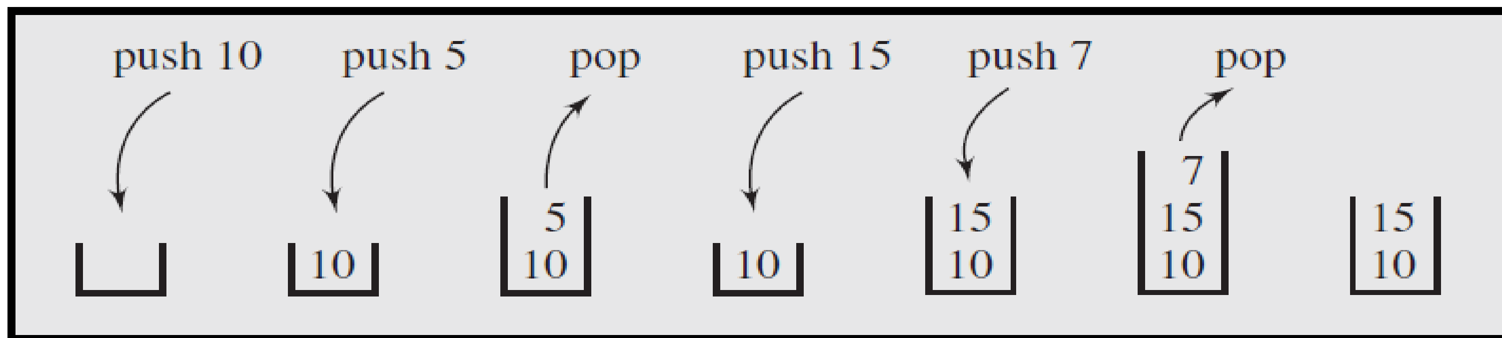
- A *stack* is a linear data structure
- It is accessed **only at one of its ends** for storing and retrieving data.

Unlike queue, in stack both ends are not used



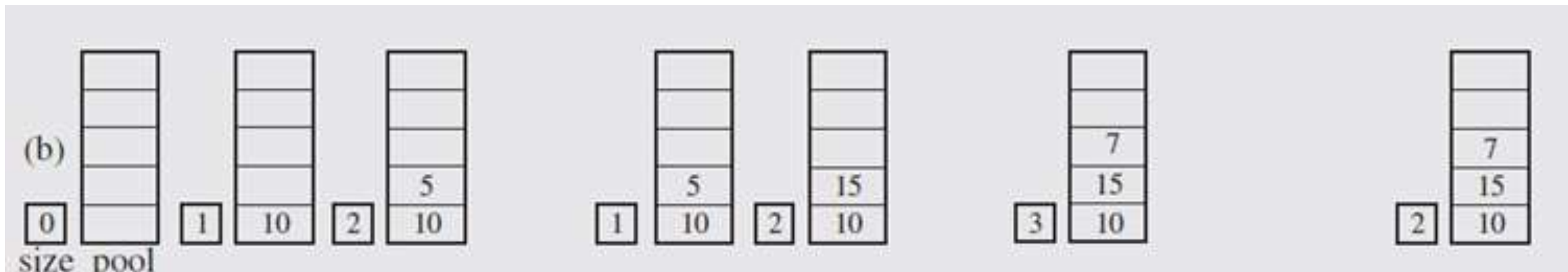
Stack Opeations

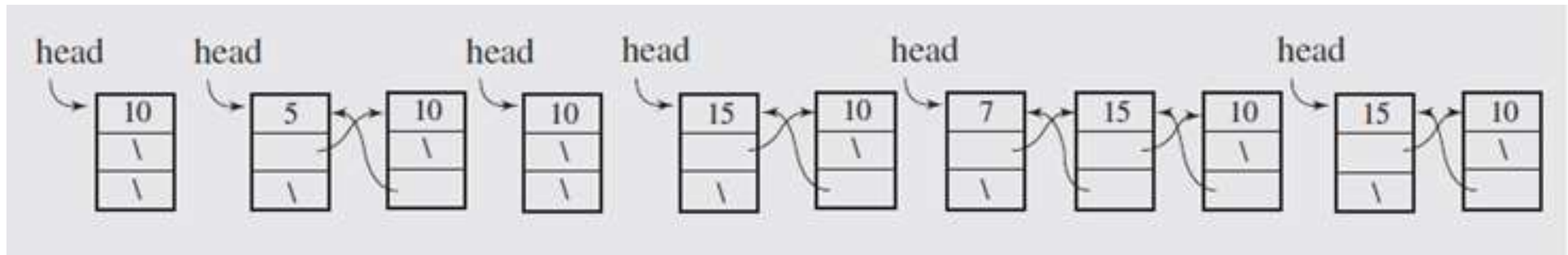
Operation	Description
clear()	• Clear the stack.
isEmpty()	• Check to see if the stack is empty.
push(x)	• Put the element *x* on the top of the stack.
pop()	• Take the topmost element from the stack.
topItem()	• Return the topmost element without removing it.



Implmentation of Stack

- Which data structure should we use to implement stack?
  - **Arrays**
  - **Linked List (SLL, DLL, CLL)**





STACK OPS

If we restrict the operations allowed on a list, those operations can be performed very quickly.

The big surprise, however, is that the small number of operations left are so powerful and important.

Applications of Stack

The stack is useful in situations when data must be stored and then retrieved in reverse order.

- Go Back and Forward in a Browser
- Undo-Redo in a Text Editor
- Adding Large Numbers
- Matching delimiters in a program.
- Evaluation of Fully Parenthesized Expression
- System Stack
- Converting Infix notation to PostFix

Application 2 – UNDO and REDO

UNDO

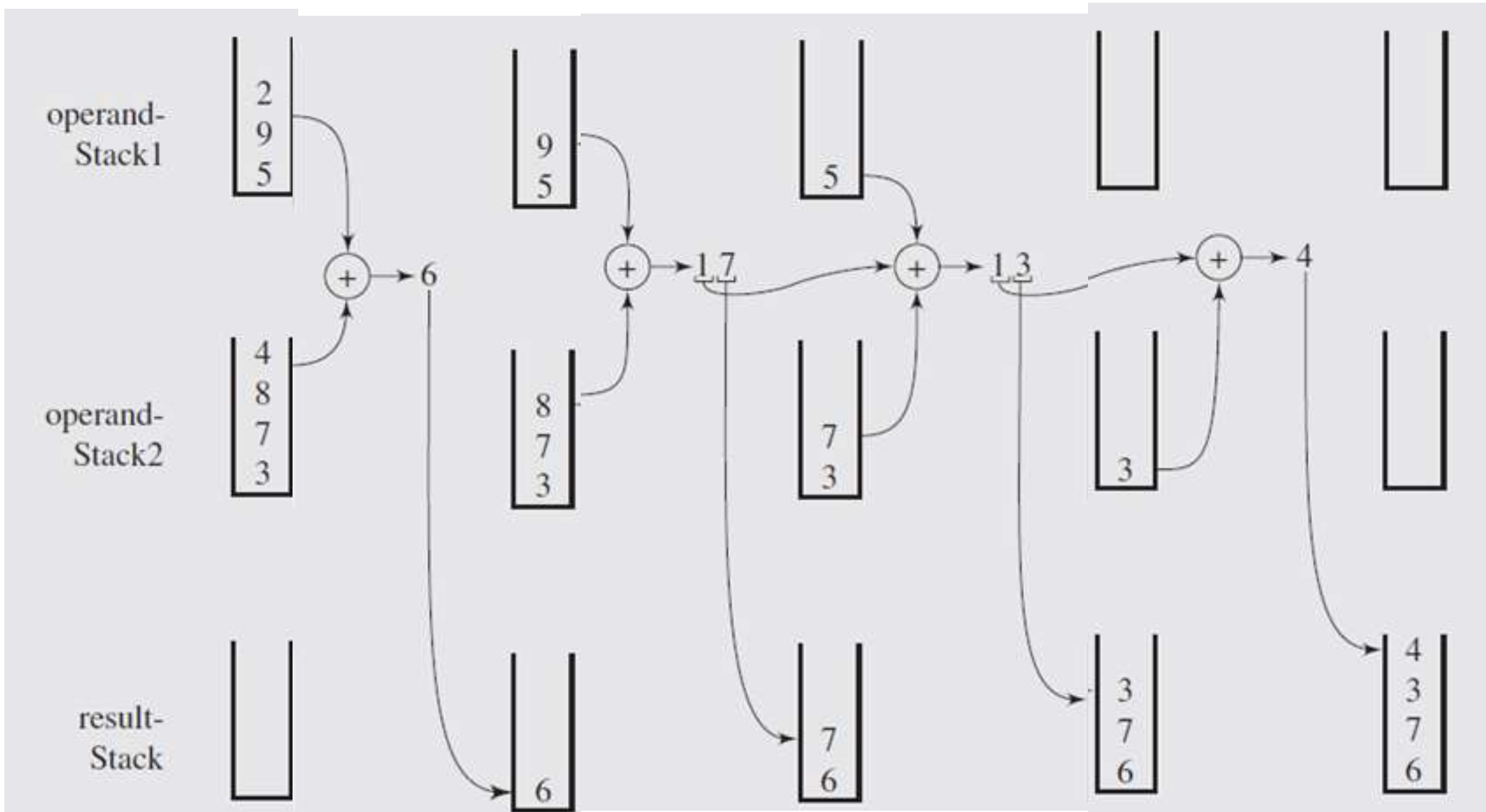
Add elements (operations performed) on the stack and to Undo POP

REDO

- Put the popped items from UNDO stack to another stack REDO stack
- When asked to REDO: simply pop from REDO stack and push on the UNDO stack

Application 3 – Adding Large Numbers

$$\begin{array}{r} 592 \\ +\,3784 \\ \hline 4376 \end{array} \qquad \begin{array}{r} 2 \\ +\,4 \\ \hline 6 \end{array} \qquad \begin{array}{r} 9 \\ +\,8 \\ \hline 17 \end{array} \qquad \begin{array}{r} 1 \\ 5 \\ +\,7 \\ \hline 13 \end{array} \qquad \begin{array}{r} 1 \\ +\,3 \\ \hline 4 \end{array}$$

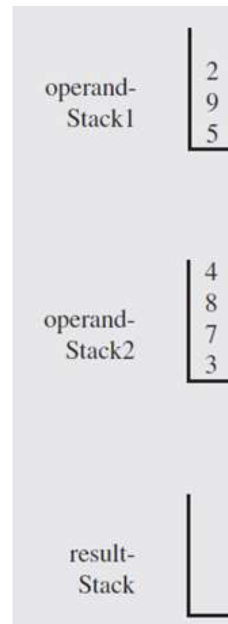


Application 3 – Adding Large Numbers

- **AddingLargeNumbers()**
  - Store First input number on Stack1
  - Store Second input number on Stack2
  - Carry =0;
  - while (!Stack1.isEmpty() || !Stack2.isEmpty()){

    //Pop a number from each non-empty stacks and add them to Carry

    - if(!Stack1.isEmpty()) n1=Stack1.pop()
    - if(!Stack2.isEmpty() n2=Stack2.pop()
    - res = n1+n2+ Carry
    - Push unit_part of res on ***ResultStack***
    - Set Carry equal to tens_part of res
  - }
  - If (carry !=0) ResultStack.push(Carry)
  - while((! ResultStack.isEmpty())
    - Print ResultStack.pop



Application 4 - Matching delimiters

- Matching delimiters in a program.
  - Delimiter matching is part of compiler: No program is considered correct if the delimiters are mismatched.
  - In C++ programs, delimiters are
    - parentheses "(" and ")",
    - square brackets "[" and "]",
    - curly brackets "{" and "}", and
    - comment delimiters "/*" and " */".

Find in which statement are delimiters not properly matched

```
a = b + (c - d) * (e - f);
g[10] = h[i[9]] + (j + k) * l;
a = b + (c - d) * (e - f));
g[10] = h[i[9]] + j + k) * l;
while (m < (n[8] + o)) { p = 7; /* initialize p */ r = 6; }
while (m < (n[8) + o]) { p = 7; /* initialize p */ r = 6; }
```

Matching delimiters

- How to match delimiters in **while (m < (n[8] + o)) ?**

Processing string with Delimiter Matching Algorithm using Stack

```
S = t[5] + u/(v* (w+y));
```

Application 3 - Matching delimiters

- The basic idea of delimiter matching algorithm
  - Input a character
  - If input character is an opening delimiter store it on a Stack
  - else if it is a closing delimiter,
    - Then compare it to a delimiter popped off the stack.
    - If they match, processing continues; if not, processing discontinues by signaling an error

Matching delimiters

- delimiterMatching(file)
  - ch = file.read()
  - while (!file.eof())
    - if ch is '(', '[', or '{'
      - ?
    - elseif ch is ')', ']', '}'
      - ?
    - elseif ch is '/'
      - ?
    - else other characters
      - ?
    - Ch =file.read()
  - **How do you know if delimiters are matched or not ?**

Matching delimiters

- delimiterMatching(file)
  - ch = file.read()
  - while (!file.eof()){
    - if ch is '(', '[', or '{'
      - Stack.Push(ch);
    - elseif ch is ')', ']', '}'
      - if (ch != Stack.pop()) report failure
    - elseif ch is '/'
      - ch2 = file.read()
      - if ch2 =='*'
        - » Skip all characters until '*/' is found. Report failure if eof reached before this is found
      - else ch= ch2 continue
    - else other characters
      - ignore
    - Ch =file.read()
  - If (Stack.isempty()) success
  - Else failure

Application 5:
Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	**Stack**	**Remarks**
(	(	Push

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	**Stack**	**Remarks**
(	(	Push
a	(a	push

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	**Stack**	**Remarks**
(	(	Push
a	(a	push
+	(a+	push

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	**Stack**	**Remarks**
(	(	Push
a	(a	push
+	(a+	push
(	(a+(	push

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	Stack	Remarks
(	(	Push
a	(a	push
+	(a+	push
(	(a+(	push
b	(a+(b	push

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	**Stack**	**Remarks**
(	(	Push
a	(a	push
+	(a+	push
(	(a+(	push
b	(a+(b	push
/	(a+(b/	push

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	Stack	Remarks
(	(	Push
a	(a	push
+	(a+	push
(	(a+(	push
b	(a+(b	push
/	(a+(b/	push
c	(a+(b/c	Push

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	**Stack**	**Remarks**
(	(	Push
a	(a	push
+	(a+	push
(	(a+(	push
b	(a+(b	push
/	(a+(b/	push
c	(a+(b/c	Push
)	(a+2	Pop"(b/c" and evaluate and push the result back

Application 5: Evaluation of Fully Parenthesized Expression

(a+(b/c)) assuming a=2, b=6, c=3

Input Symbol	Stack	Remarks
(	(	Push
a	(a	push
+	(a+	push
(	(a+(	push
b	(a+(b	push
/	(a+(b/	push
c	(a+(b/c	Push
)	(a+2	Pop"(b/c" and evaluate and push the result back
)	4	Pop"(a+2" and evaluate and push the result back

APPLICATION 6 - SYSTEM STACK

Function Calls

Can a function call another function ?

Can a function call itself ?

What happens to the local variables of the calling function?

- when a call is made to a new function, all the variables local to the calling routine need to be saved by the system

How to save the local variables of the calling function ?

- Stack

Application 6 - System Stack

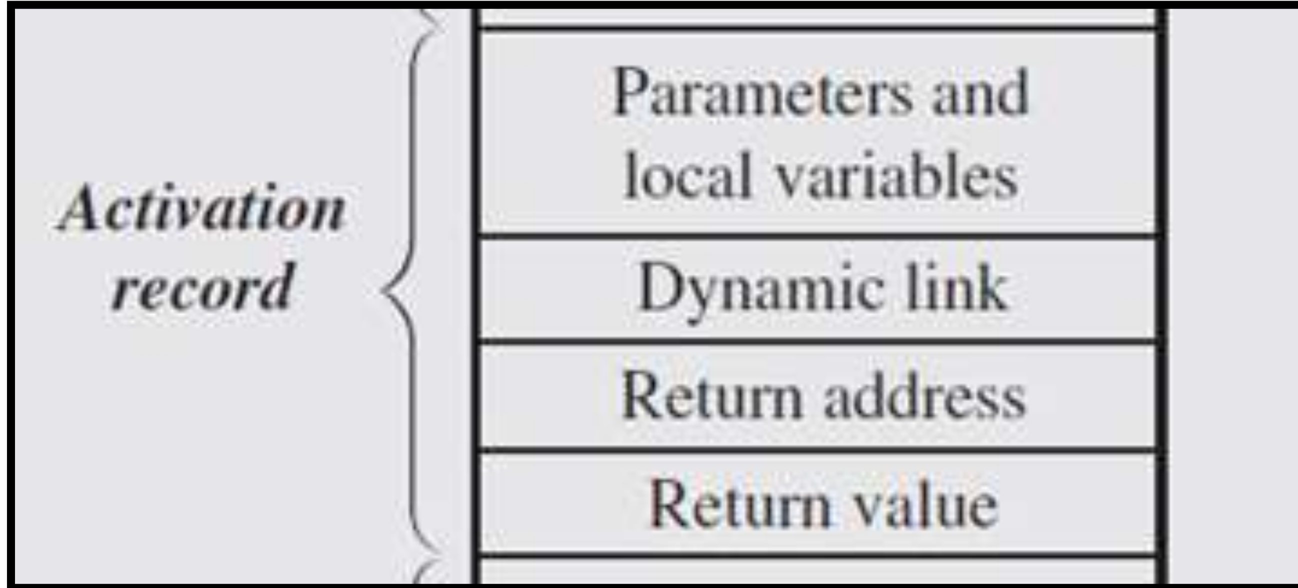
Address of the next instruction in the calling program must be saved. Why ?

- in order to resume the execution from the point of function call.

Can the function calls be nested to an arbitrary depth?

- Yes,
  - hence use of stack is a natural choice to preserve the return address.

Activation Records



Activation record is a data structure which keeps important information about a sub program.

The information stored in an activation record includes

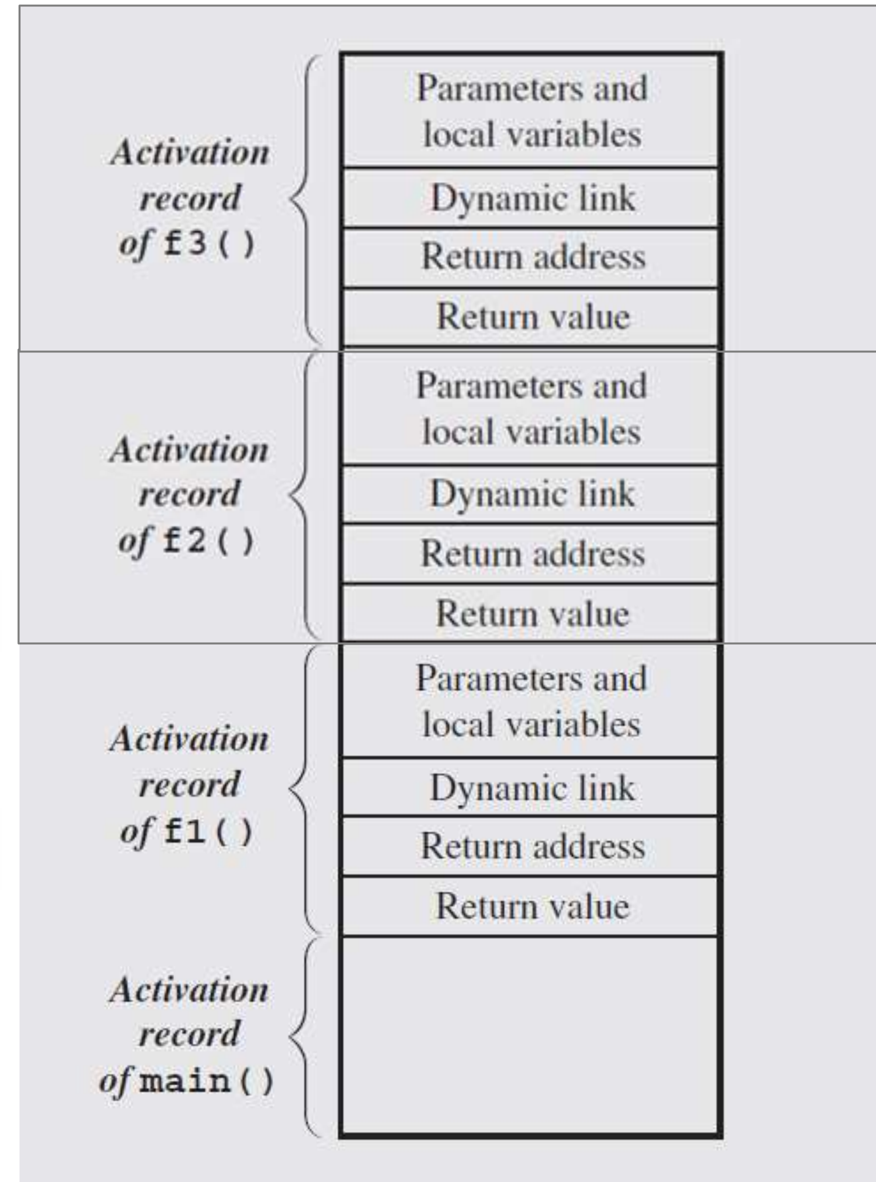
- the address of the instruction to be executed next,
- current value of all the local variables and parameters. i.e. the context of a subprogram is stored in the activation record.

Activation Records

```
main(){
   f1()
}
f1(){
   f2()
}
f2(){
   f3()

f3(){Print "Hello World"}
```

Contents of the run-time stack when



When a Function is called

- its activation record is created and
- pushed into the System stack.

When the Function ends

- its activation record is popped from the stack and destroyed-
- The control returns back to the calling function restoring its context

Activation records

```
int main()
{
      int x,y;
      statement1;
      A();
      statement2;
      statement3;
      B();
      statement4;

}
void A(){
      C();
      statement 5;
}
```

C()

A()

Main()