Lecture 17 Stacks

* Abstract stack is specified as a linearly ordered collection of data elements with push, pop, and peek operations

1. The push operation inserts an element into the collection. The linear ordering of elements are determined by the decreasing ordering of time stamps they are inserted. The first element (top element) is the latest inserted element.
2. The pop operation deletes and returns the top element
3. The peek operation returns the top element

* Stacks have the Last-In-First-Out characteristic
* A stack data structure is an implementation of abstract stack
  + Stack can be implemented by arrays or linked lists with one variable top representing the position of the top element. Push, pop, and peek operation are at the top position.
  + A stack is empty if there is no element in the stack
  + The length (height) of a stack is the number of elements in the stack
* Underflow and overflow
  + When a stack is empty, pop operation cannot be done (underflow)
  + When the height reaches the maximum height that a stack is allowed, then push cannot be done (overflow)

**Array Stacks**

* An Array based stack uses an array to store stack elements, and variable top to represent the index position of insertion and deletion.

1. An array based stack is created by creating an array of MAX (given) height, and set top = -1 (empty stack)
2. **Push:** when top < MAX-1, setp tep = top+1, set the new element at the top position
3. **Pop:** when the stack is not empty, deleting an element from the stack by setting top = top-1
4. **Peek:** when the stack is not empty, get the element at the top position

Push an element onto array stack

Input: stack[MAX], top, value

1. If top = MAX-1

Print OVERFLOW

Goto step 4

1. Top = top+1
2. Stack[top] = value
3. Stop

Time: O(1), space: O(1)

Pop an element from array stack

Input: stack[MAX], top

1. If top = -1

Print UNDERFLOW

Goto step 2

Else

Top = top-1

1. Stop

Time: O(1), space: O(1)

Peek on array stack

Input: stack [MAX]

1. If top = -1

Print UNDERFLOW

Goto step 3

1. Output stack[top]
2. Stop

Time: O(1), space: O(1)

**Linked list stacks**

* A linked list stack uses a singly linked list to represent a stack with pointer top pointing to the first element of the linked list
* Push:
  + Step 1 create a node containing the data value
  + Step 2 insert the node at the beginning of the linked list
  + Step 3 update the top pointer
* Pop:
  + Delete the first node and update top pointer
* Peek
  + Get the data ->data
* Empty
  + Top==NULL

Push onto linked list stack

Input: pointer top, value

1. Create node new\_node = new\_node(value)
2. If top==NULL

Top = new\_node

Goto step 4

1. Set new\_node->next=top, top = new\_node;
2. Output top

Time: O(1), space: O(1)

Pop an element from a linked list stack

Input: top

1. If top==NULL

Print UNDERFLOW

Goto step 5

1. Ptr = top
2. Top = top->next
3. Free ptr
4. Output top

Time: O(1), space: O(1)

Peek the top element from linked stack

Input: top

1. If top == NULL

Print UNDERFLOW

Goto step 3

1. Output top->data
2. Stop

Time: O(1), space: O(1)

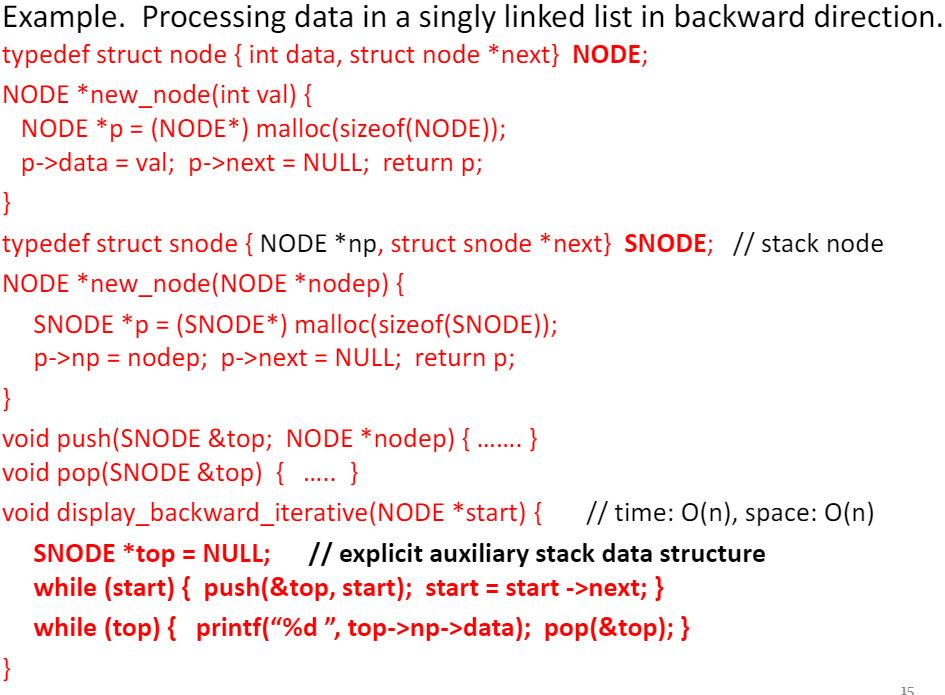
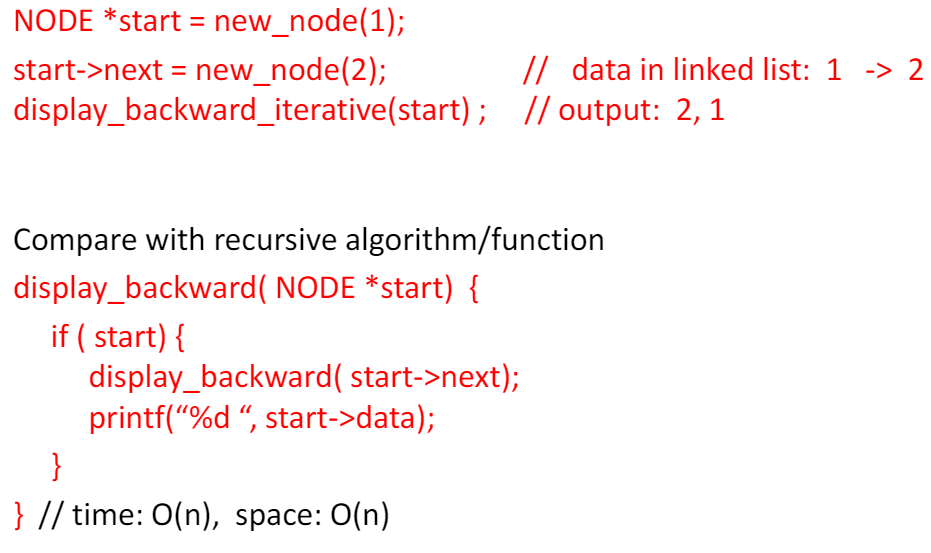
**Applications of stacks**

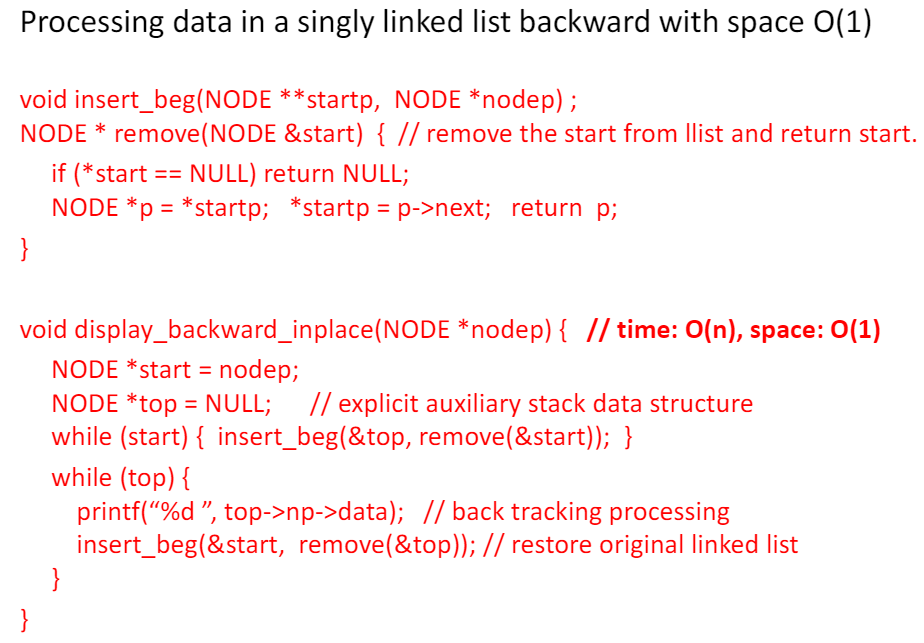
1. Back tracking

In algorithm and program designs, stacks are used to remember the path of past activities. Back tracking is to recall the past activities in the LIFO order.

Example: traversal of a linear data structure, and display visited nodes in backward order

Solution: Traverses the linear data structure, and use stack to remember visited nodes, then do back tracking by stack pops



1. Parentheses

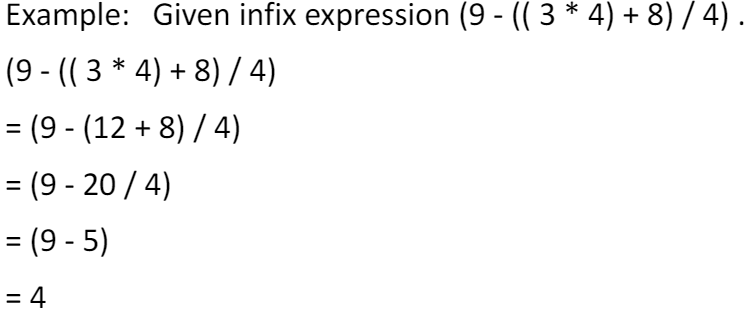
Parentheses are used in mathematical expressions to enforce precedence of operation

You can check if parentheses are valid using a stack, pushing all open brackets onto stack and popping when a close bracket is found, all brackets should match and at the end the stack should be empty.

1. Infix, postfix, prefix expressions and evaluations

* Infix notation places an operator between two operands, eg. A+B
  + Infix expression is infix notation based expression eg. (A+B)\*C
* Postfix notation puts an operator after the operands
  + The postfix notation for A+B is A B +
  + The postfix notation of (A+B)\*C is AB+C\*
* Prefix notation puts an operator before the operands
  + The prefix notation for A+B is + A B
  + The prefix notation for (A+B)\*C is \* + A B C
* Prefix and postfix notations were given by polish logician, mathematician and philosopher Jan Lukasiewicz
* The prefix notation is known as Polish notation, postfix is known as reverse polish notation or RPN
* Both of them are parentheses free, this turns out to be useful for evaluation of expressions by computers

Infix expression evaluation



How do we write a program to evaluate infix expression?

1. Convert the infix expression to postfix expression
2. Evaluate postfix expression

Postfix expression evaluation

* The postfix expression can be provided by a string or linear data structure like an array, linked list, or queue. The evaluation of a postfix expression can be done from the beginning to the end using a stack to store temporary operands.

Algorithm: evaluate a postfix expression

Input: postfix expression in queue PE

1. Create a stack S using operand’s data type
2. If PE is empty, goto step 6
3. X = dequeue(PE)
4. If X is an operand

Push (S,X)

Goto step 2

1. If X is an operator

B = pop(S), A = pop (S)

Set c = A X B

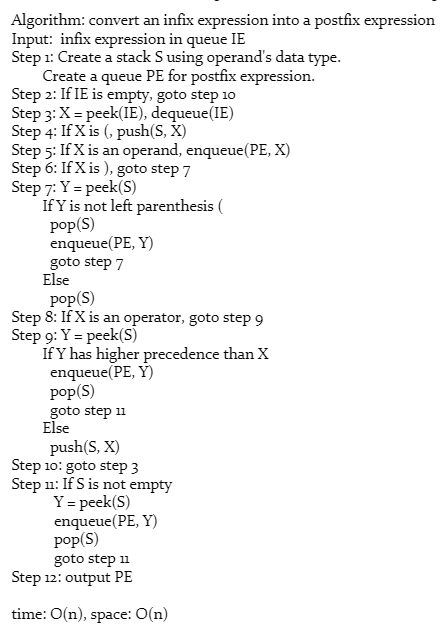
Push (S, c)

Goto step 2

1. Output the top element of S

Time: O(n), space: O(n)

Infix to postfix expression conversion





1. Recursions

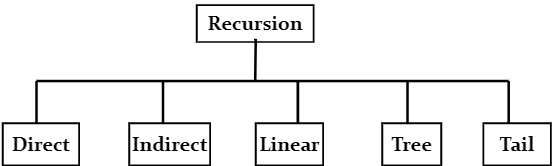
* Function execution uses system Stack to remember the calling functions from the main to the current function, so that when a function finishes it enables to track back the position of parent functions
* A recursive function is a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself

**Recursive algorithm**

* Every recursive solution/algorithm has two major cases:
  + The base case: the problem is simple enough to be solved directly without making any further calls to the same function
  + Recursive case:
    - The problem is divided into simple subparts
    - The function calls itself but with subparts of the problem obtained in the first step
    - Third, the result is obtained by combining the solutions of simpler sub-parts

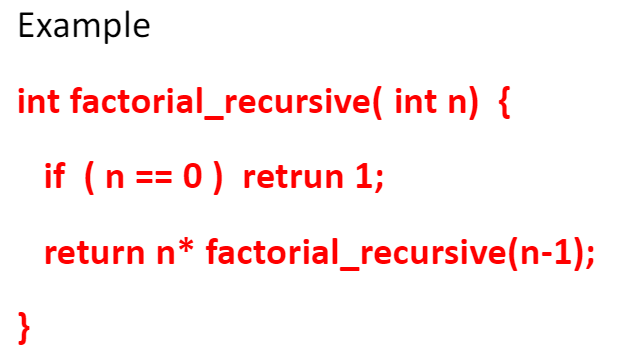
**Types of recursions**

* Any recursive function can be characterized based on:
  + Whether the function calls itself directly or indirectly (direct or indirect recursion)
  + Whether any operation is pending at each recursive call (tail-recursive or not)
* The structure of the calling pattern (linear or tree-recursive)



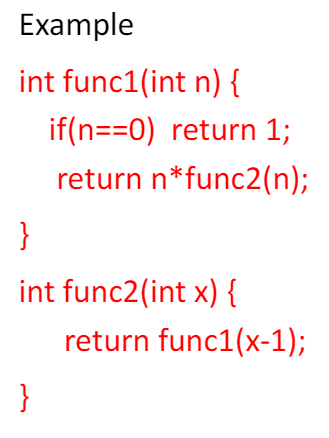
Direct recursion

* A function is directly recursive if it explicitly calls itself



Indirect recursion

* A function is indirectly recursive if it contains a call to another function which ultimately calls it

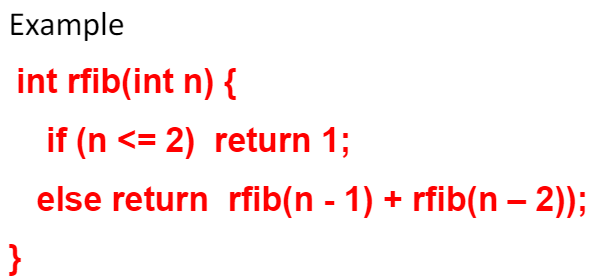


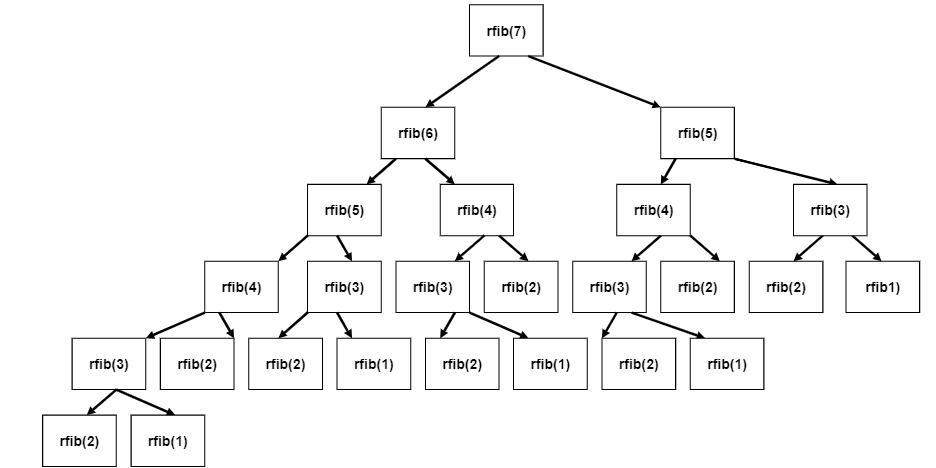
Linear recursion

* Recursive functions can also be characterized depending on the way in which the recursion grows: in a linear fashion or forming a tree structure
* A recursive function is said to be linearly recursive when pending operation involves on recursive call to the function
* For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to fact() function

Tree recursion

* A recursive function is tree recursive (or non-linearly recursive) if the pending operations makes more than one recursive call to the function





Tail recursion

* A recursive function is tail recursive if no operations are pending to be performed when the recursive function returns to its caller
* That is, when the called function returns m the returned value is immediately returned from the calling function
* Tail recursive functions are highly desirable because they are much more efficient to use as in their case the amount of information that has to be stored on the system stack is independent of the number of recursive calls

Example:

Factorial\_recursive(int n) is tail recursive

Rfib(int n) is not

Pros and Cons of recursion

Pros:

1. Recursive solutions often ten to be shorter and simpler than non-recursive ones
2. Follows a divide and conquer method to solve problems
3. Code is clearer and easier to use

Cons:

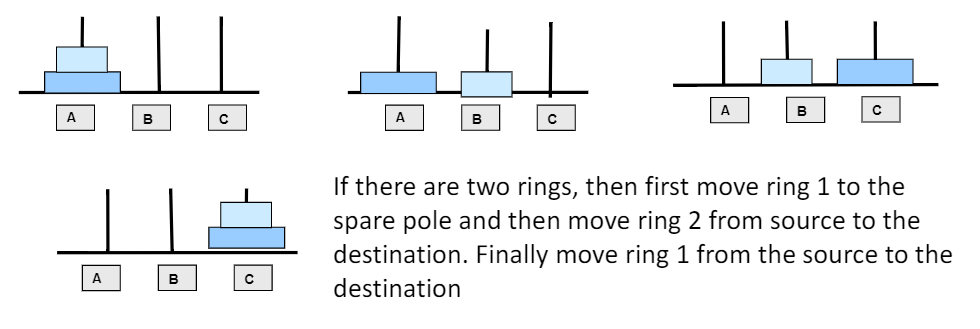
1. Recursion function is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level may run out of memory.
2. Using a recursive function takes more memory and time to execute compared to its non-recursive counterpart
3. It is difficult to debug, particularly when using global variables.

**Hanoi Tower game solver**

Tower of Hanoi is one of the main applications of recursion. It says if you can solve n-1 cases, then you can easily solve the nth case.

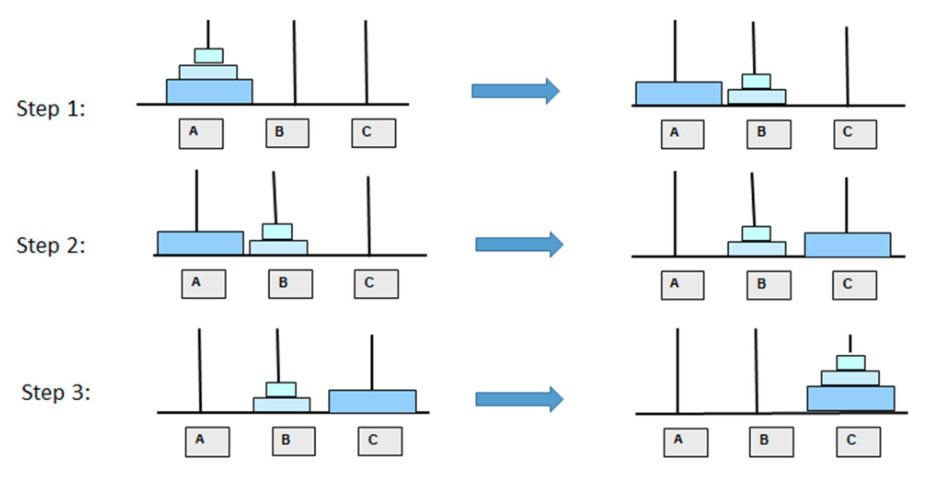


If there is only one ring, then simply move the ring from source to destination.



Divide and conquer solution

Consider the working with three rings, use divide and conquer method.



**Implement by recursive function**

Void towers(int num, char frompeg, char topeg, charauxpeg){

If(num==1){

Printf(“\n Move disk 1 from peg %c to peg %c”, frompeg, topeg);

Return;

}

Towers(num-1, frompeg, auxpeg, topeg);

Printf(“\nMove disk %d from peg %c to peg %c”, num, frompeg, topeg);

Towers(num-1, auxpeg, topeg, frompeg);

}