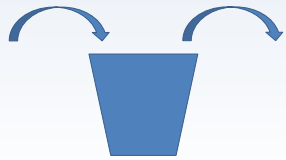
Data Structures

Stack

Stack

- The stack is the simplest of data structures.
- Placing something onto the stack and removing something from the stack is restricted to the top of the stack.
  - That is why it is also defined as Last In First Out (LIFO) storage.



- Stacks are used when nested blocks, local variables, recursive definitions, and backtracking operations are needed in programming.
- A typical programming example where stacks are used is the processing of arithmetic terms (containing parentheses and operator precedence).
- Another example could be the path-finding problem in a labyrinth that necessitates backtracking.

Stack Operations

- pushing: operation of adding a new element onto the top of the stack. The added element becomes the topmost element in the stack.

  push(...)
- popping: operation of pulling the topmost element from the stack.

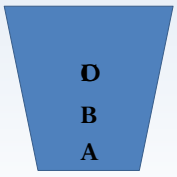
  pop()
- checking emptiness: operation of checking if the stack is empty.

  isempty()

Exercise

Perform these operations in order on a stack:

- push('A');
- push('B');
- push('C');
- pop();
- push('D');
- pop();
- pop();



Example: Arithmetic operations

- Checking for balanced parentheses in an arithmetic expression:

  7-((x* (x+y)/(y-3) +y)/4)
- Constraints:
  - There must be an equal number of right and left parentheses
  - A left parenthesis must correspond to each right parenthesis
- Solution
  1. Every time left parenthesis encountered, push onto stack.
  2. Every time right parenthesis encountered, pop from stack. Error if no popped element.
  3. When the end of the expression has been reached, error if the stack is not empty.

Example

- For checking correct usage of parentheses
- The system accepts three kinds of parentheses:
  (), {}, []
- On each line of input, for each type of left parenthesis, we will test the presence of a right parenthesis of the same kind and their appearance in the right order.

Valid	Invalid
• ( ) • { }[ ] • ({ [ ] [ ] }) • [{ { { } ( ) } [ ] } [ ]] • ( ) ( ) ( )	•) • [• {] • ( )) (( )

Solution

For every c character in the input line
  if c is a left_symbol → push onto stack.
  if c is a right_symbol →
    if stack empty → error "left symbol missing"
    else, pop an s symbol from the stack
      if s and c not compatible → error
        "ordering not appropriate"
If stack not emptied → error "missing right symbol"

Example: Evaluation of Arithmetic Expressions

- X ← ((A/(B**C)) + (D*E)) - (A*C)
- operands: A, B, C, D, E
- operators: / , ** , + , - , *
- Each operation has a certain precedence. The use of parentheses affects the result. Using parentheses, different results could be obtained
- While generating code, compilers first convert the given expression to a representation called **postfix**. This "postfix" expression is then processed using the stack structure. In this representation, the operands are written before the operators (A B +). When the operation to be processed (+) is popped from the stack, the number of operands this operation requires is popped from the stack (+ operator requires 2 operands).

Example: Call Stack

- Operating systems push the current state of the code and data into a stack before branching out into each function.
- Thus, when it returns to the branching point, it has returned to the state at the time of branching.

Stack Data Structure

```
#ifndef STACK_H
#define STACK_H
#define STACKSIZE 5
typedef char StackDataType;
struct Stack{
  StackDataType element[STACKSIZE];
  int top;

  void create();
  void close();
  bool push(StackDataType);
  StackDataType pop();
  bool isempty();
};
#endif
```



```
void Stack::create(){
  top = 0;
}
void Stack::close(){
}
bool Stack::push(StackDataType newelement){
  if (top < STACKSIZE) {
      element[top++] = newelement;
      return true;
  }
  return false;
}
StackDataType Stack::pop(){
  return element[--top];
}
bool Stack::isempty(){
  return (top == 0);
}
```

Had the stack data type not been a simple character, but a struct structure, the copy operation could still have been performed successfully.

The error message is due to the selected data type.

Could an error not arise here? The caller should consider the possibility of the stack being empty.

Realizing Stacks on an Array

- The code we have written correctly realizes the member functions of a stack, but it places a constraint on the maximum number of elements the stack may contain.
- This shortcoming results from using an array to store the data. (Arrays have constant sizes, and their sizes need to be decided in advance.) In fact, this is not the desired stack.
- In this realization, the stack becomes full after accepting a certain number of elements. The push operation returns an error message in the case of the stack being full!

Realizing Stacks Using Lists

- The restrictions imposed as a result of using an array to realize a stack can be removed by using linked lists.
- In list operations, it is faster to add to and remove from the beginning of the list. The whole list has to be traversed to find the end of the list.
- That is why the top of the stack is designed to be first element of the linked list.
- Pushing onto the stack can be interpreted as adding an element to the beginning of the list.
- Popping from the stack can be interpreted as removing an element from the beginning of the list.

Stack Data Structure

```
#define STACKSIZE 5
typedef char StackDataType;

struct Stack{
  StackDataType element[STACKSIZE];
  int top;

  void create();
  void close();
  bool push(StackDataType);
  StackDataType pop();
  bool isempty();
};
```

```
typedef char StackDataType;
struct Node{
        StackDataType data;
        Node *next;
};
```

Node *head;

void push(StackDataType);


```
#include <iostream>
#include <string.h>
#include "stack_1.h"

void Stack::create(){
  head = NULL;
}
void Stack::close(){
  Node *p;
  while (head){
        p = head;
        head = head->next;
        delete p;
  }
}
```

```
void Stack::push(StackDataType newdata){
        Node *newnode = new Node;
        newnode->data = newdata;
        newnode->next = head;
        head = newnode;
}

StackDataType Stack::pop(){
        Node *topnode;
        StackDataType temp;
        topnode = head;
        head = head->next;
        temp = topnode->data;
        delete topnode;
        return temp;
}

bool Stack::isempty(){
        return head == NULL;
}
```



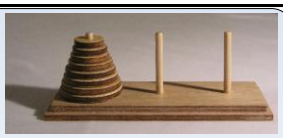
```
int main(){

  Stack s;
  s.create();
  s.push('A');
  s.push('B');
  char c = s.pop();
  c = s.pop();
  // if (!s.isempty())
      s.pop();
  s.close();

  return EXIT_SUCCESS;
}
```

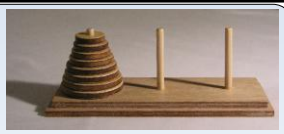
Since the stack is empty, the pop operation will give an error. This error will not show up during the compilation phase, but will occur when the program is running.

Towers of Hanoi



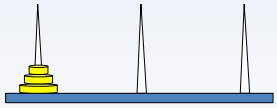
- **Towers of Hanoi** is a mathematical game.
- It consists of three pegs and disks of various sizes.
- You can slide these disks onto any peg you like.
- The puzzle starts off with the disks ordered from smallest to largest on a peg (the smallest disk is at the top).
- Thus, a conical shape has been created.

Towers of Hanoi

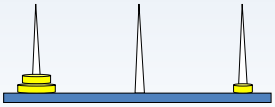


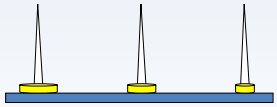
The aim of the game is to move all disks to another peg based on the following rules:

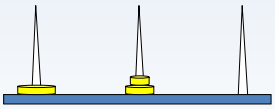
- In each move, we can move only one disk.
- Each move consists of removing the topmost disk from one of the pegs and sliding it onto another peg. Other disks may already be present on this new peg.
- No disk may be placed on top of a disk smaller than itself.

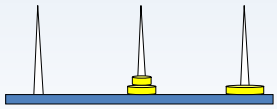


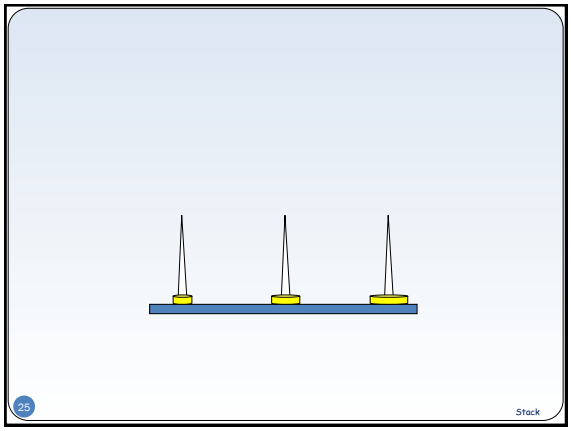
n = 3

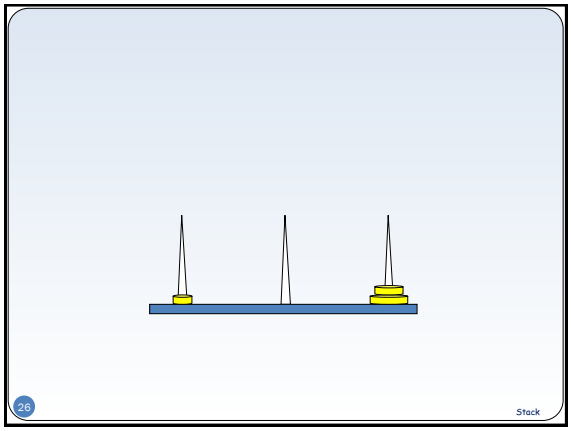


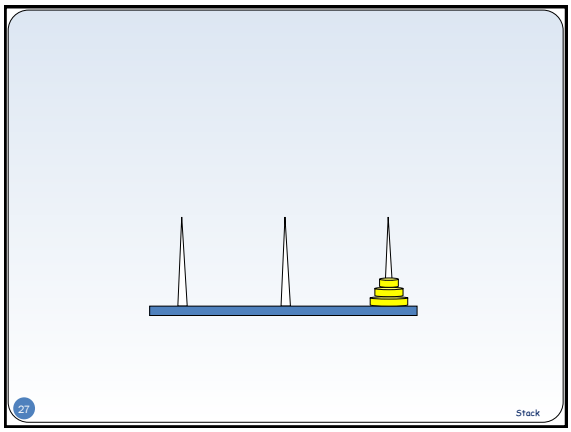


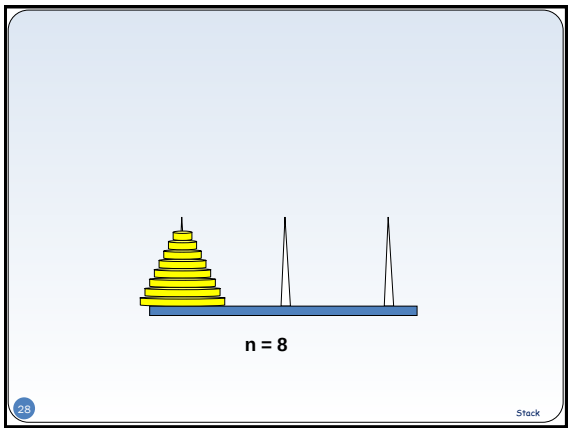


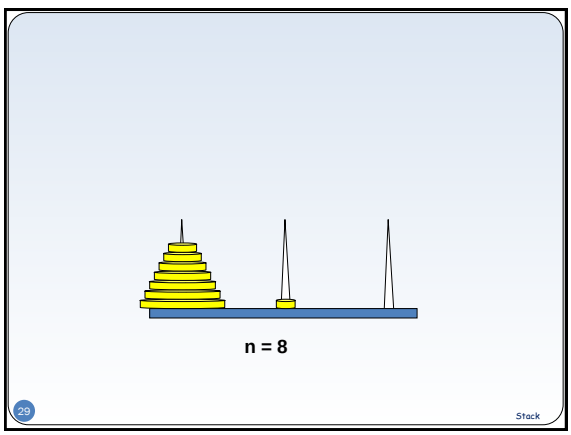


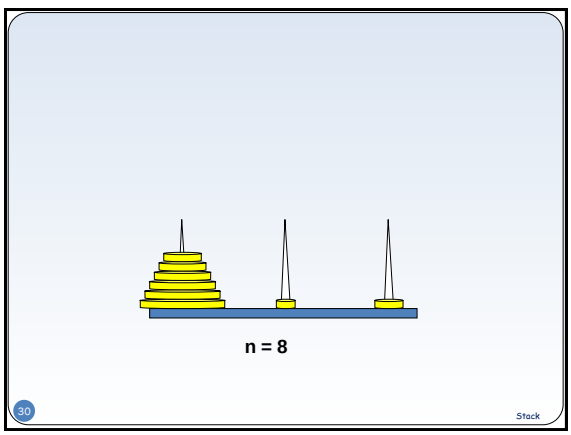

25
Stack

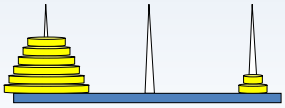

26
Stack


27
Stack

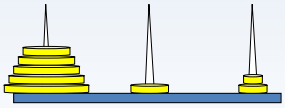

n = 8
28
Stack


n = 8
29
Stack

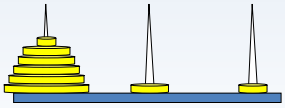

n = 8
30
Stack



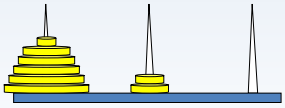
n = 8



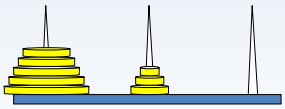
n = 8



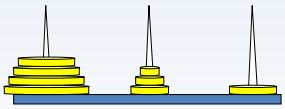
n = 8



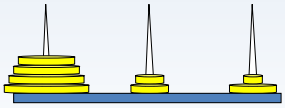
n = 8



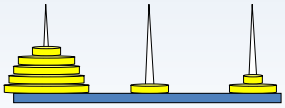
n = 8



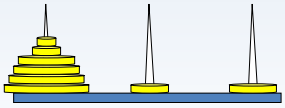
n = 8



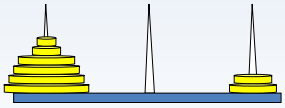
n = 8



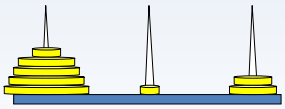
n = 8



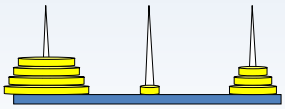
n = 8



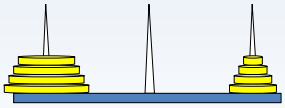
n = 8



n = 8



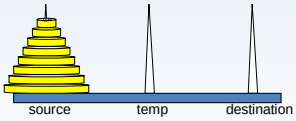
n = 8



n = 8

Problem Solution

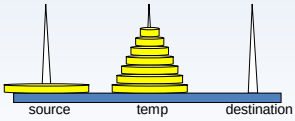
First, n - 1 disks are copied to a temporary place.



n = 8

Problem Solution

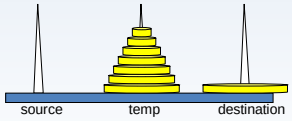
First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.



n = 8

Problem Solution

First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.
The problem has been solved when n - 1 disks in the temporary place have been copied to the destination.



n = 8

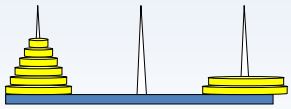
Problem Solution

We try to solve the problem of n - 1 disks being copied to a temporary place similarly:
n - 2 disks are copied to a temporary place.
Thus, (n - 1)st disk is copied to a place above nth disk.

n = 8

Problem Solution

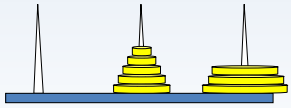
n - 2 disks are copied to a temporary place.
Thus, (n - 1)st disk is copied to a place above nth disk.
We continue the operation in this manner by decreasing n.
The movement of each (n - x)th disk from one place to another should be handled as a similar subproblem.



n = 8

Problem Solution

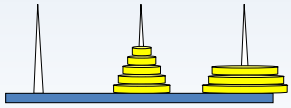
First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.
We try to solve the problem of n - 1 disks being copied to a temporary place similarly.
Then, n - 2 disks are copied to a temporary place.
Thus, (n-1)th disk is copied to a place above nth disk.
We continue the operation in this manner by decreasing n.
The movement of each (n-x)th disk from one place to another should be handled as a similar subproblem.



n = 8

Problem Solution

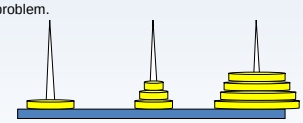
First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.
We try to solve the problem of n - 1 disks being copied to a temporary place similarly.
Then, n - 2 disks are copied to a temporary place.
Thus, (n-1)th disk is copied to a place above nth disk.
We continue the operation in this manner by decreasing n.
The movement of each (n-x)th disk from one place to another should be handled as a similar subproblem.



n = 8

Problem Solution

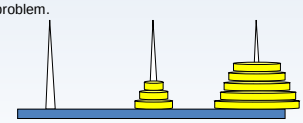
First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.
We try to solve the problem of n - 1 disks being copied to a temporary place similarly.
Then, n - 2 disks are copied to a temporary place.
Thus, (n-1)th disk is copied to a place above nth disk.
We continue the operation in this manner by decreasing n.
The movement of each (n-x)th disk from one place to another should be handled as a similar subproblem.



n = 8

Problem Solution

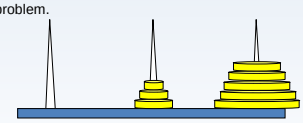
First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.
We try to solve the problem of n - 1 disks being copied to a temporary place similarly.
Then, n - 2 disks are copied to a temporary place.
Thus, (n-1)th disk is copied to a place above nth disk.
We continue the operation in this manner by decreasing n.
The movement of each (n-x)th disk from one place to another should be handled as a similar subproblem.



n = 8

Problem Solution

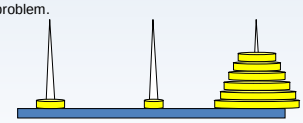
First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.
We try to solve the problem of n - 1 disks being copied to a temporary place similarly.
Then, n - 2 disks are copied to a temporary place.
Thus, (n-1)th disk is copied to a place above nth disk.
We continue the operation in this manner by decreasing n.
The movement of each (n-x)th disk from one place to another should be handled as a similar subproblem.



n = 8

Problem Solution

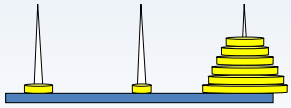
First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.
We try to solve the problem of n - 1 disks being copied to a temporary place similarly.
Then, n - 2 disks are copied to a temporary place.
Thus, (n-1)th disk is copied to a place above nth disk.
We continue the operation in this manner by decreasing n.
The movement of each (n-x)th disk from one place to another should be handled as a similar subproblem.



n = 8

Problem Solution

First, n - 1 disks are copied to a temporary place.
Then, the nth disk gets copied to the destination.
We try to solve the problem of n - 1 disks being copied to a temporary place similarly.
Then, n - 2 disks are copied to a temporary place.
Thus, (n-1)th disk is copied to a place above nth disk.
We continue the operation in this manner by decreasing n.
The movement of each (n-x)th disk from one place to another should be handled as a similar subproblem.

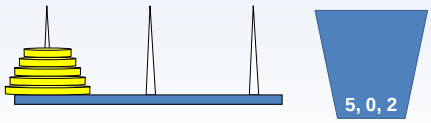


n = 8

The problem could be solved iteratively using 4 stacks:

- 3 stacks for the pegs that hold the disks,
- 1 move stack that stores the moves as the operations are being performed

- The program starts by sliding n disks onto the first peg.
- The first move (that is, the goal of the system, i.e., n disks being transferred from stack 1 to stack 3) is copied onto the move stack
  - example: 5, 0, 2: five disks being copied from s[0] to s[2]

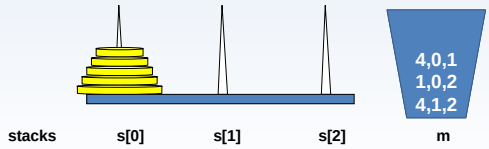



```
Stack s[3]; // the 3 pegs that hold the disks
int main() {
  for (int i = 0; i < 3; i++) {
     s[i].create();
  }
  for (int i = 0; i < 5; i++) { // disk n = 5
     s[0].push(5 - i);   // pushed onto stack 0
  }
  Hanoi_iterative(5);
  for (int i = 0; i < 3; i++) {
     s[i].close();
  }
  return EXIT_SUCCESS;
}
```



```
void Hanoi_iterative(int n) {
     Move_Stack m;
     m.create();
     StackMoveType move = {5, 0, 2};
     m.push(move);
     .
     .
     .
     .
}
```


- Then, as long as the move stack is full, we pop a move (n, source, destination) from this stack, and push these moves to the stack:
  - (n-1, temp, destination)
  - (1, source, destination)
  - (n-1, source, temp) (the move that should be executed first is at the top of the stack)



```
void Hanoi_iterative(int n) {
   Move_Stack m;
   m.create();
   StackMoveType move = {n, 0, 2};
   m.push(move);
   while ( !m.isempty() ) {
         move = m.pop();
         if (move.n == 1)
                s[move.destination].push(s[move.source].pop());
         else {
                int temp = 0 + 1 + 2 - move.destination - move.source;
                StackMoveType newmove = {move.n-1, temp, move.destination};
                m.push(newmove);
                newmove.n = 1;
                newmove.source = move.source;
                newmove.destination = move.destination;
                m.push(newmove);
                newmove.n = move.n-1;
                newmove.source = move.source;
                newmove.destination = temp;
                m.push(newmove);
         }
   }
   m.close();
}
```


To do: In recitation

Infix – Postfix Conversion