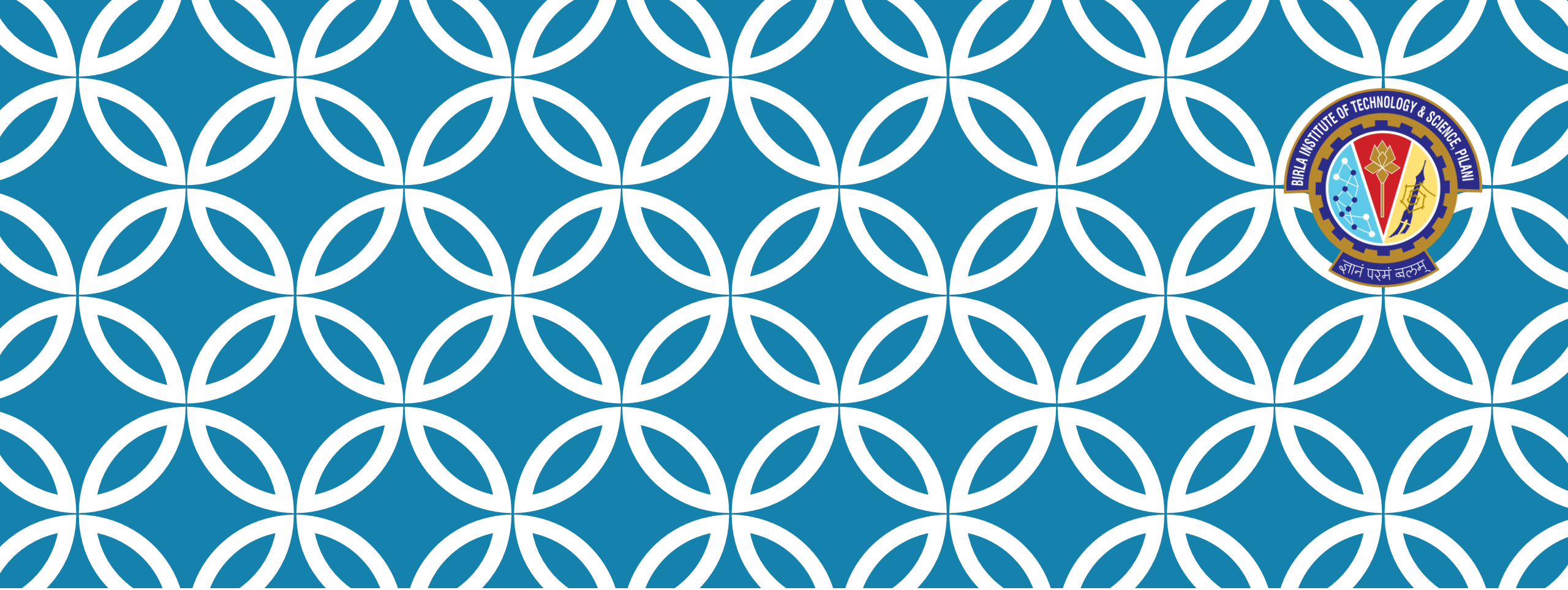


BITS F232: FOUNDATIONS OF DATA STRUCTURES & ALGORITHMS (1ST SEMESTER 2023-24) STACK ADT

Chittaranjan Hota, PhD
Sr. Professor of Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

RECURSIVE FUNCTIONS: RECURRENCE RELATION (EX1)

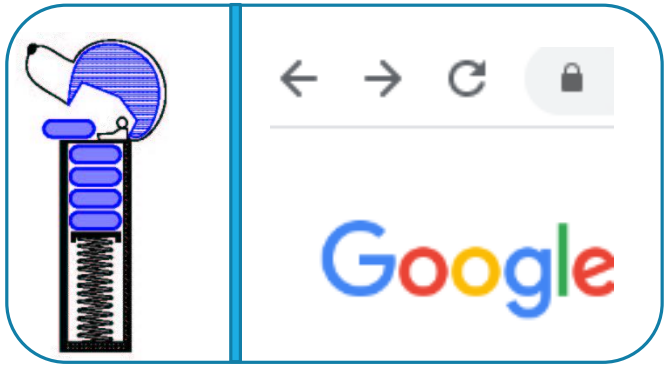
```
void fun(int n) {
   if (n > 0)
   {
     printf("%d", n);
     fun(n-1);
   }
}

Int main() {
  int x = 4;
  fun(x);
  return 0;
}
```

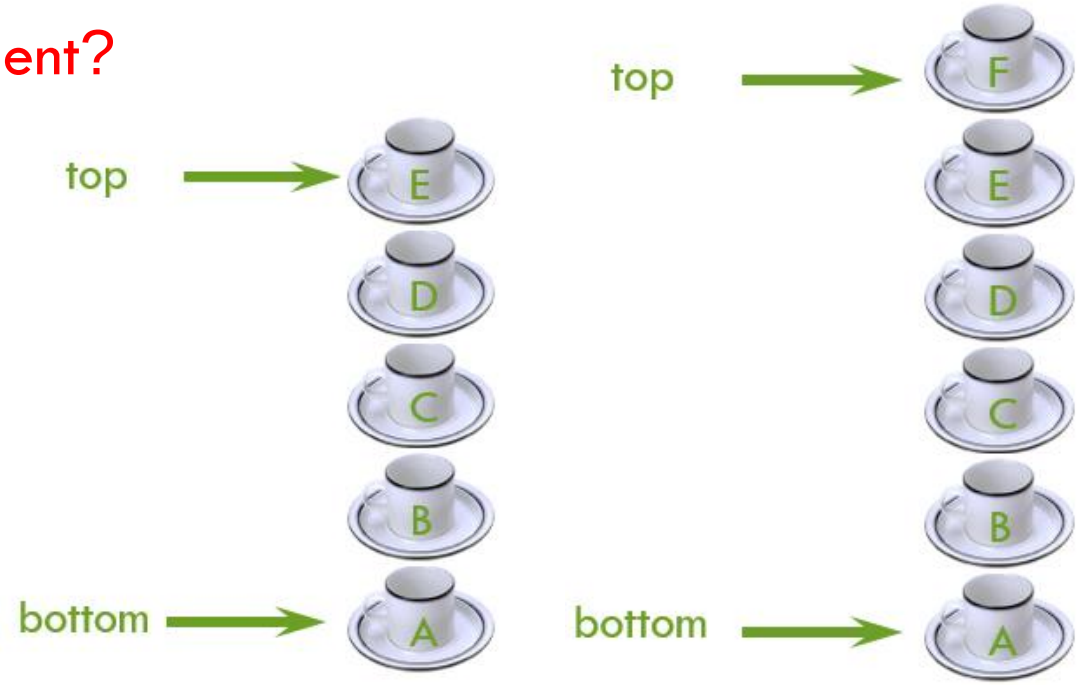
EXAMPLE 2

$T(n) = 2\,T(n/2) + n$ Where, $T(1) = 1$

STACK ADT

- A stack S is a linear sequence of elements to which elements x can only be inserted and deleted from the head of the list in the order they appear.
- What type of policy does a stack implement?

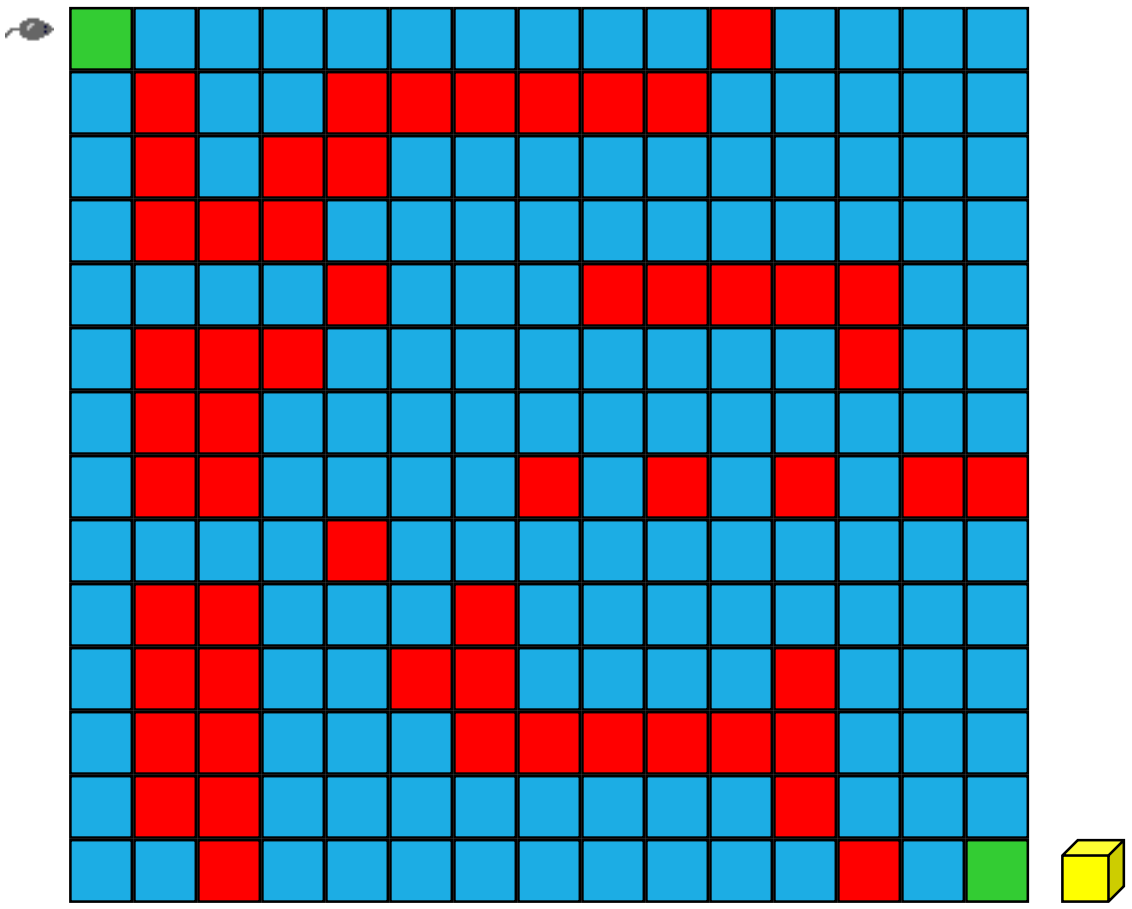


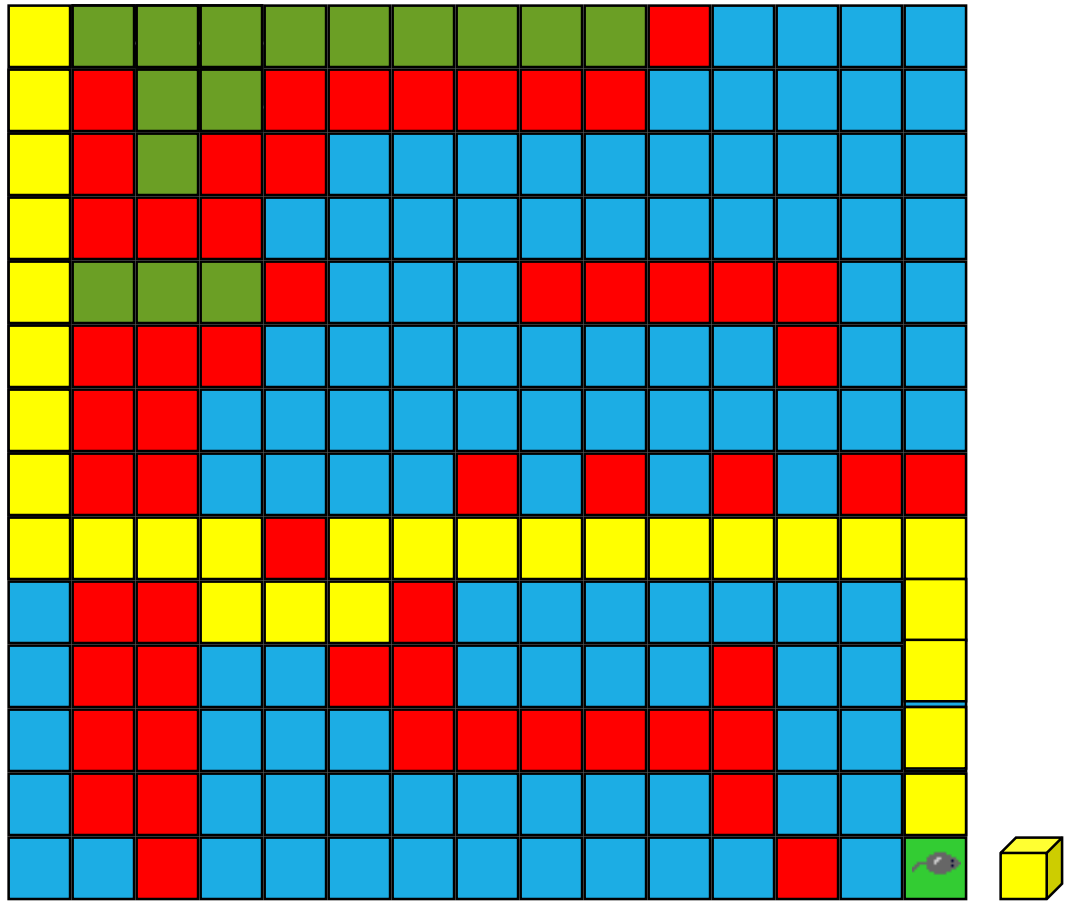


Example usage: Parenthesis matching

Depth first search, expression conversion/evaluation etc.

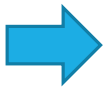
ANOTHER EXAMPLE USAGE: RAT IN A MAZE





STACK USAGES CONTINUED…

```
main () {
        int i = 5;
        foo(i);
}

foo (int j)

{
         int k;
         k = j+1;
         bar(k);

}

bar (int m) {

        int h = 7;

        int l = m + h;

}
```



(Runtime Stack)

(Stock span: last 5 days)

ABSTRACT DATA TYPES (ADT) & STACK INTERFACE

- An ADT is an abstraction of a data structure. Specifies data stored, operations on data and error conditions associated with operations on these data, if any.
- Recollect the GRAPH ADT we used for traffic light system design (in the introductory lectures).
- Another example: Stock Trading(Data: Orders; Operations: Buy/Sell/Cancel; Errors: Buying non-existent stock etc).
- STACK ADT: (Data: Arbitrary objects; Operations: Push, Pop; Auxiliary operations: top (returns the last inserted element without removing it, size (returns the no. of elements stored, empty (if no elements are there in the stack).

```
#include <iostream>
using namespace std;

template <typename E>
class ArrayStack {
    enum { DEF_CAPACITY = 100 };
    public:
        ArrayStack(int cap = DEF_CAPACITY);
        int size() const;
        bool empty() const;
        const E& top();
        void push(const E& e);
        void pop();
    private:
        E* S;     // array of stack elements
        int capacity;
        int t;
};
```

ARRAY-BASED STACK IMPLEMENTATION

```
56  template <typename E>
57  class ArrayStack
58  {
59      enum
60      {
61          DEF_CAPACITY = 100
62      }; // default stack capacity
63  public:
64      ArrayStack(int cap = DEF_CAPACITY);
65      int size() const;
66      bool empty() const;
67      const E &top();
68      void push(const E &e);
69      void pop();
70
71  private:
72      E *S;
73      int capacity;
74      int t;
75  };
76
77  template <typename E> // push element
78  void ArrayStack<E>::push(const E &e)
79  {
80      if (size() == capacity)
81          cout << "Push to full stack\n";
82      S[++t] = e;
83  }
```

COMPLEXITY?

```
85  template <typename E> // pop the stack
86  void ArrayStack<E>::pop()
87  {
88      if (empty())
89          cout << "Pop from empty stack\n";
90      --t;
91  }
```

```
97  template <typename E>
98  int ArrayStack<E>::size() const
99  {
100     return (t + 1);
101 } // number of items in the stack
102
103 template <typename E>
104 bool ArrayStack<E>::empty() const
105 {
106     return (t < 0);
107 } // is the stack empty?
108
109 template <typename E> // return top of stack
110 const E &ArrayStack<E>::top()
111 {
112     if (empty())
113         cout << "Top of empty stack\n";
114     return S[t];
115 }
```