COMPUTER SCIENCE 123

DATA STRUCTURES

ABSTRACT DATA TYPE

DATA STRUCTURE

TREES

BST

AVL

HEAPS

ANALYSIS OF ALGORITHMS

GRAPHS

SORTING ALGORITHMS

HASHING

ABSTRACT DATA TYPE

DATA STRUCTURE

DATA STRUCTURE

Organization of data and its storage allocation in a computer.

DATA STRUCTURE

Array
Record
Singly-Linked List
Doubly-Linked List
Adjacency Matrix

ABSTRACT DATA TYPE (ADT)

A mathematical model, together with various operations defined in the model.

ABSTRACT DATA TYPE (ADT)

List ADT

insert();

delete();

find();

ABSTRACT DATA TYPE (ADT)

Set ADT

union();

intersection();

size();

ABSTRACT DATA TYPE (ADT)

Graph ADT

addVertex();

subGraph();

merge();

ADT

The List ADT

DATA STRUCTURE

implemented using

Array or Linked-List

ADT

The Graph ADT

DATA STRUCTURE

implemented using

Adjacency Matrix or
Adjacency List

ADTs

- LIST
- STACK
- QUEUE

LIST

Abstract Data Type

LIST ADT

A sequence of zero or more elements of the same type.

LIST ADT

$$a_1, a_2, a_3, \ldots a_n$$

n = size of the list

i = position of a_i

LIST ADT Operations

insert
delete
find
findKth

next, previous
printList
makeNull

List ADT

Array IMPLEMENTATION

Easy to implement.
Fast operation: **`findKth`**.
Size is fixed (compile time or run-time).
Expensive operations: **`insert`** and **`delete`**.

Array IMPLEMENTATION

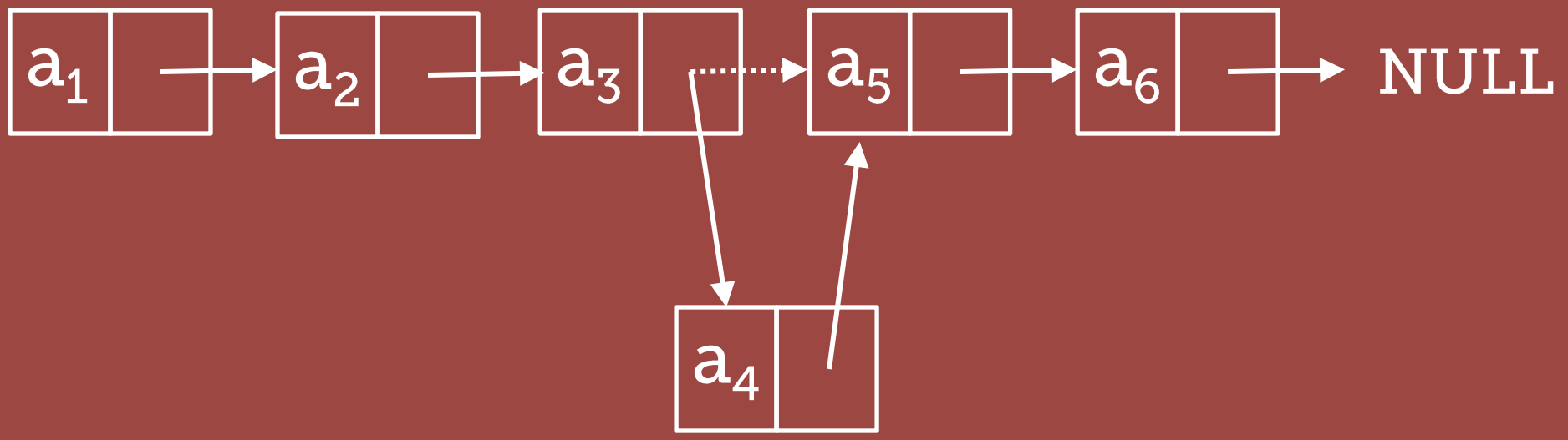
List ADT

Linked List IMPLEMENTATION

Size can grow or shrink.
Easier to do: **insert** and **delete**.
Difficult to implement (?).
Implementation issues (?).
findKth no longer fast.

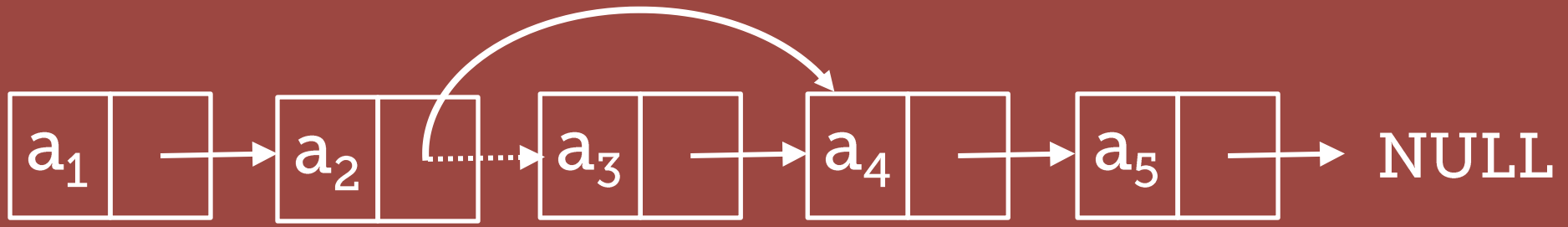
Linked List IMPLEMENTATION

insert



Linked List IMPLEMENTATION

delete



Singly-Linked List

IMPLEMENTATION

```
typedef struct node{
    int num;
    struct node *next;
}list;

list *head;
```

Singly-Linked List IMPLEMENTATION

```
head = (list *)malloc(sizeof(list));
head->num = 1;
head->next = NULL;
```

Singly-Linked List IMPLEMENTATION

```
temp = (list *)malloc(sizeof(list));
temp->num = 1;
temp->next = NULL;
head->next = temp;
```

Singly-Linked List IMPLEMENTATION

```
p = head;
while(there is data){
     temp = (list *)malloc(sizeof(list));
     temp->num = data;
     temp->next = NULL;
     p->next = temp;
     p = p->next;
}
```

Singly-Linked List IMPLEMENTATION

```
void printList(list *head){
    list *temp;
    temp = head;
    while(temp!=NULL){
        print temp->num;
        temp = temp->next;
    }
}
```

Singly-Linked List IMPLEMENTATION

Circular Linked List

IMPLEMENTATION

```
typedef struct node{
    int num;
    struct node *next;
}list;

list *ptr;
```

Circular Linked List IMPLEMENTATION

```
ptr = (list *)malloc(sizeof(list));
ptr->num = 1;
ptr->next = ptr;
```

Circular Linked List IMPLEMENTATION

```
p = ptr;
while(there is data){
     temp = (list *)malloc(sizeof(list));
     temp->num = data;
     temp->next = ptr;
     p->next = temp;
     p = p->next;
}
```

Circular Linked List IMPLEMENTATION

```
void printList(list *ptr){
     list *temp;
     temp = ptr;
     do{
          print temp->num;
          temp = temp->next;
     }while(temp!=ptr);
}
```

Circular Linked List IMPLEMENTATION

Doubly-Linked List

IMPLEMENTATION

```
typedef struct node{
    int num;
    struct node *prev, *next;
}list;

list *head;
```

Doubly-Linked List IMPLEMENTATION

```
head = (list *)malloc(sizeof(list));
head->num = 1;
head->prev = NULL;
head->next = NULL;
```

Doubly-Linked List IMPLEMENTATION

```
p = head;
while(there is data){
      temp = (list *)malloc(sizeof(list));
      temp->num = data;
      temp->prev = p;
      temp->next = NULL;
      p->next = temp;
      p = p->next;
}
```

Doubly-Linked List IMPLEMENTATION

STACK

Abstract Data Type

STACK ADT

A List with a restriction:

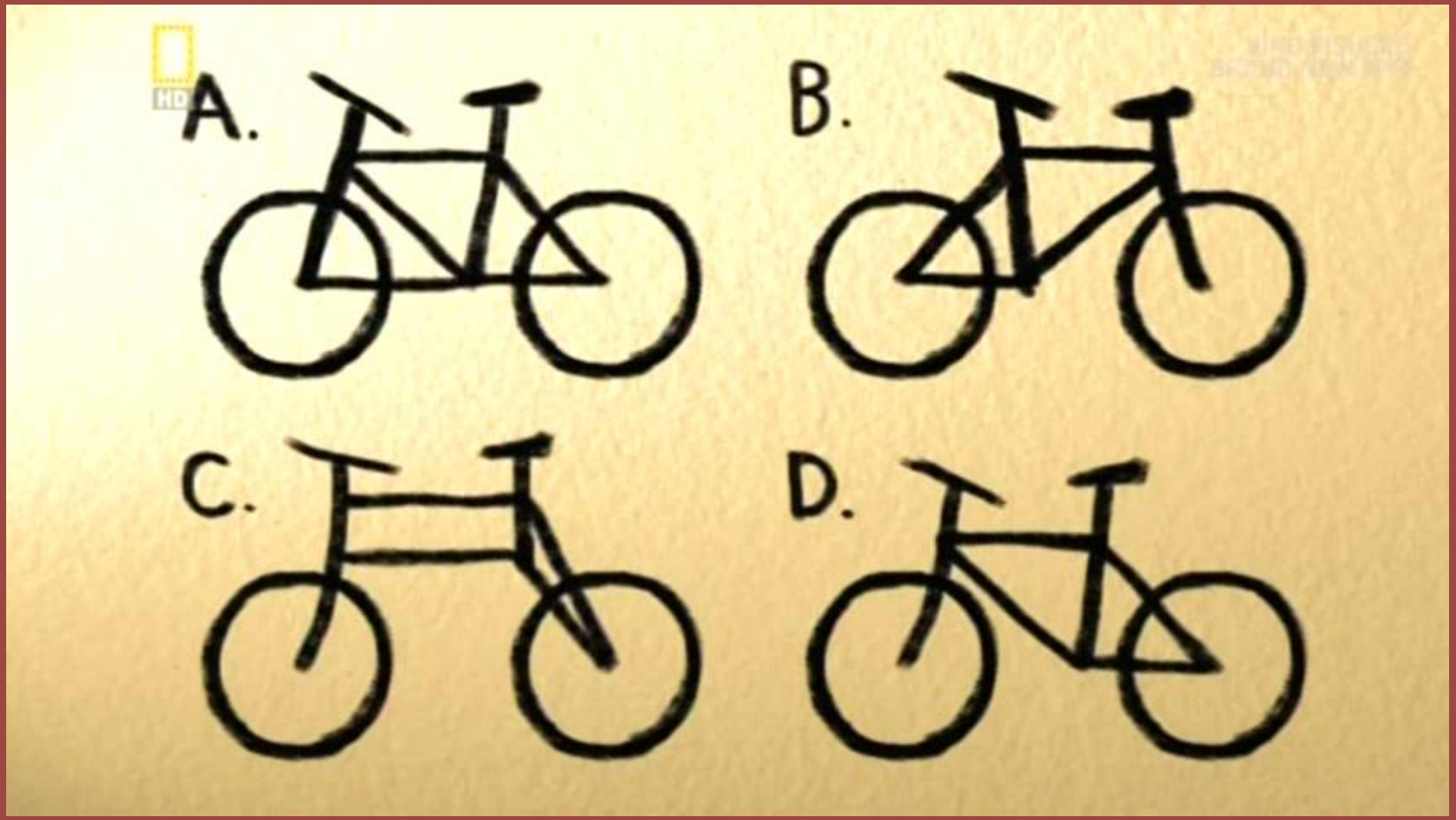
STACK ADT

`insert` and `delete` can be performed in only one position:
end of the list called TOP.

STACK ADT

LIFO | Last In, First Out


NOW
CRISPIER
Pringles
CHEDDAR
CHEESE
Super
Stack


A.
B.
C.
D.

STACK ADT

TOS | Top of Stack

STACK ADT Operations

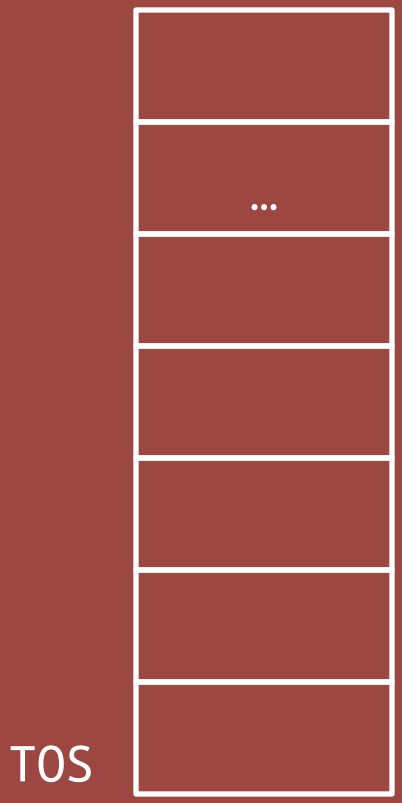
push | equivalent to insert

STACK ADT Operations

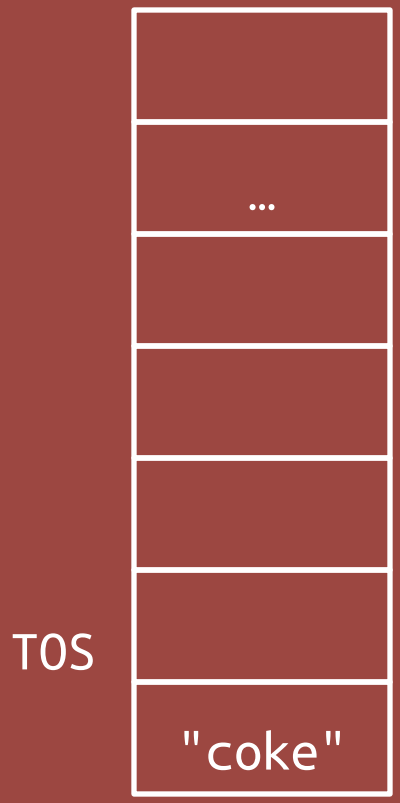
pop

equivalent to

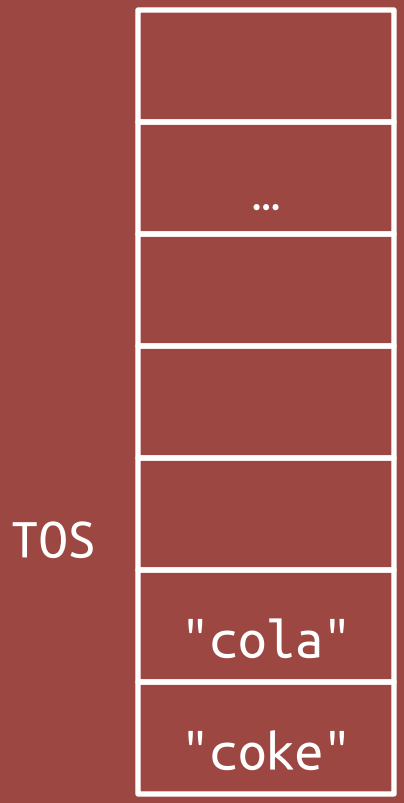
delete



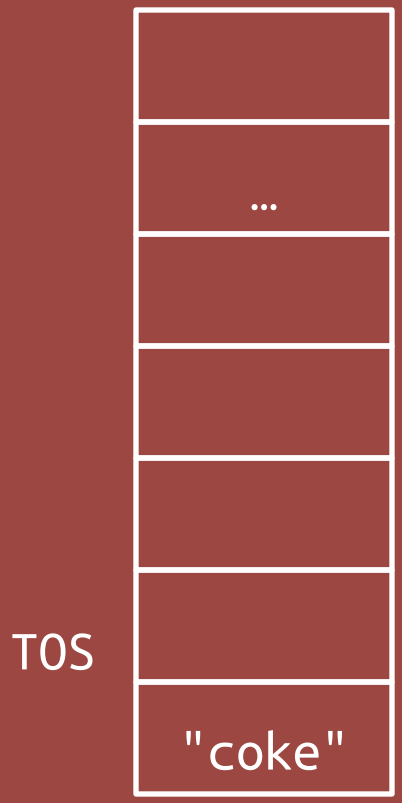

```
push("coke");
push("cola");
x = pop();
push("pepsi");
x = pop();
x = pop();
```



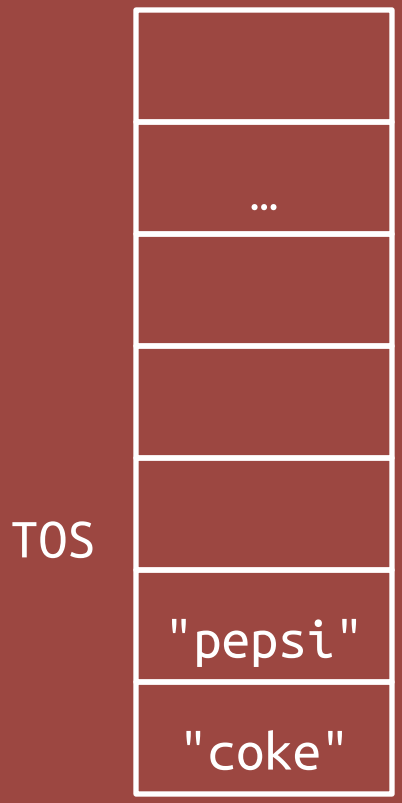

```
push("coke");
push("cola");
x = pop();
push("pepsi");
x = pop();
x = pop();
```



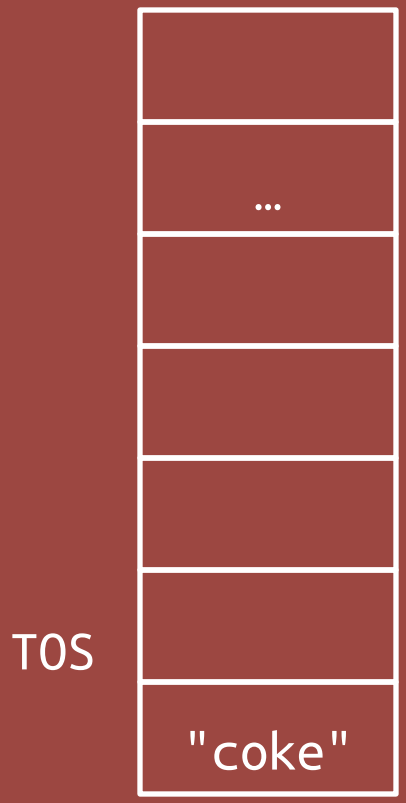

```
push("coke");
push("cola");
x = pop();
push("pepsi");
x = pop();
x = pop();
```



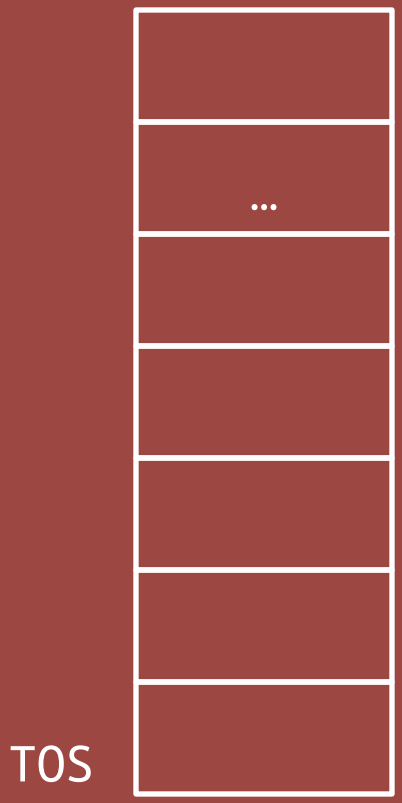

```
push("coke");
push("cola");
x = pop();
push("pepsi");
x = pop();
x = pop();
```




```
push("coke");
push("cola");
x = pop();
push("pepsi");
x = pop();
x = pop();
```




```
push("coke");
push("cola");
x = pop();
push("pepsi");
x = pop();
x = pop();
```




```
push("coke");
push("cola");
x = pop();
push("pepsi");
x = pop();
x = pop();
```

STACK

POSSIBLE ERRORS

Stack Underflow

attempt to **pop** a value from an **empty** stack.

Stack Overflow

attempt to **push** a value into a **full** stack.

STACK
Array IMPLEMENTATION

```
#define LIMIT 1000
int stack[LIMIT];
int top = 0;
```

Array IMPLEMENTATION

```
void push(int x){

    if (top < LIMIT)
        stack[top++] = x;
    else
        stack overflow

}
```

Array IMPLEMENTATION

```
int pop(){

    if (top > 0)
        return stack[--top];
    else
        stack underflow

}
```

Array IMPLEMENTATION

STACK

Singly-Linked List

IMPLEMENTATION

```
typedef struct node{
    int value;
    struct node *next;
}stack;

…
stack *top = NULL;
```

Singly-Linked List

IMPLEMENTATION

```
void push(int x, stack *top){
   stack *temp;
   temp=(stack*)malloc(sizeof(stack));

   if(temp == NULL){
      stack overflow
   }
   else{
      temp->value = x;
      temp->next = top;
      top = temp;
   }
}
```

Singly-Linked List

IMPLEMENTATION

```
int pop(stack *top){
    stack *temp; int x;

    temp = top;
    if(temp == NULL){
      stack underflow
    }
    else{
      top = top->next;
      x = temp->value;
      free(temp);
      return x;
    }
}
```

Singly-Linked List
IMPLEMENTATION

STACK

ADT

Applications

```
#DFS
my @STACK;
my %visited;
push @STACK, $arb;
do{
    my $v = pop @STACK;
    if(!exists $visited{$v}){
        $visited{$v}=1;
        print "$v\n";
        for (keys %{ $adjMST{$v} }){
            my $s = $_;
            if($s ne 0){
                push @STACK, $s;
            }
        }
    }
}while(@STACK ne 0);
```

Balancing Symbols

```
make an empty stack
read characters until end of file:
    if the character is an open symbol
        push it onto the stack.
    if it is a close symbol
        if the stack is empty
            report error
        else
            pop the stack
            if the symbol does not
            correspond to the
            opening symbol
                report error
if the stack is not empty
    report error
```

4 6 + 3 5 + * 2 *

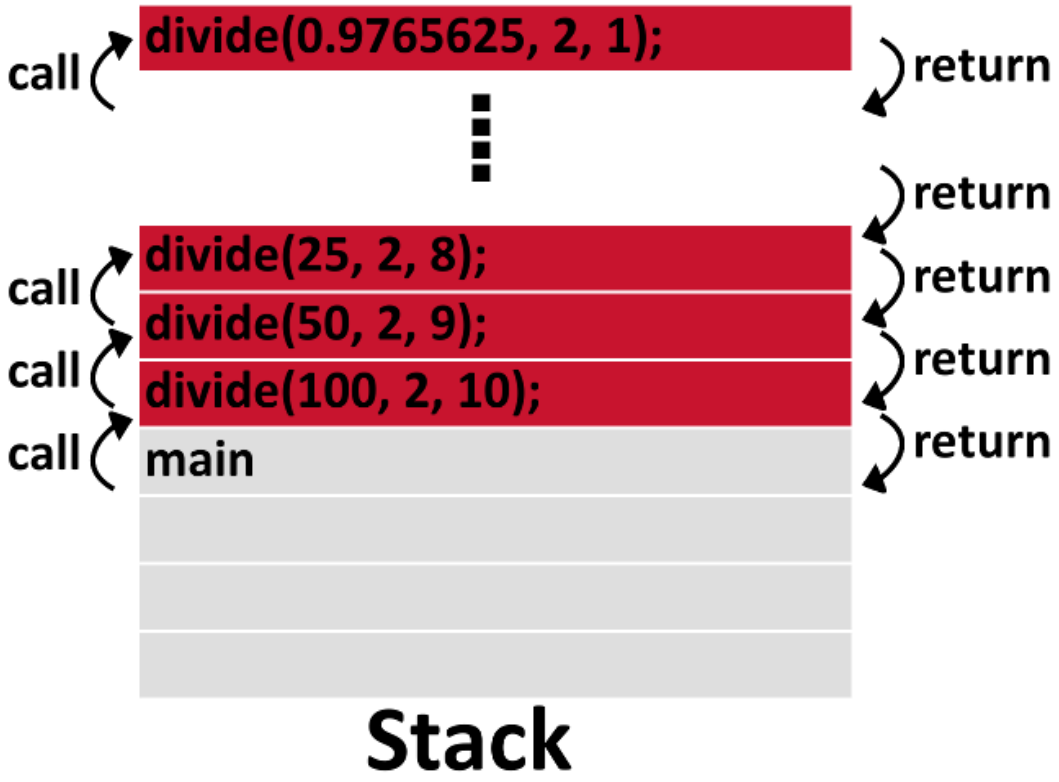
Postfix Expressions

```
make an empty stack
read characters until end of file:
    if a number is encountered
        push it onto the stack.
    if it is an operator symbol
        apply it to the two numbers that
        are popped.
        push the result onto the stack
```

4 + 6 * 3 + 5 * 2

4 6 + 3 5 + * 2 *

Infix to Postfix Conversion



Function Calls

QUEUE

Abstract Data Type

tail head

QUEUE ADT

A List with a restriction:

QUEUE ADT

`insert` is done at one end, whereas
`delete` is performed at the other end.

QUEUE ADT

FIFO | First In, First Out

QUEUE ADT Operations

enqueue | equivalent to insert

QUEUE ADT Operations

dequeue | equivalent to delete

head tail


```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```

head

tail

```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```

head

tail

```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```


```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```

tail

head

```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```

head

tail

```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```


```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```

tail

head

```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```

head

tail

```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```

head

tail

```
enqueue(45);
enqueue(6);
x = dequeue();
enqueue(123);
x = dequeue();
x = dequeue();
```

QUEUE

POSSIBLE ERRORS

Queue Underflow

attempt to **dequeue** a value from an **empty** queue.

Queue Overflow

attempt to **enqueue** a value into a **full** queue.

QUEUE

Array IMPLEMENTATION


123
42
66
97
head
tail

tail

head

```
#define LIMIT 1000
int queue[LIMIT];
int tail = 0, head = 0;
```

Circular Array IMPLEMENTATION

```
void enqueue(int x){

    tail = (tail+1)%LIMIT;
    if(tail!=head)
        queue[tail] = x;
    else
        queue overflow

}
```

Circular Array IMPLEMENTATION

```
int dequeue(int x){

    if(head!=tail){
        head = (head+1)%LIMIT;
        return queue[head];
    }
    else
        queue underflow

}
```

Circular Array IMPLEMENTATION

QUEUE
Singly-Linked List
IMPLEMENTATION

```
typedef struct node{
    int value;
    struct node *next;
}queue;

…
queue *head = NULL;
queue *tail = NULL;
```

Singly-Linked List

IMPLEMENTATION

```
void enqueue(int x, queue *head, queue *tail){
   queue *temp;
   temp=(queue*)malloc(sizeof(queue));

   if(temp == NULL){
      queue overflow
   }
   else{
        temp->value = x;
        temp->next = NULL;


   }
}
```

Singly-Linked List
IMPLEMENTATION

```
void enqueue(int x, queue *head, queue *tail){
    queue *temp;
    temp=(queue*)malloc(sizeof(queue));

    if(temp
        queue
    }
    else{

        temp->value = x;
        temp->next = NUL

    }
}
```

```
temp->value = x;
temp->next = NULL;
```

Singly Linked List

IMPLEMENTATION

```
void enqueue(int x, queue *head, queue *tail){
    queue *temp;
    temp=(queue*)malloc(sizeof(queue));

    if(temp
        queue
    }
    else{

        temp->value = x;
        temp->next = NUL
        if(tail ==NULL)
                    hea
        else{
                    tai
                    tai
        }
    }
}
```

```
temp->value = x;
temp->next = NULL;
if(tail == NULL)
    head = tail = temp;
else{
    tail->next = temp;
    tail = temp;
}
```

Singly Linked List

IMPLEMENTATION

```
int dequeue(queue *head, queue *tail){
   queue *temp; int x;

   temp = head;
   if(temp == NULL){
      queue underflow
   }
   else{
        if(temp->next == NULL)
                    head = tail = NULL;
        else{
                    head = head->next;
        }
        x = temp->value;
        free(temp);
        return x;
   }
}
```

Singly-Linked List
IMPLEMENTATION

```
int dequeue(queue *head, queue *tail){
   queue *temp; int x;

   temp = h
   if(temp
      queue
   }
   else{

      if(temp->next ==
                  hea
      else{
                  hea
      }
      x = temp->value;
      free(temp);
      return x;
   }
}
```

```
if(temp->next == NULL)
      head = tail = NULL;
else{
      head = head->next;
}
x = temp->value;
free(temp);
return x;
```

d List

IMPLEMENTATION

QUEUE

ADT

Applications