NANYANG TECHNOLOGICAL UNIVERSITY

# CE1007/CZ1007 DATA STRUCTURES

## Lecture 5A: **Stacks**

Dr. Owen Noel Newton Fernando

**College of Engineering**

School of Computer Science and Engineering

# OUTLINE

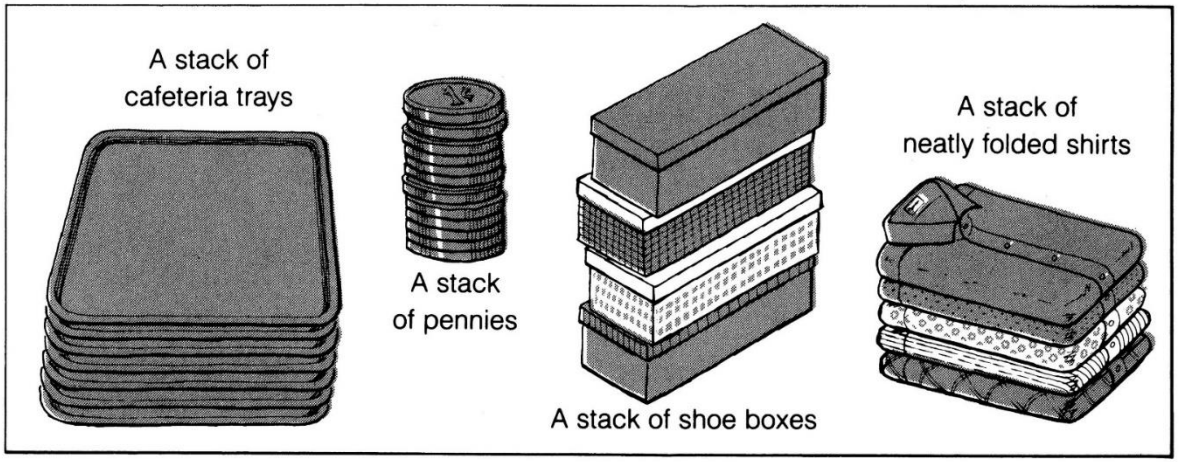
- Examples of Stacks
- Stack data structure
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Worked examples: Applications
- Array-based Stack Implementation

# YOU SHOULD BE ABLE TO...

- Explain how a stack data structure operates
- Implement a stack using a linked list
- Choose a stack data structure when given an appropriate problem to solve

# EXAMPLES OF STACKS

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack



# OUTLINE

- Examples of Stacks
- **Stack data structure**
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Worked examples: Applications
- Array-based Stack Implementation

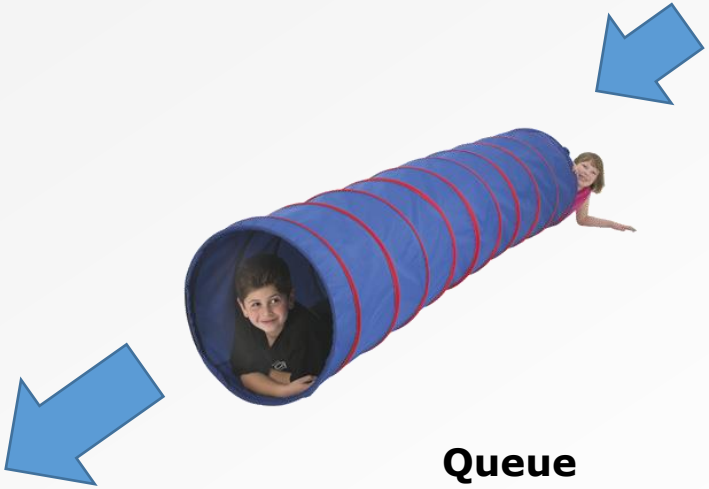
# PREVIOUSLY

- Arrays
  - **Random access** data structure
- Linked lists
  - **Sequential access** data structure
  - Have to go through a particular sequence when accessing elements
    - Temp->next until you find the right node
- Today, **another limited-access** sequential data structure

# LINKED LIST, QUEUE & STACK

Linked List

Stack



Queue

Photos here are downloaded online from google.

# STACK DATA STRUCTURE

- A Stack is a data structure that operates like a physical stack of things
  - Stack of books, for example
  - Elements can only be added or removed at the top
- Key: **Last-In, First-Out (LIFO) principle**
  - Or First-In, Last-Out (FILO)
- Built on top of one other data structure
  - Arrays, linked lists, etc.
  - We'll focus on a linked list-based implementation

# STACK DATA STRUCTURE

- Core operations
  - Push: Add an item to the top of the stack
  - Pop: Remove an item from the top of the stack
- Common helpful operations
  - Peek: Inspect the item at the top of the stack without removing it
  - IsEmptyStack: Check if the stack has no more items remaining
- Corresponding functions
  - **push()**
  - **pop()**
  - **peek()**
  - **isEmptyStack()**
- We'll build a stack assuming that it only deals with integers
  - But as with linked lists, can deal with any contents depending on how you define the functions and the underlying implementation

# OUTLINE

- Examples of Stacks
- Stack data structure
- **Stack implementation using linked lists**
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Worked examples: Applications
- Array-based Stack Implementation

# STACK IMPLEMENTATION USING LINKED LISTS

- Recall that we defined a LinkedList structure
  - Encapsulates all required variables inside a single object
  - Conceptually neater to deal with
- Similarly, define a Stack structure
  - We're going to build our stack on top of a linked list

```
typedef struct _stack{
    LinkedList ll;
} Stack;
```

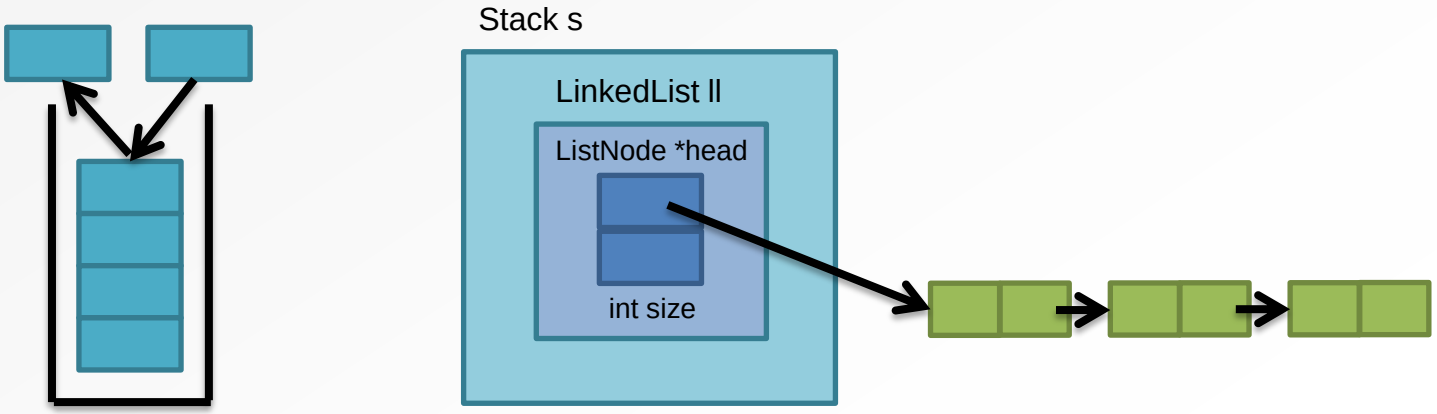
# STACK IMPLEMENTATION USING LINKED LISTS

- Stack structure

```
typedef struct _stack{
    LinkedList ll;
} Stack;
```

Notice this is a LinkedList, not a LinkedList *

- Basically wrap up a linked list and use it for the actual data storage
- Just need to ensure we control where elements are added/removed
- Notice that the LinkedList already takes care of little things like keeping track of number of nodes, etc.

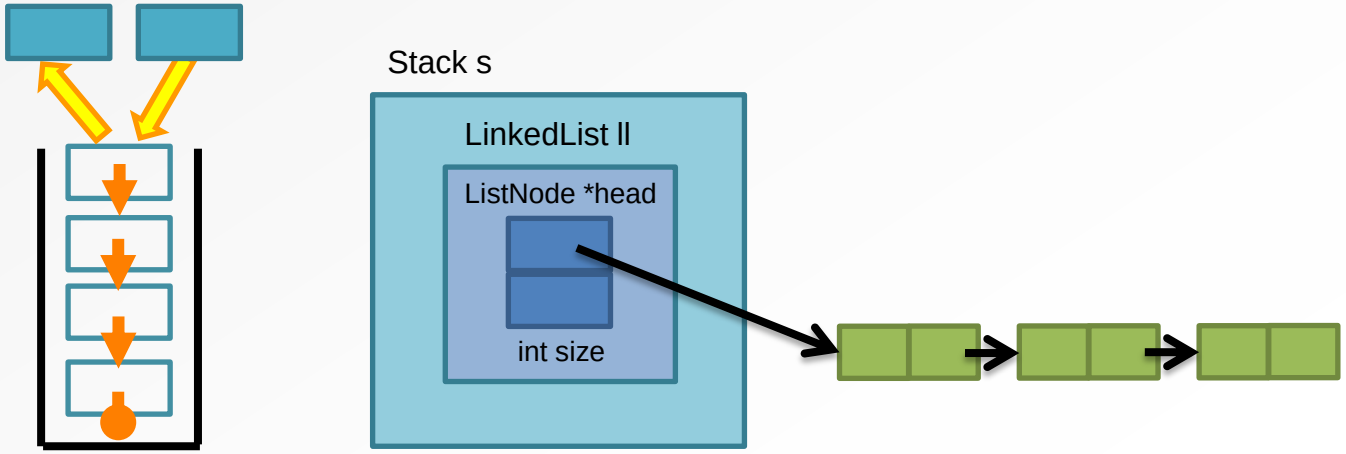


# OUTLINE

- Examples of Stacks
- Stack data structure
- Stack implementation using linked lists
- **Stack functions**
  - **push()**
  - **pop()**
  - **peek()**
  - **isEmptyStack()**
- Worked examples: Applications
- Array-based Stack Implementation

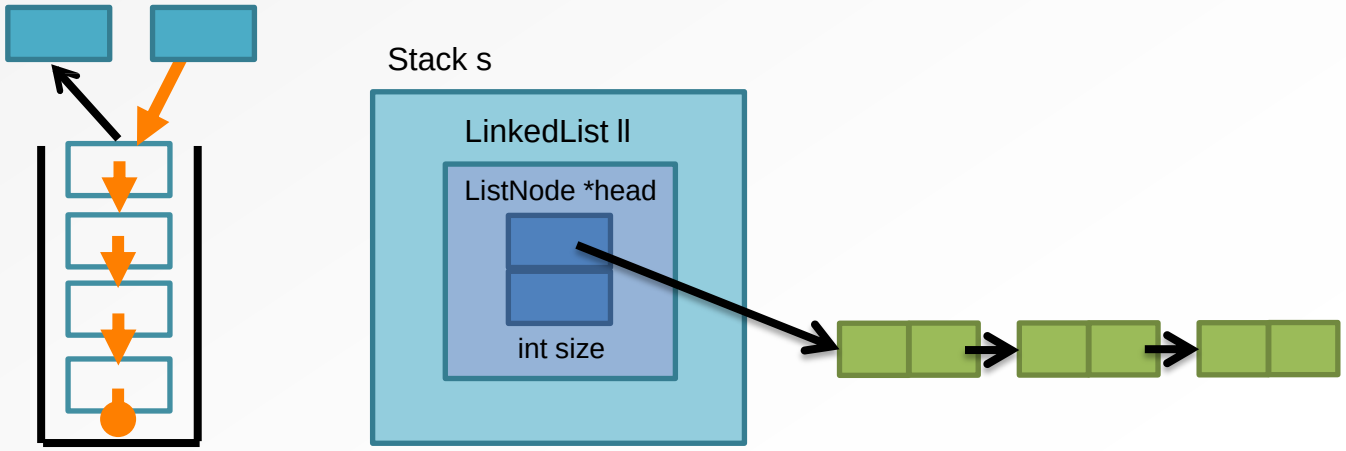
# STACK FUNCTIONS: push()

- push() function is the **only way to add** an element to the stack data structure
- Only allowed to **push()** onto the **top** of the stack
- Question:
  - Using a linked list as the underlying data storage, does the first linked list node represent the top or the bottom of the stack?



# STACK FUNCTIONS: push()

- **Write the push() function**
  - Define the function prototype
  - Implement the function
- Requirements
  - Make use of the LinkedList functions we've already defined
  - Insert at the top only (what index position?)

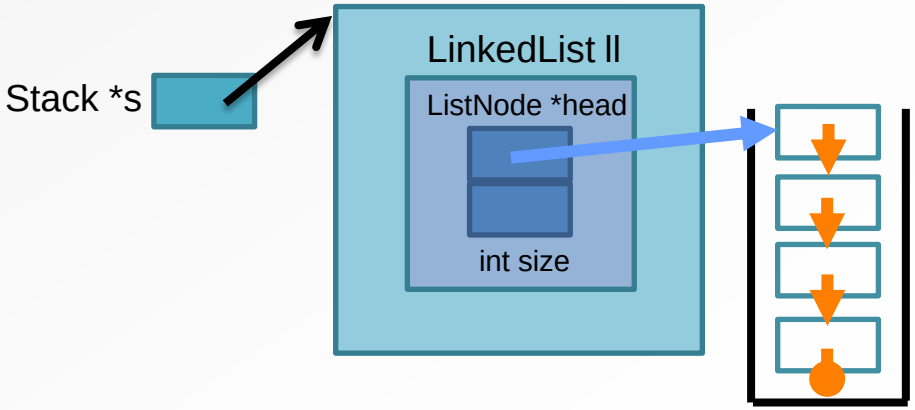


# STACK FUNCTIONS: push()

```
void push(Stack *s,                          ){



}
```



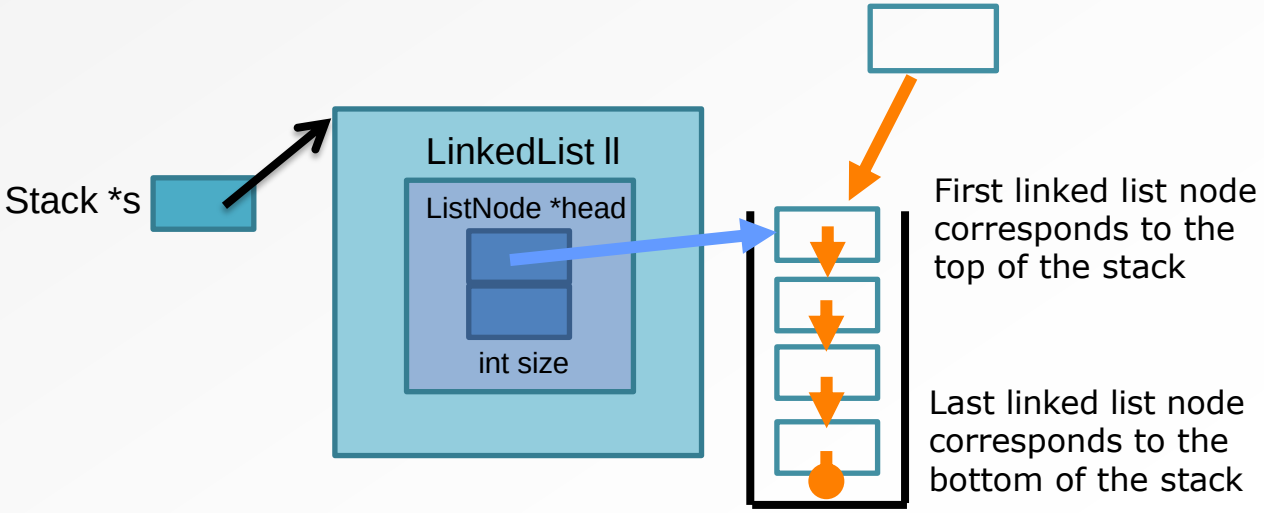
# STACK FUNCTIONS: push()

```
void push(Stack *s ,       int item      ){

  insertNode(&(s->ll), 0, item);

}
```

Pushing a new node onto the stack → adding a new node to the front of the linked list




```
int insertNode(LinkedList *ll, int index, int value);
```

# STACK FUNCTIONS: push()

- First linked list node corresponds to the top of the stack
- Last linked list node corresponds to the bottom of the stack
- Pushing a new node onto the stack ➔ adding a new node to the front of the linked list
  - Notice that this is a very efficient operation

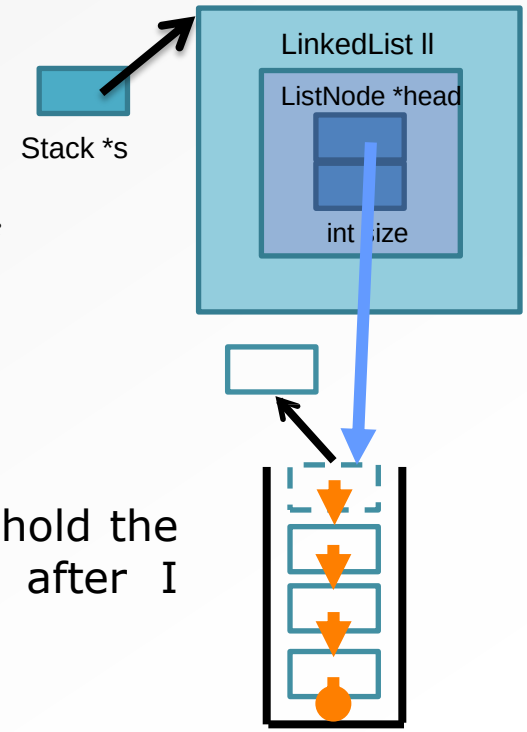
```
void push(Stack *s, int item){
    insertNode(&(s->ll), 0, item);
}
```

**`int insertNode(LinkedList *ll, int index, int value);`**

# STACK FUNCTIONS: pop()

- Popping a value off the top of the stack is a two-step process
  - Get the value of the node at the front of the linked list
  - Removing that node from the linked list

```
int pop(Stack *s){
    int item;
    item = ((s->ll).head)->item;
    removeNode(&(s->ll), 0);
    return item;
}
```

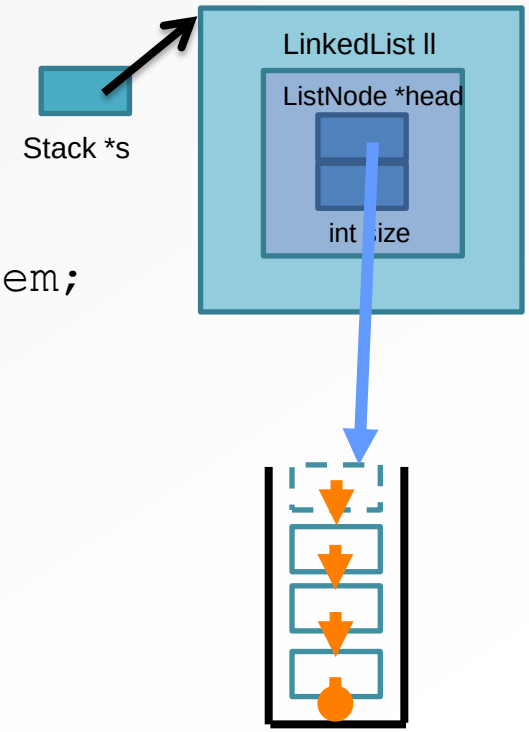



- Need a temporary int variable **item** to hold the stored value because I can't get it after I remove the top node

**`int removeNode(LinkedList *ll, int index);`**

# STACK FUNCTIONS: peek()

- Peek at the value on the top of the stack
  - Get the value of the node at the front of the linked list
    - Without removing the node




```
int peek(Stack *s){
    return ((s->ll).head)->item;
}
```

# STACK FUNCTIONS: isEmptyStack()

- Check to see if number of nodes == 0
- Make use of the built-in size variable in the LinkedList struct

```
int isEmptyStack(Stack *s){
    if ((s->ll).size == 0) return 1;
    return 0;
}
```

# OUTLINE

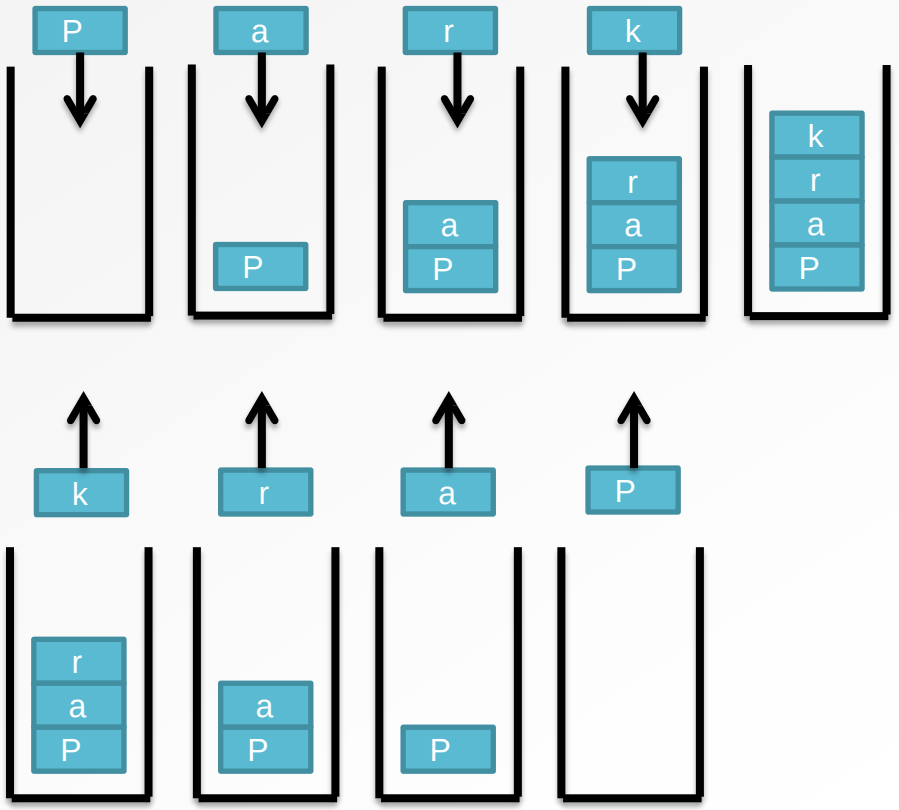
- Examples of Stacks
- Stack data structure
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- **Worked examples: Applications**
- Array-based Stack Implementation

# SIMPLE APPLICATION #1: REVERSE STRING

- Stacks are useful for reversing items
- **Reverse** a string: **Park** → **kraP**
- Idea:

# SIMPLE APPLICATION #1: REVERSE STRING

- Step 1
  - Push onto stack until no more letters
- Step 2
  - Pop from stack until stack is empty



# SIMPLE APPLICATION #2: REVERSE LIST OF INTEGERS

- Similar to previous application, but with numbers

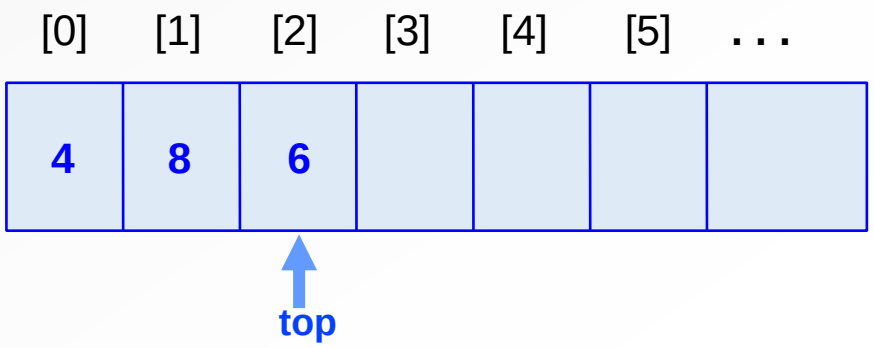
```
1   int main(){
2       int i = 0;
3       Stack s;
4       s.ll.head = NULL;
5
6       printf("Enter a number: ");
7       scanf("%d", &i);
8       while (i != -1){
9           push(&s, i);
10          printf("Enter a number: ");
11          scanf("%d", &i);
12      }
13      printf("Popping stack: ");
14      while (!isEmptyStack(&s))
15          printf("%d ", pop(&s));
16      return 0;
17  }
```

# OUTLINE

- Examples of Stacks
- Stack data structure
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Worked examples: Applications
- **Array-based Stack Implementation**

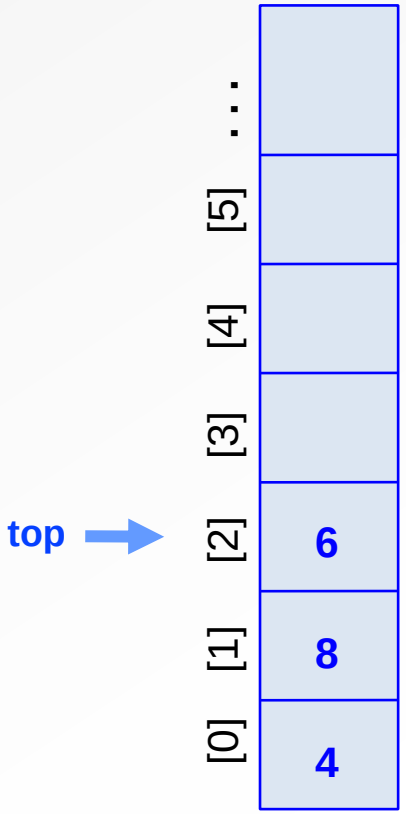
# ARRAY-BASED STACK IMPLEMENTATION

- A stack can be implemented with an array because we can only add/remove from the top.
- For example, this stack contains the integers 6 (at the top), 8 and 4(at the bottom).
- Array: **very natural and simple** implementation for stacks



# ARRAY-BASED STACK IMPLEMENTATION

- A stack can be implemented with an array because we can only add/remove from the top.
- For example, this stack contains the integers 6 (at the top), 8 and 4(at the bottom).
- Array: **very natural and simple** implementation for stacks



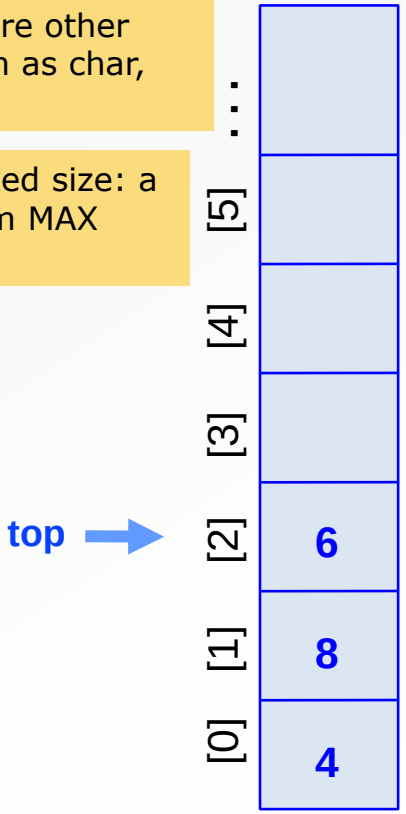
# ARRAY-BASED STACK IMPLEMENTATION

- New C structure:

```
typedef struct {
    int num[MAX];
    int top;
} stack;
```

The array can store other type of data, such as char, string, etc.

The array is of fixed size: a stack of maximum MAX elements

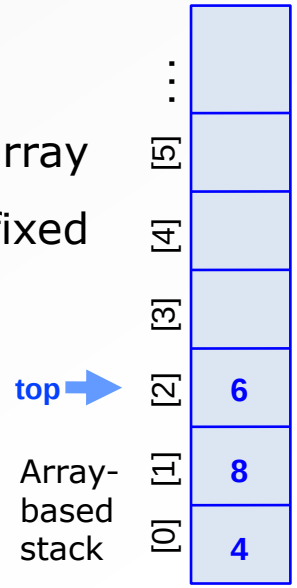



  - Bottom stack item stored at element 0
  - Last index in the array is the *top*
  - Increment *top* when one item is push(), decrement after pop()
  - size is not necessary here

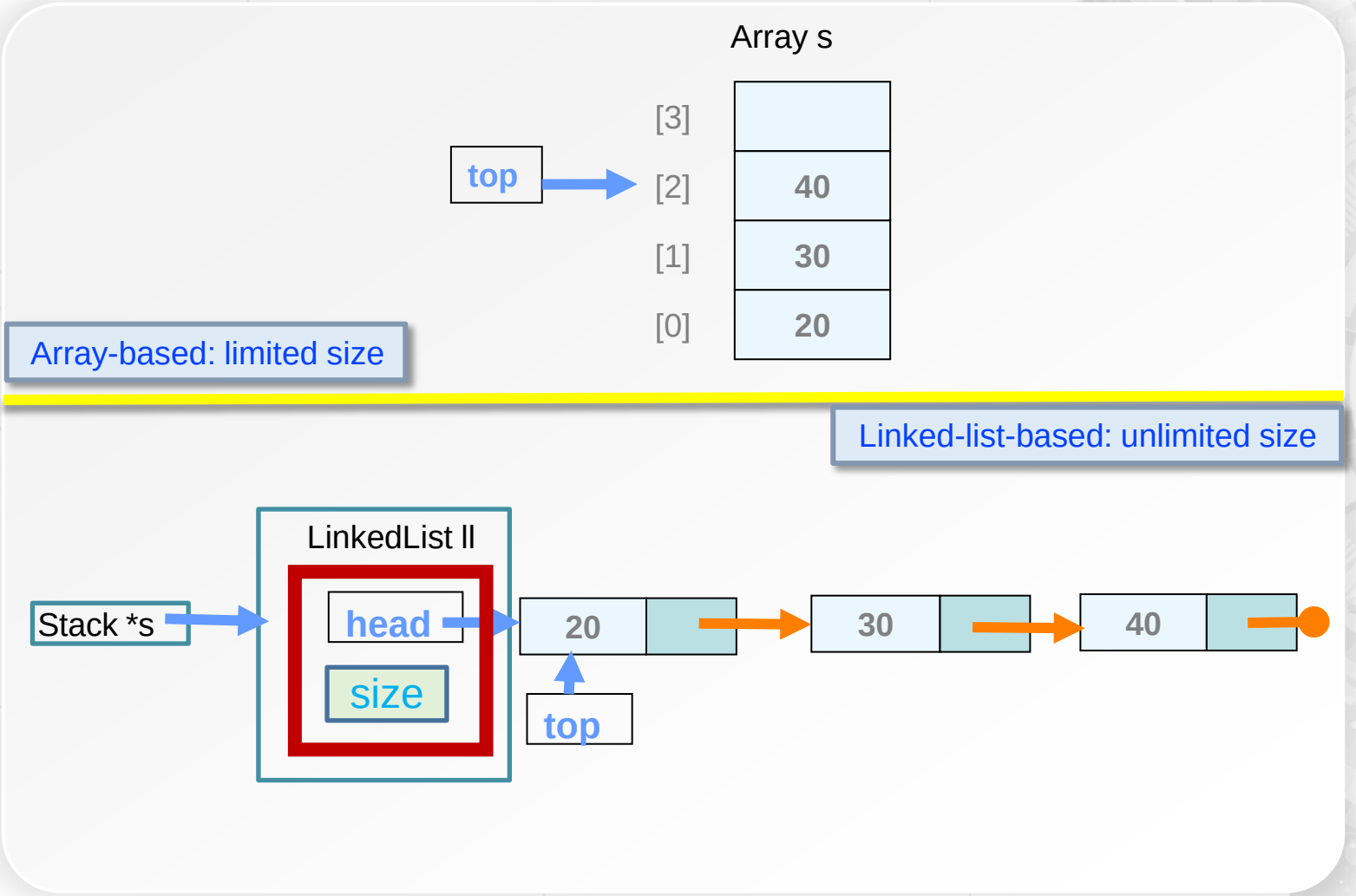
- Functions: push(), pop(), peek(), isEmptyStack(), isFullStack()

# REMARKS ON ARRAY-BASED IMPLEMENTATION

- **Easy to implement**, simple coding
- **For memory usage**
  - Save memory: If size of the stack is predetermined, no extra space for pointers.
  - Waste of memory: if we use less elements.
  - Cannot add(push) more elements than the array can hold. it has a limited capacity with a fixed array



# STACK: ARRAY-BASED VS. LINKED-LIST-BASED



# YOU SHOULD BE ABLE TO...

- Explain how a stack data structure operates
- Implement a stack using a linked list
- Choose a stack data structure when given an appropriate problem to solve