Stacks and Queues

CSE 373 - Data Structures

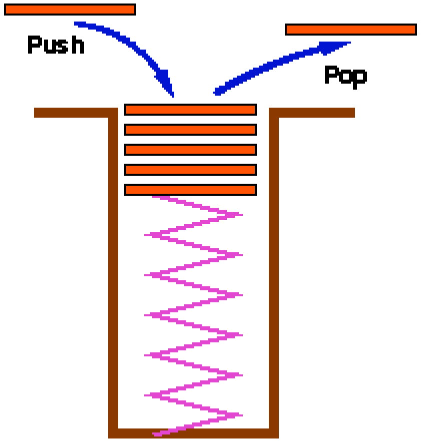
April 12, 2002

Readings and References

- Reading
  - › Section 3.3 and 3.4, *Data Structures and Algorithm Analysis in C*, Weiss
- Other References

Stacks

- A list for which Insert and Delete are allowed only at one end of the list (the *top*)
  - › the implementation defines which end is the "top"
  - › LIFO – Last in, First out
- Push: Insert element at top
- Pop: Remove and return top element (aka TopAndPop)



a tray stack

Stack ADT

`void push(Stack S, ElementType E)`

› **add an entry to the stack for E**

`ElementType pop(Stack S)`

› **remove the top entry from the stack and return it**

`Stack CreateStack(void)`

› **create a new, empty stack**

`void DestroyStack(Stack S)`

› **release all memory associated with this stack**

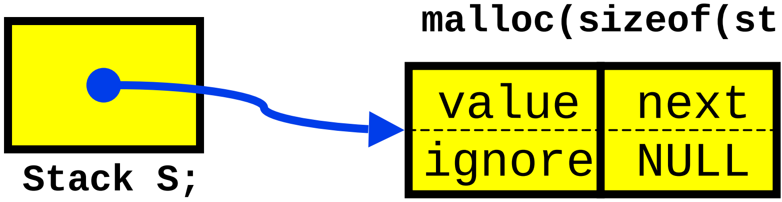
Pointer based Stack implementation

- Linked list with header
- **`typedef struct ListNode *Stack;`**
  - › "Stack" type is a pointer to a List header node
- **`S->next`** points to top of stack, the first node in the List that contains actual data
  - › the data is of type ElementType
- **`push(S,ElementType E);`**
  - › insert a new node at the start of the list

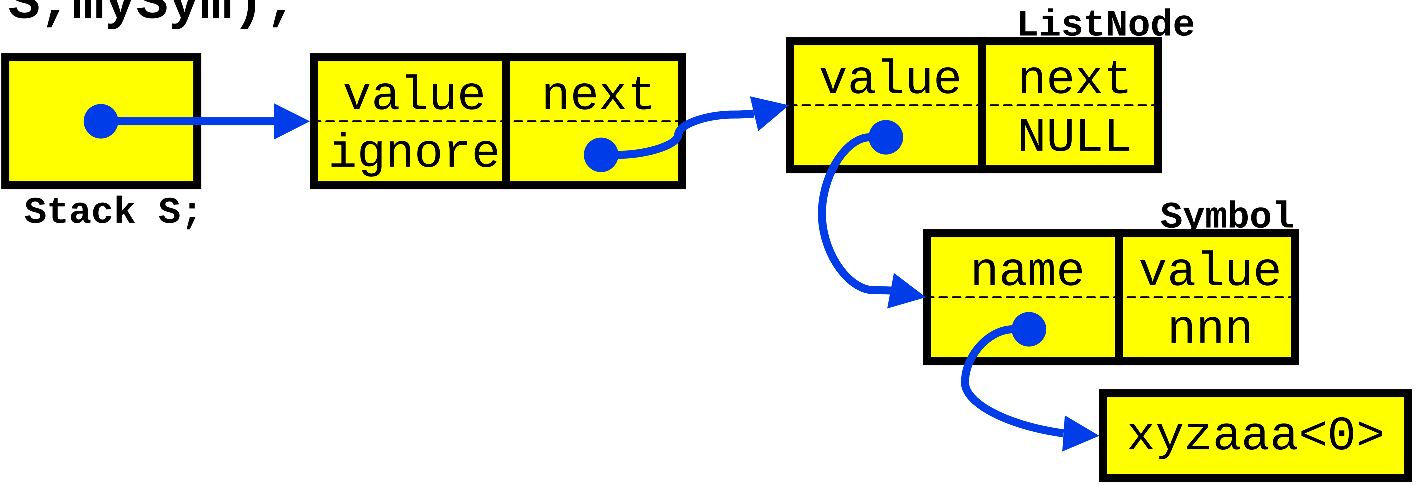
Pointer based stack elements

```
Stack S;
S = CreateStack(100);
```

`malloc(sizeof(struct ListNode));`



```
push(S,mySym);
```



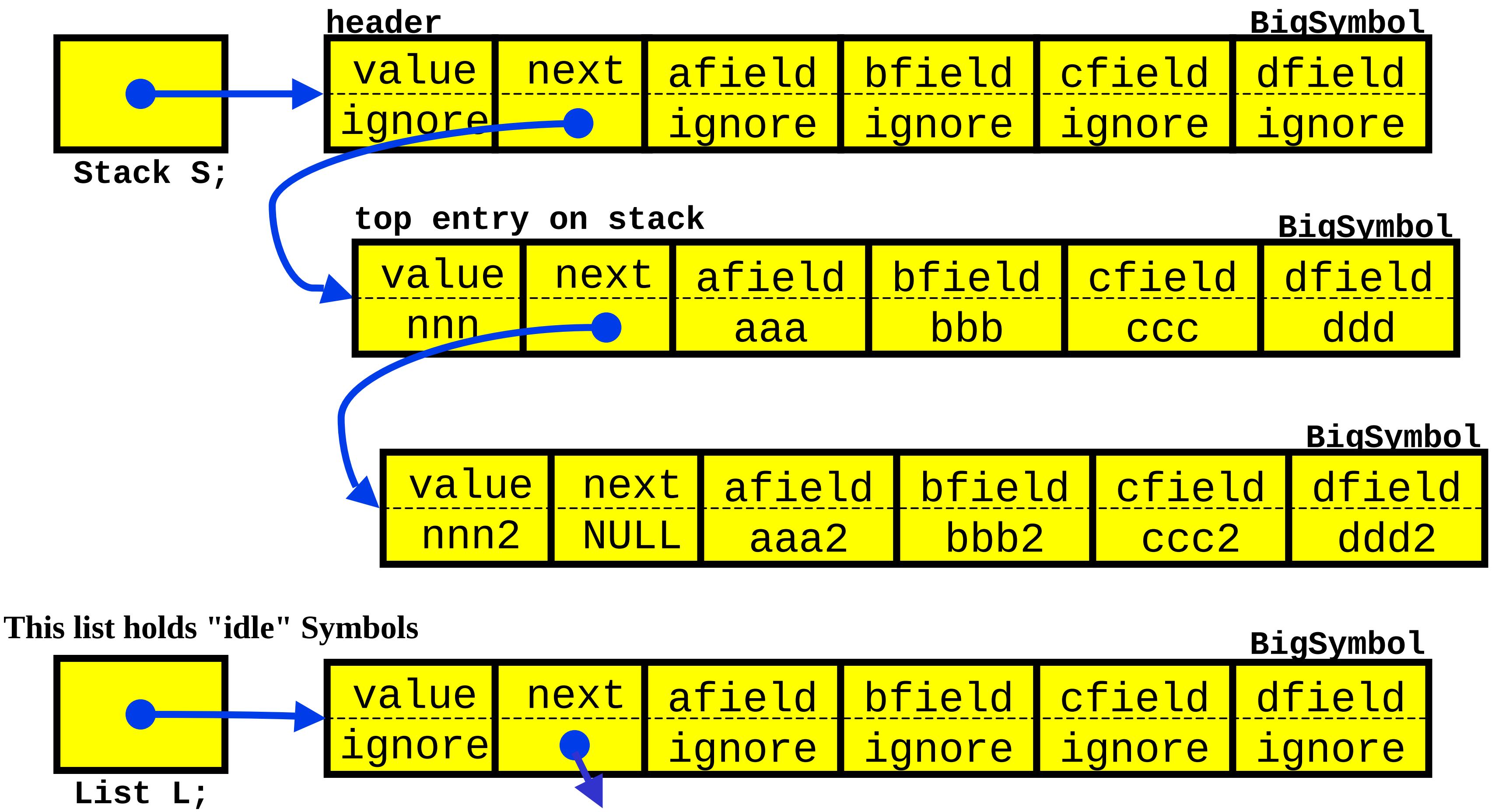
Pointer based Stack issues

- Potentially a lot of calls to malloc and free if the stack is actively used
  - › memory allocation and release require expensive trips through the operating system
- Relatively elaborate data structure for the simple push/pop functions performed
  - › overhead of ListNodes
  - › insert and delete only take place at one end

Pointer based Stack

- Under some circumstances a pointer based stack can be a good choice
- For example, assume
  - › a **`struct Symbol`** is allocated once for each symbol
  - › the symbol is used for a long time in various ways
  - › there is a **`struct Symbol *next`** in each **`struct Symbol`**
  - › then you can use the **`Symbol`** objects as list nodes and link / unlink them with no **`malloc/free`** needed

Stack with BigSymbol nodes



header

BigSymbol

value	next	afield	bfield	cfield	dfield
ignore	●	ignore	ignore	ignore	ignore

Stack S;

top entry on stack

BigSymbol

value	next	afield	bfield	cfield	dfield
nnn	●	aaa	bbb	ccc	ddd

BigSymbol

value	next	afield	bfield	cfield	dfield
nnn2	NULL	aaa2	bbb2	ccc2	ddd2

This list holds "idle" Symbols

BigSymbol

value	next	afield	bfield	cfield	dfield
ignore	●	ignore	ignore	ignore	ignore

List L;

Array based Stack implementation

- Recall the array implementation of Lists
  - › Insert and Delete took O(N) time because we needed to shift elements when operating at an arbitrary position in the list
- What if we avoid shifting by inserting and deleting only at the end of the list?
  - › Both operations take O(1) time!
- Stack: A list for which Insert and Delete are allowed only at one end of the list (the *top*)

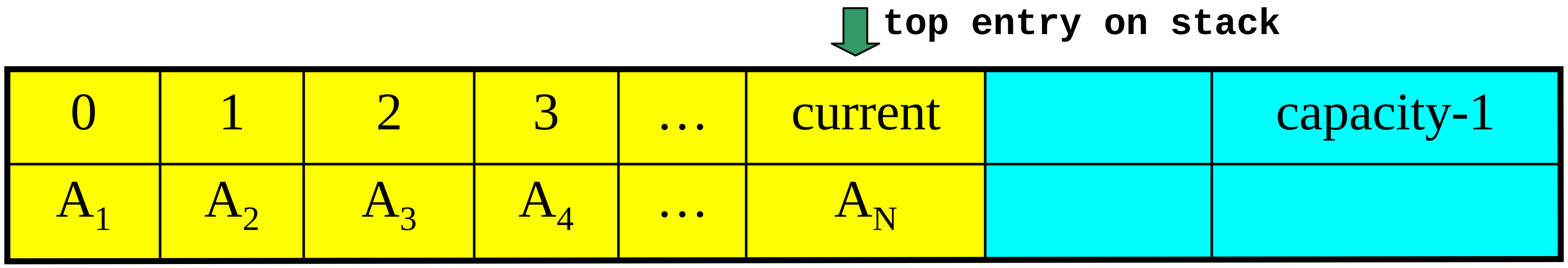
Array based Stack implementation

- An array of ElementType entries
  - › dynamically allocated array
- **`typedef struct StackRecord *Stack;`**
  - › "Stack" type is a pointer to a Stack data record
- **`S->current`** is the array index of the entry at the top of the stack
  - › the data is of type ElementType
- **`push(S,ElementType E);`**
  - › add a new entry at the end (top) of the current list

Array based Stack elements

```
struct StackRecord {
  int capacity;          /* max number of elements */
  int current;           /* offset to most recently pushed value */
  ElementType *buffer;   /* pointer to actual stack area */
};

//Empty stack has allocated array and current = -1
```

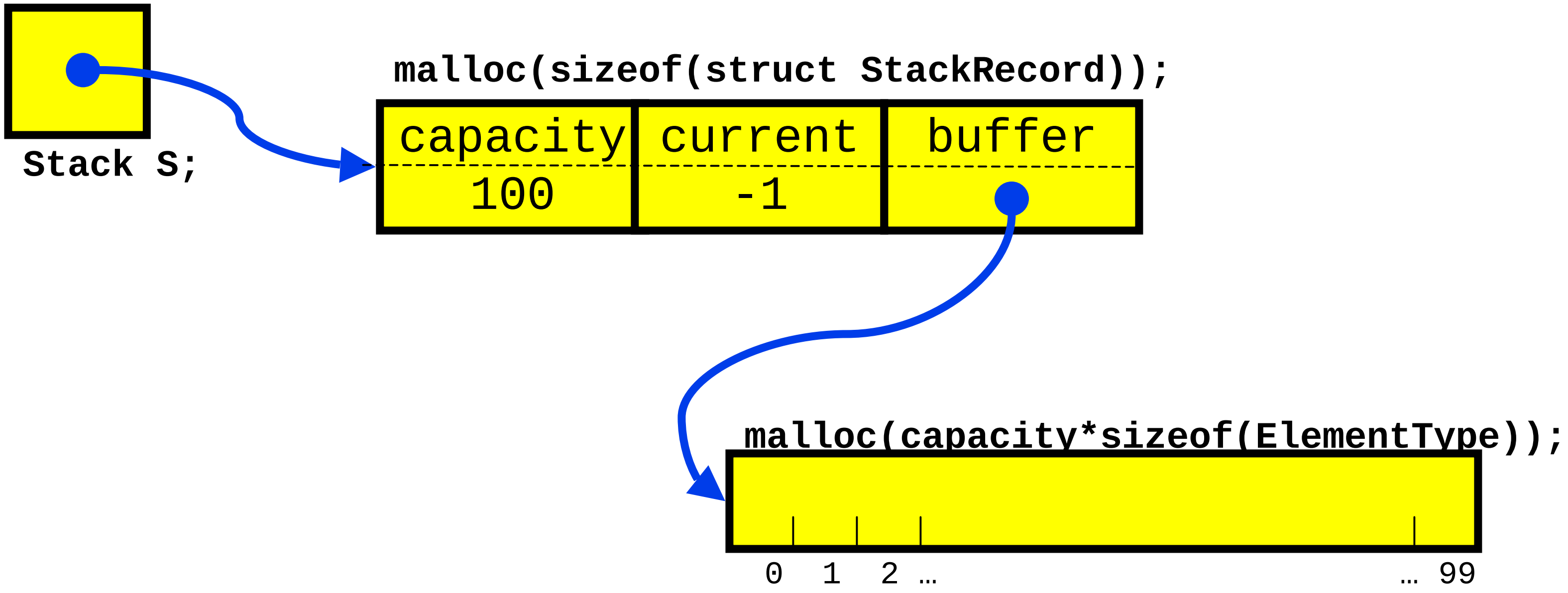


top entry on stack

0	1	2	3	…	current		capacity-1
A_1	A_2	A_3	A_4	…	A_N		

Array based stack create

```
Stack S;
S = CreateStack(100);
```



Stack S;

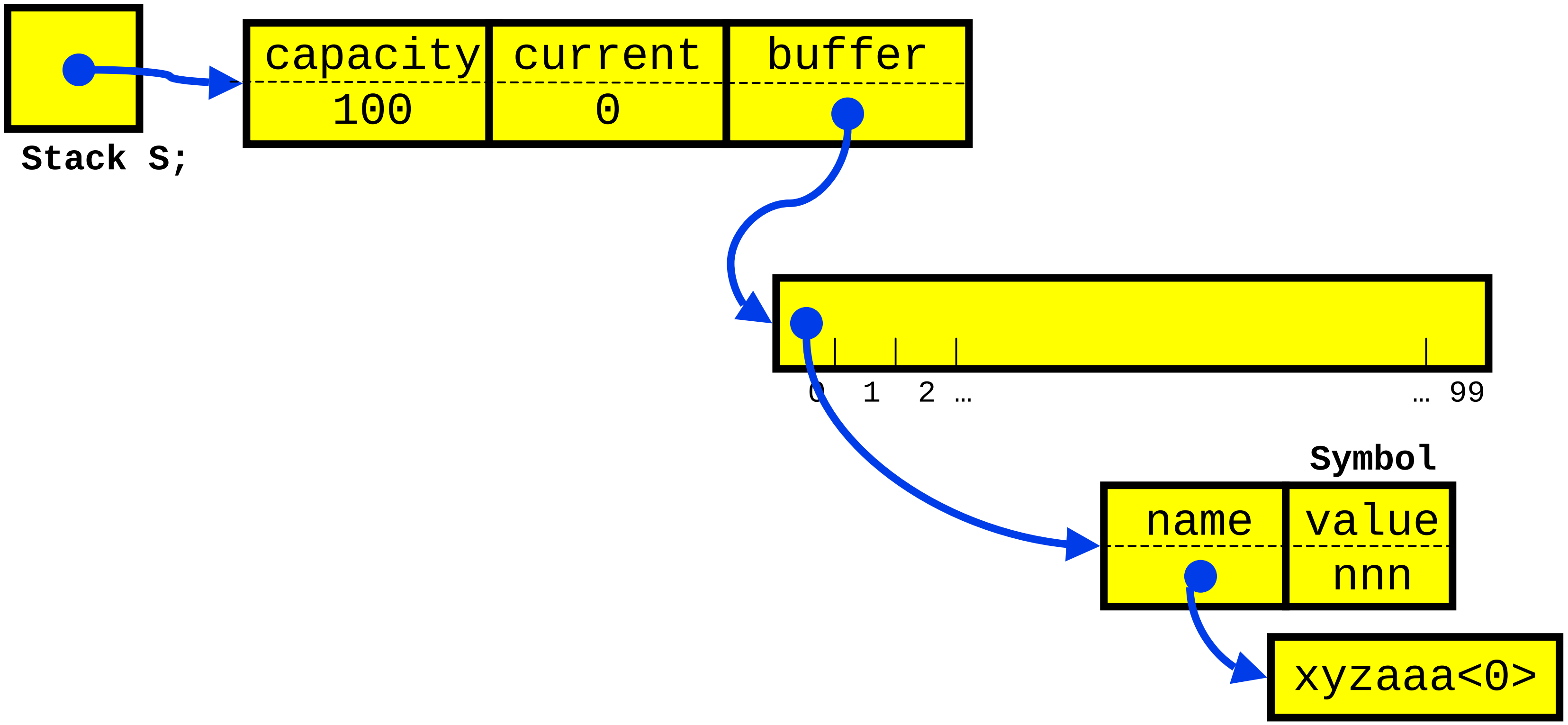
malloc(sizeof(struct StackRecord));

capacity	current	buffer
100	-1	●

malloc(capacity*sizeof(ElementType));

Array based stack push

```
push(S,mySym);
```



Array based Stack issues

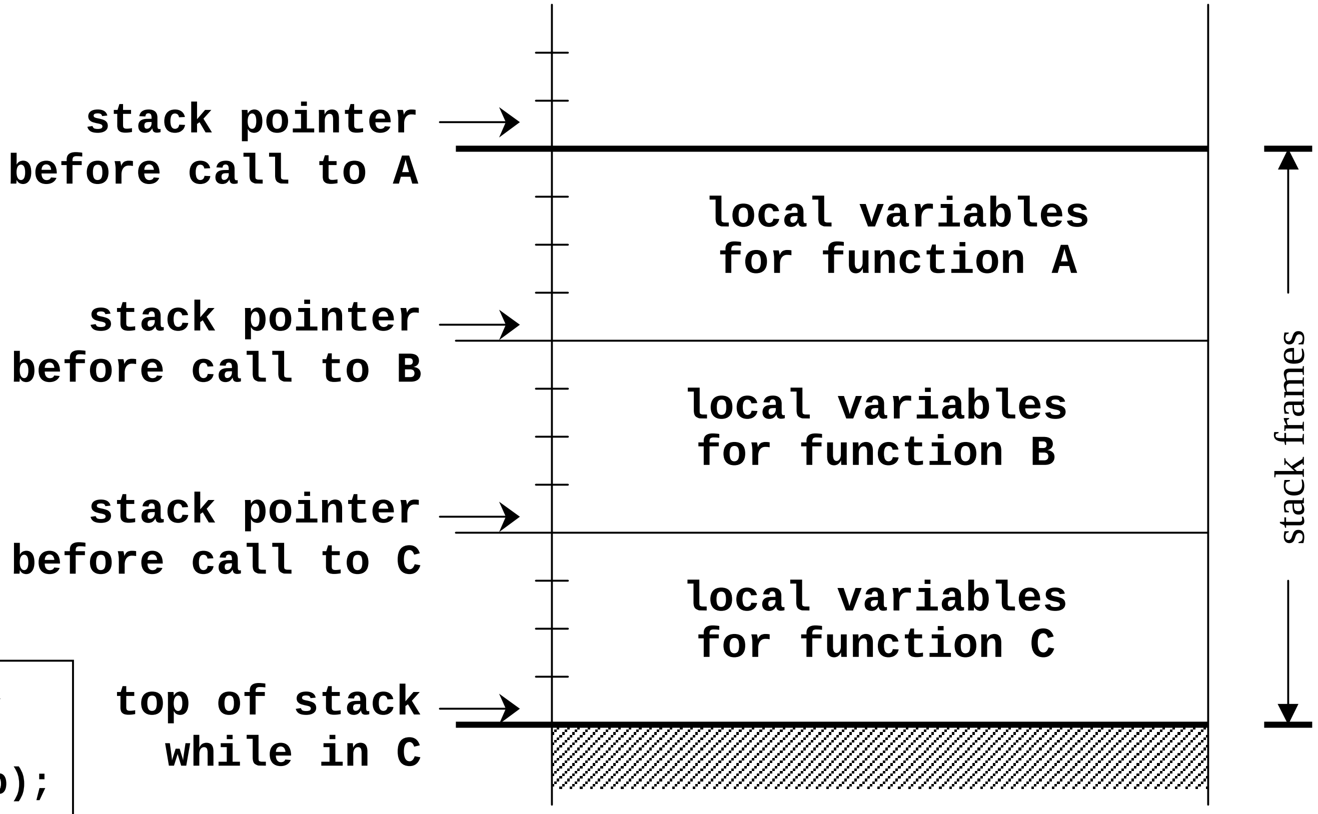
- The array that is used as the Stack must be allocated and may be too big or too small
  - › can dynamically reallocate bigger array on stack overflow
- Error checking
  - › who checks for overflow and underflow?
  - › an array based Stack is so simple that error checking can be a significant percentage cost

(i + 5*(17 – j/(6*k)) : Balanced?

- Balance Checker using Stack
  - › create an empty stack and start reading symbols
  - › If input is an opening symbol, push onto stack
  - › If input is a closing symbol
    - If stack is empty, report error
    - Else, Pop the stack

      Report error if popped symbol is not corresponding open symbol
  - › If EOF and stack is not empty, report error

Using a stack for function calls



stack pointer before call to A

stack pointer before call to B

stack pointer before call to C

top of stack while in C

local variables for function A

local variables for function B

local variables for function C

```
main: myVar = A(x);
in A: k = B(x,2);
in B: z = C(500,tmp);
in C: myVal = a+b;
```

Using a Stack for Arithmetic

- infix notation : a+b*c+(d*e+f)*g
  - › the operators are between the operands
- postfix notation: abc*+de*f+g*+
  - › the operators follow the operands
- convert to postfix using a stack
  - › read the input stream of characters
  - › output operands as they are seen
  - › push and pop operators according to priority
- evaluate postfix expression using a stack

Queue

- Insert at one end of List, remove at the other end
- Queues are “FIFO” – first in, first out
- Primary operations are Enqueue and Dequeue
- A queue ensures “fairness”
  - › customers waiting on a customer hotline
  - › processes waiting to run on the CPU

Queue ADT

- Operations:
  - › void Enqueue(Queue Q, ElementType E)
    - add an entry at the end of the queue
  - › ElementType Dequeue(Queue Q)
    - remove the entry from the beginning of the queue
    - aka ElementType FrontAndDequeue(Queue Q)
  - › int IsEmpty(Queue Q)

Queue ADT

- Pointer-based: what pointers do you need to keep track of for O(1) implementation?
- Array-based: can use List operations Insert and Delete, but O(N) time due to copying
- How can you make array-based Enqueue and Dequeue O(1) time?
  - Use Front and Rear indices: Rear incremented for Enqueue and Front incremented for Dequeue

Applications of Queues

- File servers: Users needing access to their files on a shared file server machine are given access on a FIFO basis
- Printer Queue: Jobs submitted to a printer are printed in order of arrival
- Phone calls made to customer service hotlines are usually placed in a queue