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)
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));

Stack S;

push(S,mySym);

ListNode

Symbol

value

next

NULL

value

next

NULL

value

next

NULL

name

value

nnn

xyzaaa<0>
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

Stack $S$;

List $L$;

This list holds "idle" Symbols

12-Apr-02
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,ElementDataType E);`
  - add a new entry at the end (top) of the current list
Array based Stack elements

```c
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
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>current</th>
<th>capacity-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>A_2</td>
<td>A_3</td>
<td>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>
Array based stack create

Stack S;
S = CreateStack(100);

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

```
Stack S;
Stack S;
malloc(sizeof(struct StackRecord));
capacity 100  current -1  buffer
```

```
malloc(capacity*sizeof(ElementType));
```

```
0 1 2 ... 99
```
Array based stack push

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

Stack $S$;

- capacity: 100
- current: 0
- buffer:

Symbol
- name: nnn
- value: xyzaaa<0>
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

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)
  - intIsEmpty(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