Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI2100B CSCI2100S DATA STRUCTURES

Spring 2011

Stacks & Queues

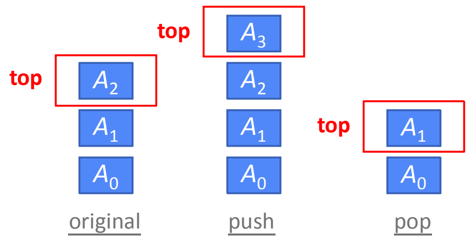
Last updated: **20/02/2011**

Tang Wai Chung, Matthew

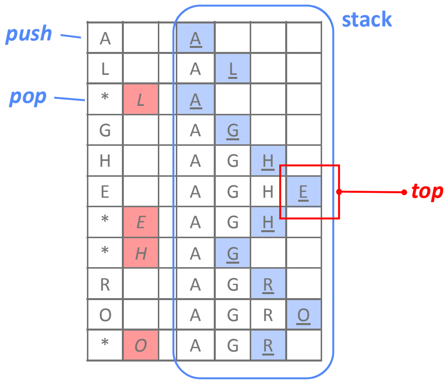
Stack ADT

- A stack is a list that can only insert and delete in only one position - **top**
  - ***Push***: insert to **top**
  - ***Pop***: delete from **top**
  - ***Top***: check the **top** without poping it.

LIFO
Last-in, first-out list



Stack ADT: Example



Stack: Array Implementation

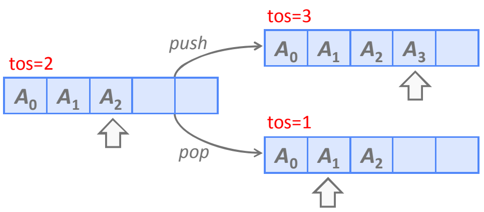
- ☑ Less pointer manipulations
  - consequently less calls to malloc() and free()
- Array implementation of stacks is a popular solution if the capacity required can be **estimated**.
- ☒ **Hazard**: need to declare an array size ahead of time.

- **Key variable**:
  tos (defined to be -1 when the stack is empty)
- To push, increment tos and assign x to stack[tos]
- To pop, return stack[tos] and decrement tos.

Stack: Array Implementation

```
struct stack_s;
typedef struct stack_s *stack_t;
```

```
#define EMPTY_TOS -1
#define MAX_SIZE 100

struct stack_s {
    int tos; /* the tos */
    int e[MAX_SIZE]; /* the data */
};
```



Stack w/ Array: *create, is_empty*

All functions run in constant time. $O(1)$

```
stack_t stack_create(void){
    stack_t s = malloc(sizeof *s);
    stack_make_empty(s);
    return s;
}

int stack_is_empty(stack_t s){
    return (s->tos == EMPTY_TOS);
}

void stack_make_empty(stack_t s){
    s->tos = EMPTY_TOS;
}
```


Stack w/ Array: *push, top & pop*

Pay attention to the error checking

```
void stack_push(stack_t s, int x){
    if (s->tos >= MAX_SIZE){
        perror("stack is full.\n");
        exit(1);
    }
    s->e[++s->tos] = x;
}

int stack_top(stack_t s){
    if (!stack_is_empty(s))
        return (s->e[s->tos]);
    return INT_MIN;
}

void stack_pop(stack_t s){
    s->tos--;
}
```


Stack w/ Array: *topandpop, free*

- Array implementation of stack also preserves constant time ***push*** and ***pop***.
- Very often we want to ***pop*** and get the element on the top. Then you can use ***topandpop***.
- Don't forget to free a stack when you finish using the stack.

```
int stack_topandpop(stack_t s){
    if (!stack_is_empty(s))
        return (s->e[s->tos--]);
    return INT_MIN;
}

void stack_free(stack_t s){
    free(s);
}
```

Stack: Linked List Implementation

- ***Push***: insert at the front of the list. $O(1)$
- ***Pop***: delete at the front of the list. $O(1)$
- ***Top***: examines the element at the front. $O(1)$
- Sometimes we may combine ***pop*** and ***top***.
- Keep a **header** node for easy coding.

Structure Declaration

```
struct stack_s;
typedef struct stack_s *stack_t;
```

```
typedef struct stack_s node;
struct stack_s {
    int e;
    node *next;
};
```


Stack w/ LL: Creation

- ***create***: allocates a header node to point its next to NULL.
- ***is_empty***: checks whether the header node points to a NULL.
- ***make_empty***: uses a loop to pop all elements until it becomes empty.

```
stack_t stack_create(void){
    node *s = malloc(sizeof(node));
    s->next = NULL;
    s->e = INT_MIN;
    return s;
}

int stack_is_empty(stack_t s){
    return (s->next == NULL);
}

void stack_make_empty(stack_t s){
    if (s == NULL) return;

    while (!stack_is_empty(s))
        stack_pop(s);
}
```


Stack w/ LL: *push*, *top* & *pop*

A good revision on the linked list operations

```
void stack_push(stack_t s, int x){
    node *t = malloc(sizeof(node));
    t->e = x;
    t->next = s->next;
    s->next = t;
}
```

```
int stack_top(stack_t s){
    if (!stack_is_empty(s))
        return (s->next->e);
    return INT_MIN; /* raise warning */
}
```

```
void stack_pop(stack_t s){
    node *t = s->next;
    if (!stack_is_empty(s)){
        s->next = t->next;
        free(t);
    }
}
```


Application 1: Balancing Symbols

- Compilers check for syntax errors, but frequently a lack of one symbol will cause it to spill out hundreds of lines of warning/errors.
- A useful tool is to check whether everything is **balanced**: every brace, bracket and parenthesis e.g. **([ ])** is legal but not **[( ])**
- We can use a stack to help checking:
  - If a character is an opening symbol, **push** it onto the stack.
  - If it is a closing, **pop** the stack and check if it matches to top.

Application 2: Postfix Evaluation

- We can also use stacks to evaluate arithmetic expressions.
- For instance, we need to find the value of a simple arithmetic expression with multiplications and additions of integers.

5 * (((9 + 8) * (4 * 6)) + 7)

- This involves saving **intermediate** results.
- Let's begin with a simpler problem:
  Evaluate the expression in a form where each operator appears **after** its two arguments, rather then between them.

Prefix, Infix & Postfix Expressions

Infix expression:
5 * (((9 + 8) * (4 * 6)) + 7)

Postfix expression: **5 9 8 + 4 6 * * 7 + ***

Prefix expression: *** 5 + * + 9 8 * 4 6 7**

We need **parentheses** in infix expressions to avoid ambiguity:

5 * (((9 + 8) * (4 * 6)) + 7)

vs

((5 * 9) + 8 * ((4 * 6) + 7)

Evaluating Postfix Expressions

☑ Parentheses are **not** necessary in postfix expressions.

Input	Stack			
5	5			
9	5	9		
8	5	9	8	
+	5	17		
4	5	17	4	
6	5	17	4	6
*	5	17	24	
*	5	408		
7	5	408	7	
+	5	415		
*	2075			

tos

Evaluation of postfix exp.

1. move from left to right
2. meet operand:
   push operand onto the stack
3. meet operator:
   pop 2 operands, perform, push result

Evaluating Postfix: Code

```
int main(int argc, char *argv[]){
    ...
    for (i = 0; i < strlen(s); i++){
        if (s[i] == '+'){
            a = stack_topandpop(stack);
            b = stack_topandpop(stack);
            stack_push(stack, a + b);
        }
        if (s[i] == '*'){
            a = stack_topandpop(stack);
            b = stack_topandpop(stack);
            stack_push(stack, a * b);
        }
        if (s[i] >= '0' && s[i] <= '9'){
            stack_push(stack, 0);
        }
        while (s[i] >= '0' && s[i] <= '9'){
            a = stack_topandpop(stack);
            stack_push(stack, 10 * a + (s[i++] - '0'));
        }
    }
    printf("%d\n", stack_top(stack));
    ...
}
```

Input	Stack	
1	0	
	0*10+1=1	
2	1*10+2=12	
	12	
3	12	0
	12	0*10+3=3
5	12	3*10+5=35
7	12	35*10+7=357
	12	357
+	369	

The while loop computes the integral value of the given string like the common utility function atoi().

Turning Infix into Postfix

- How to use a stack to convert a full parenthesized infix expression into a postfix one?

Infix to Postfix

1. move from left to right
2. meet operand: pass
3. meet operator: push
4. meet right parenthesis: pop

Input	Output	Stack	
(			
(			
5	5		
*		*	
(		*	
6	6	*	
+		*	+
2	2	*	+
)	+	*	
)	*		
+		+	
3	3	+	
)	+		

Queue ADT

- A queue is a **list** that insert at the end (called the rear/tail) and delete at the start (front/head).
  - **First in, first out: FIFO**
- ***Enqueue***(put): inserts an element at the end
- ***Dequeue***(get): deletes (and returns) the element at the start

Queue ADT: Example

		queue			
A		A			
L		A	L		
*	A	L			
G		L	G		
H		L	G	H	
E		L	G	H	E
*	L	G	H	E	
*	G	H	E		
R		H	E	R	
O		H	E	R	O
*	H				

enqueue *dequeue* *rear* *front*

Queue ADT

```c
struct queue_s;
typedef struct queue_s *queue_t;

queue_t queue_create(void);
void queue_free(queue_t q);
void queue_make_empty(queue_t q);
int queue_size(queue_t q);
int queue_is_empty(queue_t q);
int queue_is_full(queue_t q);
void queue_enq(queue_t q, int x);
int queue_deq(queue_t q);
void queue_print(queue_t q);
```

We can implement the queue ADT using arrays or linked lists.
Arrays: constant time operations, **fixed** maximum queue size.
Linked lists: constant time operations, **more flexible** size.

Array Implementation of Queues

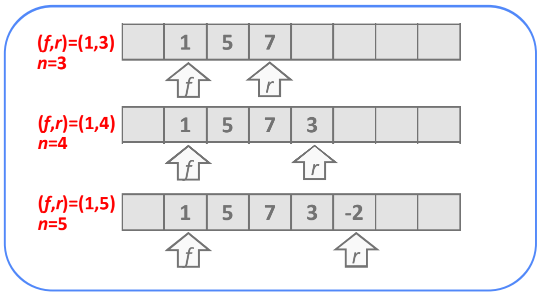
- For each queue, we keep an array e[], and the positions f and r, referring to the front and rear respectively.
- We also keep track of the number of elements in the queue with variable n.

Structure Declaration

```c
#define MAX_SIZE 100

struct queue_s {
    int e[MAX_SIZE]; /* data */
    int f; /* front */
    int r; /* rear */
    int n; /* size */
};
```

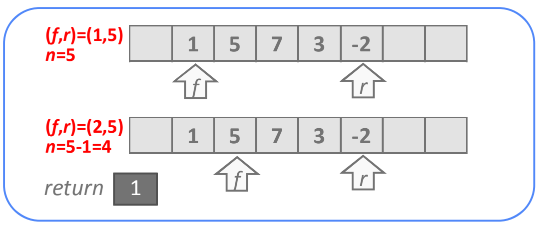

Queue w/ Array: Enqueue

- Increment size and rear
- Set queue[rear] = x

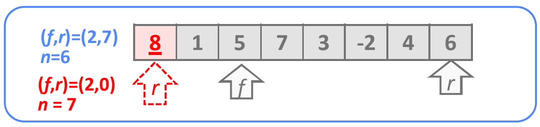


Queue w/ Array: Dequeue

- Save the return value: element pointed by front.
- Then **decrement** n and **advance** front



Queue w/ Array: Circular Array



- After a number of ***enqueue*** and ***dequeue***, the queue appears to be full.
- **Solution**: wrap around front and rear whenever it gets to the end of the array.
  - This is called the **circular** array implementation.
- In C, we may use the **modulus** operator (%).

Queue w/ Array: Coding (1)

```c
void queue_enq(queue_t q, int x){
    if (queue_is_full(q)){
        perror("The queue is full. Cannot enqueue more. exit.\n");
        exit(1);
    }
    q->n++;
    q->r = (q->r + 1) % MAX_SIZE;
    q->e[q->r] = x;
}

int queue_deq(queue_t q){
    int x;

    if (queue_is_empty(q))
        return INT_MIN;

    q->n--;
    x = q->e[q->f];
    q->f = (q->f + 1) % MAX_SIZE;

    return x;
}
```

raise error when the queue is full

raise error when the queue is empty

Queue w/ Array: Coding (2)

What are the complexities of the following operations?

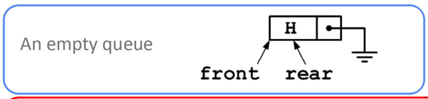
```c
queue_t queue_create(void){
    queue_t q = malloc(sizeof *q);
    queue_make_empty(q);

    return q;
}

void queue_free(queue_t q){
    free(q);
}

void queue_make_empty(queue_t q){
    q->n = 0;
    q->f = 1;
    q->r = 0;
}
```

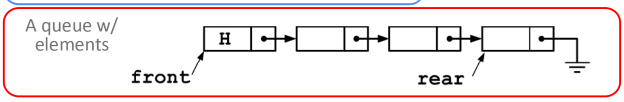

Linked List Implm. of Queue

- The array implementation is already very good in terms of running time.
  - ☺ All common operations are *O*(1).
- ☹ Still the main disadvantage is that we **cannot** handle **arbitrarily** long queues as the size of the array is **fixed** after allocation.
- The linked list implementation is more **versatile**, however, it requires some tricks in the implementation.
- We adapt the **header pointer and null tail** convention in the following implementation.

Queue w/ LL: front & rear ptr

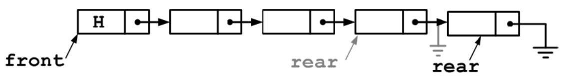
- The front and rear of the queue is maintained by the pointers to the nodes on the linked list.
- front: equivalent to header node.
  rear: the node with the null link
- *is_empty*: check if rear points to the header node. *O*(1)
- Note: you have the flexibility to store the size of the queue.



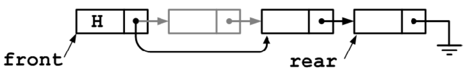


Queue w/ LL: *enqueue* & *dequeue*

enqueue: insert a new node after the node pointed by rear; update rear. *O*(1)



dequeue: delete the node pointed by the header node (which equals front). Be careful when you have 1 last element in the queue. *O*(1)



Summary

- Stack ADT: LIFO
  Insert and delete from the **same** end of the list
  - Array implementation
  - Linked Listed Implementation
  - Application: Evaluation of arithmetic expressions
- Queue ADT: FIFO
  Insert from one end and delete from another.
  - Circular Array implementation
  - Linked Listed Implementation