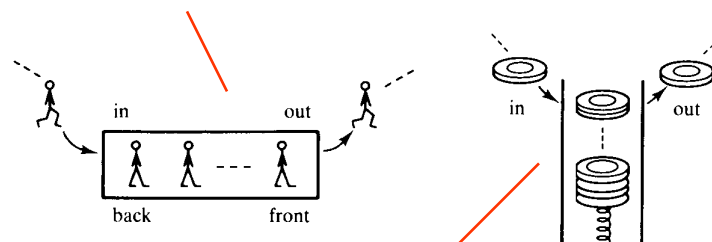


Topic 3: Stacks and queues

1. Stack operations
2. Stack examples
3. Stack implementations
4. Queue operations
5. Applications for queues
6. Queue implementations

Stacks and queues

- Specialized collections with restricted access
- A queue works on first-in, first-out principle



- A stack is last-in, first-out (LIFO)

[Diagrams: D. Harel, *The Science of Computing*]

1. Stack operations

- Initialize
- See if empty
- See if full*
- Push
- Pop

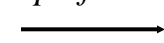
** A stack may be full in array implementation; is virtually unbounded in linked-list implementation*

Using a stack to reverse a list

```

void main()
{
    stacks stack;
    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);
    while (! stack.is_empty())
        cout << stack.pop() << endl;
}

```

top of stack


1
1 2
1 2 3
1 2 3 4
1 2 3 4
1 2 3
1 2
1

Output: 4 3 2 1

Preconditions, postconditions and invariants for stacks

- **stacks()** *Postcondition:* stack exists, stack is empty
Invariant: $0 \leq size \leq MAX_SIZE$ (array implementation)
- **void push(T n)**
Preconditions: $size < MAX_SIZE$; n is value to insert
Postcondition: n is at the top of stack, $size$ is 1 greater
- **T pop()**
Preconditions: $size > 0$
Postcondition: stack is one item smaller than before *pop*
- **bool is_empty()** *Postcondition:* returns *true* iff $size = 0$
- **bool is_full()** *Postcondition:* returns *true* iff stack at capacity (e.g., array)

Adapted from Ford/Topp text

2. Stack examples

- Activation records for method calls
- Reverse Polish Notation
- Delimiter balancing

Why activation records use a stack structure

- Program must manage local data of all currently executing methods
- Local variable and parameter space must be freed when a subprogram terminates
- *Result:* a method call triggers *push*, termination triggers *pop*

Application for stack: checking delimiter balance

balance1.dat:

```
{  
  a =  
    (b[1] + c[2])  
    * d;  
}
```

balance2.dat:

```
{  
  a =  
    (b[1] + c(2))  
    * d;  
}
```

balance3.dat:

```
{  
  a = ])  
}
```

- To determine:
are all braces,
parentheses, brackets
correctly balanced?

Using stack to check delimiters....

```

main()
{
    stacks history;
    int line_num = 1;
    bool error_found = false;
    FILE* infile = open_input();
    stack_init(&history);
    while (! feof(infile) && ! error_found){
        char input = fgetc(infile);
        putchar(input);
        if (input == '\n') ++line_num;
        if (input == '(' || input == '{' || input == '[')
            stack_push(&history,input);
        if (input == ')' || input == '}' || input == ']')
            if (! stack_is_empty(history))
                { if (input != mate_of(stack_pop(&history)))
                    error_found = true; // Mismatched delimiter }
                else // if stack empty when right delimiter found:
                    error_found = true; // Unexpected delimiter
            }
    }
}

```

Rule: push left delimiter, pop on reading right delimiter and compare with expected character

[from balance.c]

6/08

David Keil

Data Structures 3. Stacks and queues

9

... stack should be empty at end

```

// Detect unmatched left delimiter remaining on stack:
while (feof(infile) && ! stack_is_empty(history))
{
    printf("\nNo match found in file for %c\n",
        stack_pop(&history));
    error_found = true;
}
if (! error_found)
    printf("Balanced parens, brackets, braces.\n");
}

```

```

char mate_of(char delimiter)
// Returns complement to a
// left-delimiter character
{
    switch(delimiter)
    {
        case '(': return ')';
        case '[': return ']';
        case '{': return '}';
    }
    return '\0';
}

```

6/08

David Keil

Data Structures 3. Stacks and queues

10

Reverse Polish notation and stack

- Postfix notation places operator after operands
- Example: $2\ 3\ *\ 4\ +$ (for infix $2\ *\ 3\ +\ 4$)
- Steps to evaluate $2\ 3\ *\ 4\ +$:
 1. Read 2 , push 2
 2. Read 3 , push 3
 3. Read $*$, pop 3 , pop 2 , mult. $2\ *\ 3$, push 6
 4. Read 4 , push 4
 5. Read $+$, pop 4 , pop 6 , add $6\ +\ 4$, push 10
- Why calculate this way? Easier to code than with infix notation

Evaluate RPN expression

1. Repeat until input stream exhausted
2. Read operator or numeral from input
3. If input is numeral
4. Push on stack
5. If input is operator
6. Pop stack twice
7. Apply operator to 2 values, 2nd first
8. Push result on stack
9. Pop stack
10. Display result
11. If stack not empty
12. Report "Insufficient operators"

Running time for n input tokens: $O(n)$

RPN example: 3 5 + 1 -

Algorithm step	Operation	State of stack (top→)
2	Read 3	
3,4	Push 3	3
2	Read 5	3
3,4	Push 5	3 5
2	Read +	3 5
6	Pop 5	3
6	Pop 3	
7	Add 3, 5 yielding 8	
8	Push 8	8
2	Read 1	8
3,4	Push 1	8 1
2	Read -	8 1
6	Pop 1	8
6	Pop 8	
7	Subtract 1 from 8 yielding 7	
8	Push 7	7
9	Pop 7 at end of input	
10	Display 7 as result (3 + 5 - 1)	
11	Verify empty stack	

6/08

David Keil

Data Structures 3. Stacks and queues

13

RPN group work problem hints

```
#include <ctype.h>
#include <stdlib.h>
char input[80];
do {
    cin >> input;
    if (isdigit(input[0])) // numeric
        int n = atoi(input);
    else // determine operator...
        ...
}
```

At numeric input, push number;

At operator input, pop two numbers

6/08

David Keil

Data Structures 3. Stacks and queues

14

Iterative Quicksort using stack class

```
void iterative_quick(int A[],int size)
{
    int first=0,last=size; // Subscripts to <A>
    stacks left_pending,right_pending;
    do {
        if (! left_pending.is_empty())
        {
            first = left_pending.pop();
            last = right_pending.pop();
        }
        while (first < last)
        {
            int pivotloc = partition(A,first,last);
            left_pending.push(pivotloc + 1);
            right_pending.push(last);
            last = pivotloc - 1;
        }
    } while (! left_pending.is_empty());
}
```

6/08

David Keil

Data Structures 3. Stacks and queues

17

3. Stack implementations

- Array
- Linked list

6/08

David Keil

Data Structures 3. Stacks and queues

18

An integer-stack class using array

```

const int MAX_STK_SZ = 20;

class stacks
{
public:
    stacks() { size = 0; };
    void push(int n) {
        if (! is_full()) element[size++] = n; };
    int pop(){
        assert(! is_empty());
        return element[--size];
    };
    bool is_empty() { return (size == 0); };
    bool is_full() { return (size >= MAX_STK_SZ);};
private:
    int element[MAX_STACK_SZ];
    int size;
};

```

(For usage, see earlier slide "Using a stack to reverse a list")

6/08

David Keil

[intStack.cpp]

3. Stacks and queues

19

C linked-list-based stack of characters

```

struct Nodes
{
    char data;
    nodes *next;
};
typedef struct Nodes nodes;

struct Lists
{
    nodes header;
};
typedef struct Lists lists;

struct Stacks
{
    lists list;
};
typedef struct Stacks stacks;

void stack_init(stacks* s);
void stack_push(stacks* s, char c);
char stack_pop(stacks* s);
boolean stack_is_empty(stacks s);
void stack_display(stacks s);

```

[balance.cpp]

6/08

David Keil

Data Structures 3. Stacks and queues

20

Operations on *char* stack

```

char stack_pop(stacks* p_stack)
{
    lists* p_list = &(p_stack->list);
    char ch;
    if (! stack_is_empty(*p_stack))
    {
        ch = p_list->header.next->data;
        list_delete(&(p_list->header));
        return ch;
    }
    else
        return '\0';
}

void stack_push(stacks* p_stack, char ch)
{
    list_prepend(&(p_stack->list), ch);
};

```

See what's at front of list (top of stack)

Delete front node from list:

[balance.c]

6/08

David Keil

Data Structures 3. Stacks and queues

21

Linked-list implementation

- Update operations (*push*, *pop*) are at front, with constant-time access
- Stack is never full unless system is out of heap memory
- *Options:*
 1. Hand code linked-list operations as stack member functions
 2. Let stack inherit from linked list class
 3. Make list object a member of stack class
 4. Make pointer to top a member of stack ADT
- For a generic stack, use stack or list template

6/08

David Keil

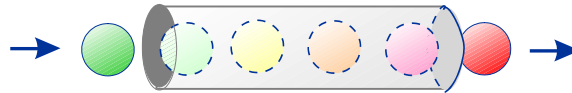
Data Structures 3. Stacks and queues

22

4. Queue operations

- Initialize
- See if empty
- See if full*
- Enqueue
- Dequeue

5. Applications for queues



- Maintain chronological order of print jobs or transactions while waiting for one to complete
- Simulate retail or bank operation
- In general, to store and update a series of items that must be retrieved in chronological order by arrival time

Checkout-line simulation using queue

Q is a queue object

1. `Q.enqueue("Bill");*`
2. `Q.enqueue("Jan");`
3. `Q.enqueue("Tina");`
4. `cout << Q.dequeue();`
5. `Q.enqueue("Val");`
6. `cout << Q.dequeue();`
7. `cout << Q.dequeue();`

The diagram illustrates the state of a queue after each operation. The queue is represented as a vertical array of slots. The 'Rear' pointer points to the slot where the next element will be added, and the 'Front' pointer points to the slot from which the next element will be removed. Elements are shown in boxes with a checkmark, indicating they are in the queue. Elements that have been removed are shown in boxes with a slash, indicating they are no longer in the queue.

6/08 David Keil Data Structures 3. Stacks and queues 25

Using a queue to match a pattern

- We may wish to search an input stream for a *pattern*
- A pattern may contain unknowns
- *Example:* Using '?' for unknown-character, "q???e" matches "quake", "quite", "queue"
- *One solution:* a circular queue that stores the most recent 10 characters

[See *searchq.c*]

Sample pattern-match I/O

Enter a stream of characters terminated
by *: T

Th
The
The
The q
The qu
The qui
The quie
The quiet
The quiet
he quiet q
e quiet qu
 quiet que
quiet queu
uquiet queue
iet queuee
iet queuee
et queuee i
t queuee is

```
queue is
queue is q
**** matches q???e ****
ueue is qu
eue is qui
ue is quit
e is quite
is quite
is quite q
s quite qu
quite qui
quite quic
**** matches q???e ****
uite quick
```

6/08

David Keil

Data Structures 3. Stacks and queues

27

Pattern matching

```
queues buffer;
q_init(&buffer);
...
do {
    input = getch();
    if (input != STOP)
    {
        q_enqueue(&buffer, input);
        display_queue(buffer);
        if (compare_queue(buffer, PATTERN))
        {
            display_queue(buffer);
            printf("*** matches %s **\n", PATTERN);
        }
    }
} while (input != STOP);
```

[searchq.c]

6/08

David Keil

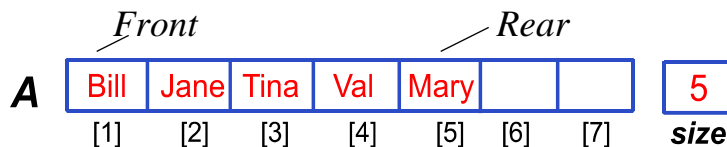
Data Structures 3. Stacks and queues

28

6. Queue implementations

- Array
- Linked list

Array-based queue



- To enqueue:
 If $size \leq capacity$
 $A[size] \leftarrow new_value;$ $O(\text{---})$
 increment $size$
- To dequeue:
 If $size > 0$, $front \leftarrow A[1]$ $O(\text{---})$
 Decrement $size$;
 For $i \leftarrow 1$ to $size$, $A[i] \leftarrow A[i+1];$
 Return $front$

Circular-array queue code

```
#define MAX_Q_SIZE 10

struct Queues
{
    char element[MAX_Q_SIZE];
    int first,
        size;
};
typedef struct Queues queues;

/* Standard queue interface: */
void q_init(queues* q);
void q_enqueue(queues* q, char c);
char q_dequeue(queues* q);
bool q_is_empty(queues q);
bool q_is_full(queues q);
```

Queues using linked lists

- Accessing and deleting first node is $O(1)$
- Enqueuing new node requires append, may involve traversal ($O(n)$)
- Linked list that stores a pointer to rear list node makes *enqueue* a $O(1)$ operation
- Doubly-linked list does same

Queue class based on linked list

```

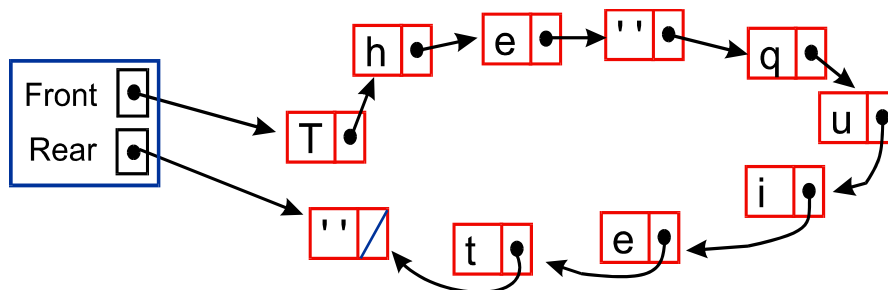
class str_queues {
private:
    str_lists list;
    list_nodes* rear;
public:
    str_queues() { rear = NULL; };
    void enqueue(char* s)
        { rear = list.insert(s,rear); };
    char* dequeue()
        { char* s = list.header.next->data();
          list.delete(header); return s; };
    bool is_empty();
};

```

Assumptions:

- Classes *list_nodes* and *str_lists* defined
- *str_lists* has functions *insert* and *delete* that accept, return appropriate pointers

Ring-buffer queue as linked list



- *Enqueue* inserts a new node after the one pointed to by *Rear*
- *Dequeue* retrieves and deletes node pointed to by *Front*

Terms

array implementation of queue	linked-list implementation of
array implementation of stack	stack, queue
circular array	pop
circular list	push
dequeue	queue
enqueue	restricted access
FIFO	reverse Polish notation
<i>is_empty</i>	ring buffer
<i>is_full</i>	stack
LIFO	top of stack