3. Stacks and queues

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)

### 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

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

**Output:** 4 3 2 1
Preconditions, postconditions and invariants for stacks

• **stacks()**  
  **Postcondition**: stack exists, stack is empty  
  **Invariant**: \(0 \leq \text{size} \leq \text{MAX\_SIZE}\) (array implementation)

• **void push(T n)**  
  **Preconditions**: \(\text{size} < \text{MAX\_SIZE}\); \(n\) is value to insert  
  **Postcondition**: \(n\) is at the top of stack, \(\text{size}\) is 1 greater

• **T pop()**  
  **Preconditions**: \(\text{size} > 0\)  
  **Postcondition**: stack is one item smaller than before \(\text{pop}\)

• **bool is_empty()**  
  **Postcondition**: returns \(true\) iff \(\text{size} = 0\)

• **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 
    }
    ...
    char mate_of(char delimiter)
    // Returns complement to a
    // left-delimiter character
    {
        switch(delimiter)
        {
            case '(': return ')';
            case '[': return ']';
            case '{': return '}';
        }
        return '\0';
    }
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
4. If input is operator
   5. Pop stack twice
   6. Apply operator to 2 values, 2nd first
   7. Push result on stack
5. Pop stack
6. Display result
7. If stack not empty
   8. Report “Insufficient operators”

Running time for \(n\) input tokens: \(O(\quad)\)
RPN example: $3 \ 5 \ + \ 1 -$  

<table>
<thead>
<tr>
<th>Algorithm step</th>
<th>Operation</th>
<th>State of stack (top→)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Read 3</td>
<td>3</td>
</tr>
<tr>
<td>3, 4</td>
<td>Push 3</td>
<td>3, 5</td>
</tr>
<tr>
<td>2</td>
<td>Read 5</td>
<td>3, 5</td>
</tr>
<tr>
<td>3, 4</td>
<td>Push 5</td>
<td>3, 5</td>
</tr>
<tr>
<td>2</td>
<td>Read +</td>
<td>3, 5</td>
</tr>
<tr>
<td>6</td>
<td>Pop 5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Pop 3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Add 3, 5 yielding 8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>Push 8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Read 1</td>
<td>8</td>
</tr>
<tr>
<td>3, 4</td>
<td>Push 1</td>
<td>8, 1</td>
</tr>
<tr>
<td>2</td>
<td>Read −</td>
<td>8, 1</td>
</tr>
<tr>
<td>6</td>
<td>Pop 1</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Pop 8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Subtract 1 from 8 yielding 7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>Push 7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>Pop 7 at end of input</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Display 7 as result $(3 + 5 - 1)$</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Verify empty stack</td>
<td></td>
</tr>
</tbody>
</table>

RPN group work problem hints

```c
#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
```
3. Stacks and queues

Stacks in solution-space searches

- A stack can store the location of last fork in a search tree (backtracking)
- *Examples:* Chess, tic-tac-toe, artificial intelligence applications

---

**Search-stack** \((S, key)\)

*[Preconditions:]* \(S\) is a stack, \(key\) is of \(S\)'s base type;

- \(aux\) is empty stack

*[Postconditions:]*

(a) returns *true* if \(key\) is in \(S\); otherwise *false*;
(b) \(S\) is in same state as at start.

\[
\begin{align*}
found & \leftarrow false \\
\text{while not empty}(S) & \\
& \quad \text{[Loop invariant: } found \text{ is true if and only if } key \text{ is in } aux]\n\end{align*}
\]

\[
\begin{align*}
test & \leftarrow Pop(S) \\
& \quad \text{if } test = key \\
& \quad \quad \text{found} \leftarrow true \\
& \quad \quad \text{Push}(aux, test)
\end{align*}
\]

*[Postcondition:]* (a) *found* is true iff *key* in *aux*.

\[
\begin{align*}
\text{while not empty} & \quad (aux) \\
& \quad \text{[Loop invariant: } aux \text{ reversed + } S \text{ contains original contents of } S]\n\end{align*}
\]

\[
\begin{align*}
& \quad \text{Push}(S, Pop(aux)) \\
& \quad \text{[Postcondition:} \quad (b) \ S \text{ in same state as at start of algorithm.}] \\
\text{return } & \quad found
\end{align*}
\]
Iterative Quicksort using stack class

```c
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());
}
```

3. Stack implementations

- Array
- Linked list
3. Stacks and queues

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”

C linked-list-based stack of characters

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

typedef struct Lists lists;

typedef struct Stacks stacks;

void stack_init(stacks* s);
void stack_push(stacks* s, char c);
char stack_pop(stacks* s);
bool stack_is_empty(stacks s);
void stack_display(stacks s);
3. Stacks and queues

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)

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
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

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();`

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 q
The qu
The qui
The quiet
The quiet qu
e quiet qu
quiet que
quiet queue

queue is
queue is q
**** matches q???e ****
queue is qu
eue is qui
ue is quit
e is quite
is quite q
s quite qu
quite qu
quite qui
quite quick

quiet queue
quiet 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);

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. Queue implementations

- Array
- Linked list

#### Array-based queue

<table>
<thead>
<tr>
<th>Front</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bill</td>
<td>Jane</td>
<td>Tina</td>
<td>Val</td>
<td>Mary</td>
<td></td>
<td>5</td>
<td>size</td>
</tr>
</tbody>
</table>

- To enqueue:
  
  *If size ≤ capacity*
  
  
  $A[size] \leftarrow \text{new\_value}$;
  
  increment size

  $O(\text{____})$

- To dequeue:
  
  *If size > 0, front \leftarrow A[1]*
  
  Decrement size;
  
  For $i \leftarrow 1$ to size, $A[i] \leftarrow A[i+1]$;
  
  Return front

  $O(\text{____})$
Circular-array queue code

```c
#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

```cpp
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

<table>
<thead>
<tr>
<th>term</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array implementation of queue</td>
<td>linked-list implementation of stack, queue</td>
</tr>
<tr>
<td>array implementation of stack</td>
<td>pop</td>
</tr>
<tr>
<td>circular array</td>
<td>push</td>
</tr>
<tr>
<td>circular list</td>
<td>queue</td>
</tr>
<tr>
<td>dequeue</td>
<td>restricted access</td>
</tr>
<tr>
<td>enqueue</td>
<td>reverse Polish notation</td>
</tr>
<tr>
<td>FIFO</td>
<td>ring buffer</td>
</tr>
<tr>
<td>is_empty</td>
<td>stack</td>
</tr>
<tr>
<td>is_full</td>
<td>top of stack</td>
</tr>
<tr>
<td>LIFO</td>
<td></td>
</tr>
</tbody>
</table>