Stacks

• **Stack**: restricted variant of list
  – Elements may be inserted or deleted from only one end \( \Leftarrow \Rightarrow \) LIFO lists
  – **Top**: the accessible end of the list
  – Operations: “push”, “pop”
  – Many applications
  – Easy to implement
    • Arrays
    • Linked lists
• Checking mathematical expressions
  – Equal number of left/right (, {}, [, ), }, ]
    $7 - ((X \times ((X + Y)/(J - 3) + Y))/(4 - 2.5))$
    - $((A + B)$
    - $A + B - C)$
    - $(A + B)) - CA$
    wrong
• **Algorithm**: read the expression from left to right
  – *p*: next symbol in expression
  – Repeat until (empty(stack)) {
    read(p);
    If p = ( or { or [ then push(stack,p);
    loop;
    if p = ) or } or ] then c = pop(stack);
    if c != p then error!!
  }
  if (not empty(stack)) then error!!
\[
\{x + (y - [a + b]) \times c - [(d + e)]\}
\]

\[
\{x + (y - [a + b]) \times c - [(d + e)]\} / (h - (j - (k - [l - n])))
\]

E.G.M. Petrakis
stacks, queues
• **Array-based implementation**: a simplified version of list implementation
  – Array of fixed size
  – **Top**: always the first element (0-th element)
  – **Current** is always the **top** element
  – push \(\leftrightarrow\) insert at current!
  – pop \(\leftrightarrow\) remove current!
template <class ELEM> class Stack {

private:
    int size; // Max size
    int top;  // Top ELEM
    ELEM * listarray; //Array holding ELEM's

public:
    Stack (const int sz = LIST_SIZE) //Constructor: initialize
    {size = sz; top = 0; listarray = new ELEM[sz];}
    ~Stack( ) { delete [ ] listarray;} //Destructor: return array space
    void clear( ) {top = 0;}  //Remove all ELEM's
    void push(const ELEM& item) //Push ELEM onto stack
    {assert (top<size); listarray[top++] = item;}
    ELEM pop( ) // Pop ELEM from top of stack
    {assert (!isEmpty( )); return listarray[--top];}
    ELEM topValue( ) const //Return value of top ELEM
    {assert(!isEmpty( )); return listarray[top-1];}
    bool isEmpty( ) const //Return TRUE if stack is empty
    {return top==0;}
};

E.G.M. Petrakis stacks, queues
• **Linked list implementation**: simplified version of the linked list implementation
  – The **head** and **tail** pointers of the list implementation are not used
template <class ELEM> class Stack {
private:
    link<ELEM> *top; //Pointer to top stack ELEM
public:
    Stack (const int) //Constructor: initialize
        { top = NULL; } //Constructor: initialize
    ~Stack( ) { clear( );} //Destructor: return ELEMs
    void clear( ) { } //Remove all ELEM's
    void push(const ELEM & item) //Push ELEM onto stack
        {top = new link <ELEM> (item, top);} //Push ELEM onto stack
    ELEM pop( );
    ELEM topLevel( ) const //Return value of top ELEM
        {assert(!isEmpty( )); return topÆelement;}
    bool isEmpty( ) const //Return TRUE if empty
        {return top == NULL;}
};
• More applications of stacks: evaluation of arithmetic expressions
  – Conversion of **infix** to **postfix** or **prefix**
  – Evaluation of the postfix or prefix
  – **Infix**, **prefix**, **postfix**: refer to the relative position of the operator with respect to the operands
    – **infix**: A+B
    – **prefix**: +AB  **polish**
    – **postfix**: AB+  **reverse polise**
• Convert infix to prefix or postfix
  – Read from left to right
  – First convert operations of higher precedence
    • Parenthesis have the highest precedence
    • “^” have precedence over
    • “*”, “/” have the same precedence but higher than
    • “+”, “-” which all have the same precedence
  – The converted prefix or postfix part is treated as a single operand
  – If all operators have the same precedence then convert from left to right
• **Infix to postfix**

  - **A+B*C** : ‘*’ > ‘+’
  - **A+(B*C)** : equivalent
  - **A+(BC*)** : B*C to postfix
  - **ABC** : postfix

  - **(A+B)*C** : (…) > ‘*’
  - **(AB+)*C** : A+B to postfix
  - **(AB+)*C** : (AB+)*C to postfix
  - **AB+C** : postfix

• **Postfix and Prefix have no parenthesis !!**
• **Infix to postfix**
  
  • \( A+B-C \) : \( AB+C- \)
  
  - \( (A+B)*(C-D) \) : \( AB+CD-* \)
  
  - \( A$B*C-D+E/F/(G+H) \) : \( AB$C*D-EF/GH+/+ \)
  
  - \( A-B/(C*D$E) \) : \( ABCDE*$/- \)

• **Infix to prefix**
  
  • \( A+B-C \) : \(-+ABC\)
  
  - \( (A+B)*(C-D) \) : \(+AB-CD\)
  
  - \( A$B*C-D+E/F/(G+H) \) : \(+-$ABCD//EF+GH\)
  
  - \( A-B/(C*D$E) \) : \(-A/B*C$DE\)
• Evaluation of postfix expressions:

While (not end of expression) {
    read symbols from left to right;
    result = 0
    if (symbol == operand) push (stack);
    else {
        operator1 = pop(stack);
        operator2 = pop(stack);
        result = operator1 operand operator2;
        push(stack, result);
    }
}
result = pop(stack);
### Postfix: $6\ 2\ 3\ +\ -\ 3\ 8\ 2\ /\ +\ *\ 2\ \$\ 3\ +$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operand1</th>
<th>Operand2</th>
<th>Value</th>
<th>stack Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>6,2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>6,2,3</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6,5</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1,3</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1,3,8</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1,3,8,2</td>
</tr>
<tr>
<td>/</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>1,3,4</td>
</tr>
<tr>
<td>+</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>1,7</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7,2</td>
</tr>
<tr>
<td>$$</td>
<td>7</td>
<td>2</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>2</td>
<td>49</td>
<td>49,3</td>
</tr>
</tbody>
</table>

E.G.M. Petrakis 49 3stacks, queues 52
• **Convert infix to postfix**: read expression from left to right
  – Output operands
  – Push operators to stack
  – Higher precedence operators must be above lower precedence operators
  – Before pushing an operator into the stack check the precedence of operators already in the stack
    • if operator in stack has higher precedence ➔ output this operator and then insert the current operator in the stack
    • “(”, “[”, “{” have higher precedence, push in stack
    • if “)”, “]”, “}” pop stack until “(”, “[”, “{” is found
    • don’t output “(”, “[”, “{”, “)”, “]”, “}”
• **Example:** \((A + B) \times C\)
  
  - push “(” in stack
  - output \(A\)
  - push “+” in stack
  - output \(B\)
  - don’t push “)”’, pop all operators in stack
  - output “+”, ignore “(”, “)”
  - “*” push in stack
  - output \(C\)
  - output “+”
Example: \((A + B) \times C\)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Postfix</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>A</td>
<td>(</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>(</td>
</tr>
<tr>
<td>+</td>
<td>A</td>
<td>( +</td>
</tr>
<tr>
<td>B</td>
<td>AB</td>
<td>(</td>
</tr>
<tr>
<td>)</td>
<td>AB+</td>
<td>(</td>
</tr>
<tr>
<td>*</td>
<td>AB+</td>
<td>*</td>
</tr>
<tr>
<td>C</td>
<td>AB+C</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>AB+C*</td>
<td></td>
</tr>
</tbody>
</table>


stack = NULL;

while (not end of input) {
    symbol = read next symbol;
    if (symbol == operand) output(symbol);
    else {
        while (not empty(stack)) and precedence(stack(top) > precedence(symbol))
        {
            top symbol = pop(stack);
            output(top symbol);
        }
        push(stack, symbol);
    }
    while not_empty(stack) {
        top symbol = pop(stack);
        output(top symbol);
    }
}

E.G.M. Petrakis

stacks, queues
Queue

- Queue elements may only be inserted from the rear and removed from the front
  - Restricted form of list
  - FIFO: first in first out
  - Insert: enqueue operation
  - Remove: dequeue operator
Enqueue (queue, D)

dequeue (queue)
• **Array-based implementation:**
  – If the elements are the first $n$ elements of array
  – And the rear element is at position 0
  – $dequeue$ requires $\Theta(1)$ operations but,
  – $enqueue$ requires $\Theta(n)$ operations (all elements must be shifted)
  – The reverse if the last element is at $n-1$ position
  – The condition of empty queue is also a problem
Condition of empty queue: rear = front
Initial values: rear = front = LIST_SIZE - 1

front points to the position proceeding the first element
The condition of empty queue is true but the queue is not empty. “E” cannot be inserted: The last array position must be left empty.

front = 4 rear = 2

front = 4 rear = 3

front = 4 rear = 4

front = 4 rear = 2

front = 4 rear = 3

front = 4 rear = 4

front = 4 rear = 2

front = 4 rear = 3
Delete “A”

front = 0
rear = 3

Delete “B”

front = 1
rear = 3
Insert “E”, “F”

Circular list

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

front = 1
rear = 0
template <class Elem> class Queue {
private:
    int size; // size of queue
    int front; // Index prior to front item
    int rear; // Index of rear item
    Elem *listarray; // Array holding the list Elem's
public:
    Queue(int sz = LIST_SIZE) // array one position larger for empty slot
    {
        size = sz + 1;
        front = rear = 0;
        listarray = new Elem[size];
    }
    ~Queue( ) { delete [ ] listarray; }
    void clear( ) { front = rear; } // Clear queue
    void enqueue(const Elem); // Enqueue Elem at rear Elem
    dequeue( ); // Dequeue Elem from front Elem
    firstValue( ) const // Get value of front Elem
    { assert(!isEmpty( )); return listarray[(front+1) % size]; }
    bool isEmpty( ) const // TRUE if queue is empty
    { return front == rear; }
};
// Enqueue Elem at rear of queue

```cpp
template <class Elem>
void Queue<Elem>::enqueue(const Elem item) {
    assert(((rear+1) % size) != front); // Queue must not be full
    rear = (rear+1) % size; // Increment rear (in circle)
    listarray[rear] = item;
}
```

// Dequeue Elem from front of queue

```cpp
template <class Elem>
Elem Queue<Elem>::::dequeue( ) {
    assert(!isEmpty()); // There must be something to dequeue
    front = (front+1) % size; // Increment front
    return listarray[front]; // Return value
}
```
• **Linked-list implementation**: simple adaptation of the linked list implementation
  – Current always points to the first element
  – The *head* pointer of the list implementation is not used
template <class ELEM> class Queue { //Linked queue class
private:
    link<ELEM>* front;       // Pointer to front queue node
    link<ELEM>* read;        // Pointer to rear queue node
public:
    Queue() { front = rear = NULL; } // Constructor: initialize
    ~Queue() { clear( ); } // Destructor: return link ELEMs
    void clear( ); // Remove all ELEM's from queue
    void enqueue(const ELEM & ); // Enqueue ELEM at rear
    ELEM dequeue( ); // Dequeue ELEM from front
    ELEM firstValue( ) const
        { assert(!isEmpty( )); return front->element; }
    bool isEmpty( ) const
        { return front == NULL; } // Return TRUE if queue is empty
}
template <class ELEM>
void Queue<ELEM>::clear() { //Remove all ELEM's from the queue
    while (front != NULL) //Return link nodes to freelist
    {
        rear = front; front = front->next; delete rear;
    }
    rear = NULL;
}

//Enqueue ELEM at rear of queue
template <class ELEM>
void Queue<ELEM>::enqueue (const ELEM& item){
    if (rear != NULL){ //Queue not empty: add to end
        rear->next = new link<ELEM>(item,NULL);
        rear=rear->next;
    } else front = rear = new link<ELEM>(item, NULL); //Empty queue
}
template <class ELEM>
ELEM Queue<ELEM>::dequeue() { //Dequeue ELEM from front
    assert(!isEmpty( )); //Must be something to Dequeue
    ELEM temp=front->element; //Store dequeued value
    link<ELEM>* ltemp = front; //Hold onto dequeued link node
    front=front->next; // Advance front
    delete ltemp; //Return link to free store
    if (front == NULL) rear = NULL; //Dequeued last element;
    return temp; //Return element value
}
Priority Queue

- **Ascending**: elements in any order but *dequeue* removes the minimum
- **Descending**: *dequeue* removes the maximum
front = 5
rear = 4

ascending

front = 5
rear = 4

descending

front = 5
rear = 4
Problem: in the array implementation, dequeue creates empty slots
Solution(1): move elements $\Rightarrow \Theta(n)$ operations on dequeue

Solution(2): store elements in ascending (descending) order $\Rightarrow \Theta(n)$ operations on enqueue