Postfix (and prefix) notation

- Also called "reverse Polish" – reversed form of notation devised by mathematician named Jan Łukasiewicz (so really lü-kä-sha-vech notation)
- Infix notation is: `operand operator operand`
  - Like `4 + 22`
  - Requires parentheses sometimes: `5 * (2 + 19)`
- Postfix form is: `operand operand operator`
  - So `4 22 +`
  - No parentheses required: `5 2 19 + *`
- Prefix is `operator operand operand`: `+ 4 22`

Evaluating postfix expressions

- Algorithm (start with an empty stack):

```
while expression has tokens {
    if next token is operand /* e.g., number */
          push it on the stack;
    else /* next token should be an operator */
          pop two operands from stack;
          perform operation;
          push result of operation on stack;
}
pop the result; /* should be only thing left on stack */
```

Postfix evaluation example

- Expression: `5 4 + 8 *`
  - Step 1: `push 5`
  - Step 2: `push 4`
  - Step 3: `pop 4, pop 5, add, push 9`
  - Step 4: `push 8`
  - Step 5: `pop 8, pop 9, multiply, push 72`
  - Step 6: `pop 72` – the result
- A bad postfix expression is indicated by:
  - Less than two operands to pop when operator occurs
  - More than one value on stack at end

Evaluating infix expressions

- Simplest type: fully parenthesized
  - e.g., `( ( ( 6 + 9 ) / 3 ) * ( 6 - 4 ) )`
- Still need 2 stacks: 1 numbers, 1 operators

```
while tokens available {
    if (number) push on number stack;
    if (operator) push on operator stack;
    if ( '(' ) do nothing;
    else { /* must be ')' */
       pop two numbers, and one operator;
       calculate; push result on number stack;
    }
} /* should be one number left on stack at end: the result */
```

Converting infix to postfix

- Operator precedence matters
  - e.g., `3+(10–2)*5` → `3 10 2 - 5 * +`
- Algorithm uses one stack; prints results
  (alternatively, could append results to a string)
  - For each token in the expression:

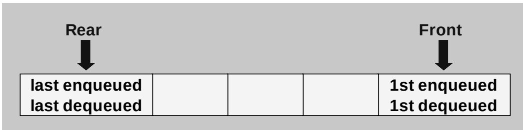
```
if ( number ) print it;
if ( '(' ) push on stack;
if ( ')' )
    pop and print all operators until '(';
    discard '(';
if ( operator ) /* more complicated – next slide */
```

Infix to postfix (cont.)

```
/* call current token the "new operator" */
while (stack is not empty)
{   peek at top operator on stack;
    if (top operator precedence
          >= new operator precedence)
        pop and print top operator;
    else break out of while loop;    }
push new operator on stack after loop ends;
```

- At end, pop and print all remaining operators
  - This algorithm does not account for all bad expressions – e.g., does not check for too many operators left at end
  - But can verify that parentheses are balanced

Queues



- FIFO data structure – First In, First Out
- Typical operations: `enqueue` (an item at rear of queue), `dequeue` (item at front of queue), `peek` (at front item), `empty`, `full`, `size`, `clear`
  - i.e., *very* similar to a stack – limited access to items

Some queue applications

- Many operating system applications
  - Time-sharing systems rely on process queues
    - Often separate queues for high priority jobs that take little time, and low priority jobs that require more time (see last part of section 7.8 in text)
  - Printer queues and spoolers
    - Printer has its own queue with bounded capacity
    - Spoolers queue up print jobs on disk, waiting for print queue
  - Buffers – coordinate processes with varying speeds
- Simulation experiments
  - Models of queues at traffic signals, in banks, etc., used to "see what happens" under various conditions

A palindrome checker

- Palindrome - same forward and backward
  - e.g., Abba, and "Able was I ere I saw Elba."
- Lots of ways to solve, including recursive
- Can use a queue and a stack together
  - Fill a queue and a stack with copies of letters
  - Then empty both together, verifying equality
- Reminder – we're using an *abstraction*
  - We still don't know how queues are implemented!!! To use them, it does not matter!

Implementing queues

- Easy to do with a list:
  - Mostly same as stack implementation
  - Enqueue: `insertLast(item, list);`
  - Then to dequeue and peek: refer to first item
- Array implementation is trickier:
  - Must keep track of front and rear indices
  - Increment front/rear using modulus arithmetic
    - Indices cycle past last index to first again – idea is to reuse the beginning of the array after dequeues
  - More efficient – but can become full
    - Usually okay, but some queues should be unbounded

Linked lists revisited: variations

- Some are meant to speed up operations
  - e.g., O(n) complexity to access last item
    - Way to make it O(1): maintain pointer to last – easy and worth it!
    - Another way: circular, double-linked list – not so easy
- Some are meant to increase usefulness
  - e.g., circular list to solve Josephus problem
  - e.g., generalized lists (lists of *lists* – upcoming topic)
- Trade-offs: use more space, harder to program

Implementing "better" lists

- Using double-linked lists – both harder and easier
  - Must keep track of twice as many pointers
  - Additional work required for most special cases
  - But easy insert before, traverse backwards, access last
- Can use sentinel nodes that are hidden from user
  - e.g., first and last sentinals – list is never really empty
    - Eliminates lots of special cases – just have to "lie" to user
  - e.g., n^{th} position sentinels – to speed access to i^{th} item
- Usually trading off: speed ↔ space ↔ effort

Generalized lists

- When list items may be sublists
  - May also contain just single items – called "atoms"
- Usually implement with `union` in node structure
  - e.g., instead of just info field, have info *or* sublist:

```
union SubNodeTag{
     InfoPointer info;
     NodePointer sublist;
} SubNode;
```

  - Also need field to identify a node as atom or sublist
- Lots of applications – see text section 8.4

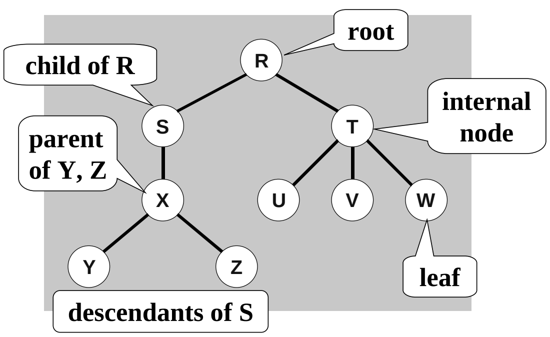
Is `<string.h>` an ADT?

- Combined with `(char *)` data it is!
- Easy to formalize – say `String.h`:

```
typedef char *String; /* the data type */
int strlen(String); /* length of string */
int strcmp(String,String); /* compare 2 strings */
String strcpy(String,String); /* copy 2nd to 1st */
/* and so on */
```

- Note what doesn't matter:
  - How strings are represented internally
  - How these functions are implemented

Trees



Binary trees

- Each node can have 0, 1, or 2 children only
- i.e., a binary tree node is a subtree that is either empty, or has left and right subtrees
  - Notice this is a recursive definition
  - Concept: a leaf's "children" are two empty subtrees
- Half (+1) of all nodes in full binary tree are leaves
  - All nodes except leaves have 2 non-empty subtrees
  - Exactly 2^k nodes at each depth `k`, $\forall$`k <` (leaf level)
- A complete binary tree satisfies two conditions
  - Is full except for leaf level
  - All leaves are stored as far to the *left* as possible

Representing trees by links

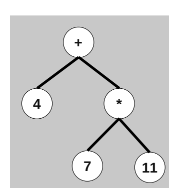
- Much more flexible than array representation
  - Because most trees are not as "regular" as heaps (later)
  - Array representation usually wastes space, and does not accommodate changes well
- Binary tree node has two links, one for each child

```
typedef struct treenode {
    DataType info;  /* some defined data type */
    struct treenode *left;  /* one child */
    struct treenode *right;  /* other child */
} TreeNode, *TreeNodePointer;  /* types */
```

- Not a binary tree? – keep *list* of children instead

Traversing binary trees

- Example: an expression tree (a type of "parse tree" built by advanced recursion techniques discussed in chapter 14) representing this infix expression: `4 + 7 * 11`



- Infix is in-order traversal
  - Left subtree → node → right subtree
- But can traverse in other orders
  - Pre-order: node → left → right, gives prefix notation: `+ 4 * 7 11`
  - Post-order: left → right → node, gives postfix notation: `4 7 11 * +`