Applications of Lists:

List of buffers (editor)
List of tabs (browser)
List of processes (OS)
List of projects to be graded (elearning)

Queues

Job scheduling
Communication / Messaging
Multimedia streaming
Data Communication
Printing
Wait Lists
Recursive listing of directories
Web Crawlers
Virus Scanners
Breadth-first Search (BFS)
Profit-Loss accounting of stock trades

Stacks:

Parsing of arithmetic expressions
Evaluation of arithmetic expressions
Conversion of infix expressions to postfix
Evaluation of postfix expressions
Implementation of function calls
Parsing of programming languages
XML Parsing (balanced parentheses)
Maze generation
Depth-first Search (DFS)

Advanced applications of lists

Arbitrary precision arithmetic
(bc in Unix, Mathematica, Big Integer in Java)
Sparse polynomials
Symbolic mathematical expressions and their manipulations
(e.g. differentiation and integration)

Arbitrary-Precision Arithmetic, Polynomials

Polynomials can be stored and manipulated using lists:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

can be represented by the list

$$\boxed{a_0} \to \boxed{a_1} \to \boxed{a_2} \to \cdots \to \boxed{a_n}$$

Functions can be written for addition, multiplication, composition, evaluation of polynomials.

Bignum arithmetic (BigInteger in Java).

choose a base B for the arithmetic.

Then a number can be expressed in base B and stored in a list:

$$X = a_0 + a_1 \cdot B + a_2 B + \cdots + a_n B^n$$

(Here $0 \le a_i < B$ for $i = 0..n$).

List for X:

$$\boxed{a_0} \to \boxed{a_1} \to \boxed{a_2} \to \cdots \boxed{a_n}$$

Example: If $B = 100$, the number 8102409 will be stored as

$$\boxed{9} \to \boxed{24} \to \boxed{10} \to \boxed{8}$$

choice of B:

If B is large, then the number of "digits" (n) stored is smaller – leads to faster algorithms

but if B is too big, it leads to overflow errors during calculations.

Usually, B is chosen such that B^2 does not overflow in that type.

Sparse polynomials

$$P(x) = 1 + 100x^{49} + 3x^{256}$$

standard way of storing $P(x)$ wastes space for many zero valued coefficients (and the algorithms are slower).

Better solution:

$$\boxed{\begin{array}{c} 1 \\ \hline 0 \end{array}} \to \boxed{\begin{array}{c} 100 \\ \hline 49 \end{array}} \to \boxed{\begin{array}{c} 3 \\ \hline 256 \end{array}} \begin{array}{l} \text{coefficient} \\ \text{exponent} \end{array}$$

Algorithm to add two numbers stored in (polynomial scheme) a list using base B:

(next() helper function discussed earlier)

```
List add ( List l1 , List l2 )
    it1 = l1. iterator ( )
    it2 = l2 . iterator ( )
    X1 = next (it1) ;  X2 = next (it2)
    Carry = 0
    Create empty list outList
    while (X1 != null && X2 != null) {
        Sum = X1 + X2 + carry
        outList.add (Sum % B)
        carry = sum / B        //Integer division
        X1 = next (it1) ;   X2 = next (it2);
    }
    while (X1 != null) {
        Sum = X1 + carry
        outList. add (Sum % B)
        Carry = Sum /B
        X1 = next (it1)
    }
    while (X2 != null) {
        Sum = X2 + carry
        outList.add (Sum % B)
        Carry = sum/B
        X2 = next (it2)
    }
    if (carry > 0) { outList. add (carry)
return outList
```

Merge one list into another (internal to list class).

```
// Precondition: Lists are sorted
// Assume that Lists are singly linked, with dummy header node.
```

element
next

Entry

```
//Invariant:  p - end of output list
//            e1 - next entry of L1 to be processed
//            e2 - next entry of L2 to be processed

p = L1. header ;   e1 = p.next ;   e2 = L2.header.next
while (e1 != null && e2 != null) {
    if (e1.element <= e2.element ) {
        p.next = e1
        p = e1
        e1 = e1.next
    } else {
        p. next = e2
        p = e2
        e2 = e2.next
    }
}
// One of the lists has reached the end
if (e1 == null) {
        p.next = e2
} else {
        p.next = e1
}
```

Parsing Context-free Languages - LR Parsers

Context-free languages (CFL) are languages that are generated by Context-free grammars (CFG) - also known as BNF Backus-Naur Form

Non-terminals = N. start symbol $S \in N$.

Terminals = Σ Empty string = ε

Production rules:

Nonterminal $\rightarrow$ string composed of $N \cup \Sigma$.

Example:

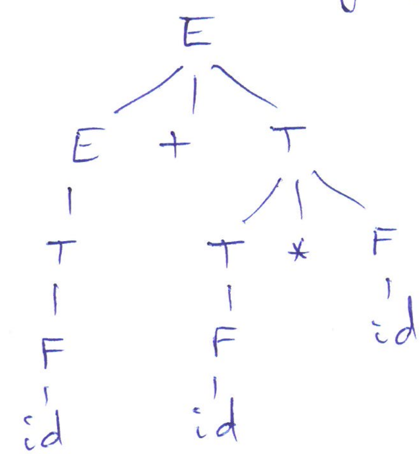
$$[S \rightarrow E]$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

Grammar generates arithmetic expressions with $*$, $+$, and parentheses, with higher precedence for $*$ than $+$.

A right-most derivation for $id + id * id$

$$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id$$
$$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id$$
$$\Rightarrow id+id*id$$

Corresponds to the following parse tree:



LR(1) parsing:

L - Left-to-right scan of input

R - Right-most derivation

1 - # of tokens of Look-ahead

Parser = Finite state machine on $N \cup \Sigma$, stack for $N \cup \Sigma \cup$ states of FSM

Parsing table = Action table + Goto table

action on Σ action on N

Actions: shift, reduce

Ex: s6 : shift to state 6
Move token from input to stack, change state of FSM to 6

r3 : reduce using rule # 3
If rule 3 is production $A \rightarrow \beta$, replace β on top of stack by A.

LR Parsing table for the following context-free grammar:

Production Rules 0-6: $E' \rightarrow E$, $E \rightarrow E+T$, $E \rightarrow T$, $T \rightarrow T*F$, $T \rightarrow F$, $F \rightarrow (E)$, $F \rightarrow id$.

State	**+**	*****	**(**	**)**	**id**	**$**	**E**	**T**	**F**
0			S4		S5		1	2	3
1	S6					accept			
2	R2	S7		R2		R2			
3	R4	R4		R4		R4			
4			S4		S5		8	2	3
5	R6	R6		R6		R6			
6			S4		S5			9	3
7			S4		S5				10
8	S6			S11					
9	R1	S7		R1	R1	R1			
10	R3	R3		R3	R3	R3			
11	R5	R5		R5	R5				

Input expression: $a+b*c$. From lexical analyzer: *id* + *id* * *id* $

State	**Stack**	**Rest of input**	**Action**
0	$	*id* + *id* * *id* $	S5
5	$ 0.*id*	+ *id* * *id* $	R6; goto(0,*F*) = 3
3	$ 0.*F*	+ *id* * *id* $	R4; goto(0,*T*) = 2
2	$ 0.*T*	+ *id* * *id* $	R2; goto(0,*E*) = 1
1	$ 0.*E*	+ *id* * *id* $	S6
6	$ 0.*E* 1.+	*id* * *id* $	S5
5	$ 0.*E* 1.+ 6.*id*	* *id* $	R6; goto(6,*F*) = 3
3	$ 0.*E* 1.+ 6.*F*	* *id* $	R4; goto(6,*T*) = 9
9	$ 0.*E* 1.+ 6.*T*	* *id* $	S7
7	$ 0.*E* 1.+ 6.*T* 9.*	*id* $	S5
5	$ 0.*E* 1.+ 6.*T* 9.* 7.*id*	$	R6; goto(7,*F*) = 10
10	$ 0.*E* 1.+ 6.*T* 9.* 7.*F*	$	R3; goto(7,*T*) = 9
9	$ 0.*E* 1.+ 6.*T*	$	R1; goto(0,E) = 1
1	$ 0.*E*	$	accept

Rightmost derivation generated by the parsing algorithm:

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id \Rightarrow E+id*id \Rightarrow T+id*id+F+id*id \Rightarrow id+id*id$