# Lists

Ordered set of elements

$e_1, e_2, \ldots, e_n$

Lists allow nulls and duplicates.

Common implementations:

1. **Array List** — resizeable array to store elements

机制：满了就resize

Automatically resize array when it gets full.

If arr[n] gets full, allocate arr[2n],

Copy elements to new array.

If growth of array is by a constant factor (or more)

then amortized cost of add = O(1)

Operations: add O(1) amortized, remove O(1), **contains O(n)**, iteration hasNext O(1), next O(1), **remove O(n)**

删，查比较慢

indexing (goto $i^{th}$ element: get – O(1), set – O(1), **add – O(n), remove – O(n)**

尽量不要indexing 尽量用iterator

Efficient ops: add, remove, indexed get, set, iteration (no remove)

比如：for(int i=0;i<list.size();i++){
list.get(i); //很慢！
}

应改成：for(x: list){...}

?? If list shrinks, should array become smaller?

Disadvantage: Space used may be more than needed
Needs contiguous memory.

2. **Linked List**

$e_1 \to e_2 \to e_3 \to \ldots e_n$

Each element has a reference to next element in list. Elements are not in contiguous memory.

List can grow to occupy entire memory.

Operations: add O(1), remove O(1), **contains O(n)**, iteration hasNext O(1), next O(1), remove O(1)

**indexing: O(n)**

因为是不连续的空间，所以可以更好占用计算机中尽量多的内存

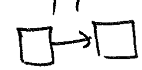
Efficient: add, remove, iteration (including remove)

Advantage: can grow to occupy full memory.

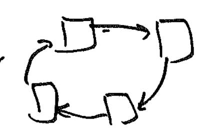
Disadvantage: non contiguous memory ⇒ cache management is not helpful.

缺点：缓存没用到

Variations: Singly linked, Doubly linked



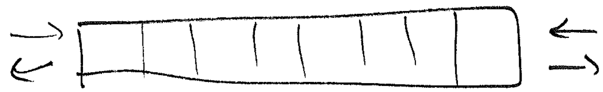
Circular



Java: LinkedList – Doubly-linked Circular list.

SLL: Dummy header or not?

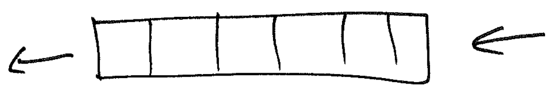
3. **ArrayDeque** (Array-based double-ended queue)



Replacement for both lists and stacks.
Resizeable array (similar to ArrayList).

~~Exten~~ Special lists:

1. Queue: FIFO (First-in, First out)



Java: Queue is an interface.
Implemented by LinkedList, ...

| **Single-threaded apps** | **multithreaded apps** (consumer/Producer) |
|---|---|
| add ( ) - add a new element to the queue (end) | **offer** ( ) - add element |
| remove( ) - remove an element (from front) | **poll** ( ) - remove element |
| | ↑ (usually on Bounded size queues) |
| isEmpty( ) - check if Q is empty | |
| peek ( ) - retrieve (but not remove) front | ≡ element ( ) ← Do not throw exceptions |

2 **stacks**: LIFO (Last-in First out)

Operations: Push Pop

Java Implementations: ~~stack~~ ArrayDeque

stack implementation: Array.

Running time per operation of List implementations on a list with n elements:

| Operation | Linked List | Array List |
|---|---|---|
| add | O(1) | O(1) amortized |
| remove | O(n) | O(n) |
| contains | O(n) | O(n) |
| size, isEmpty, clear | O(1) | O(1) |
| iteration | O(1) per element | O(1) per element |
| remove of iterator | O(1) | O(n) |
| Indexing: get, set, add | O(n) | O(1) |

**Applications of lists**
List of buffers (editor)
List of tabs (browser)
List of processes (OS)
List of projects to be graded (elearning)
Graph adjacency lists
Finish time order of DFS (topological order)

**Applications of stacks**
Parsing of arithmetic expressions
Evaluation of arithmetic expressions
Conversion of infix arithmetic expressions to postfix
Evaluation of postfix arithmetic expressions
Implementation of function calls
LL/LR parsing (parsing of programming languages)
XML parsing (balanced parentheses)
Maze generation
Depth-first search (DFS)

**Applications of queues**
Job scheduling
Communication / Messaging
Multimedia streaming
Data communication
Printing
Wait lists
Recursive listing of directories in a file system
Web crawlers
Virus scanners
Breadth-first search (BFS)
Profit/Loss accounting of stock trades

**Advanced applications**
Arbitrary precision arithmetic (bc, BigInteger)
Sparse polynomials
Symbolic mathematics (functions, polynomials, equation solvers, calculus)

Some useful links:

| | |
|---|---|
| Shunting yard algorithm for parsing | https://en.wikipedia.org/wiki/Shunting-yard_algorithm |
| Maze generation | https://en.wikipedia.org/wiki/Maze_generation_algorithm |

Arbitrary precision integer arithmetic: Choose a base B for the arithmetic. $X = a_0 + a_1 B + \ldots + a_n B^n$. Then, X can be stored using the list $\{a_0, a_1, \ldots, a_n\}$. Implement arithmetic operations, where numbers are stored in this format. Example: B=100, X=8102409 is stored as {9, 24, 10, 8}. If B=32768, then X is stored as {8713, 247}.

X=8713+247*B^1

As B gets larger, the list has smaller length, leading faster running times for operations. But, if B is too large, arithmetic operations cause overflows. Good choice of B: a power of 2, such that $B^2$ still fits in one word.

基数选太大会溢出，太小会太慢。 选择2的幂

**Predictive parsing (LL(1) parser)**

Use a stack to generate a leftmost derivation, using a lookahead of 1 token. Left recursion must be removed from the grammar LL(1) parsing table is constructed using first and follow sets of nonterminals. First(x) is the set of all characters that can appear as the first character in strings that can be derived from x, using the productions of the grammar. Follow(A) is the set of all terminal symbols that can appear to the immediate right of A in any derivation from S (the start symbol).

Example: $E \rightarrow E + T \mid T$ $\quad T \rightarrow T * F \mid F$ $\quad F \rightarrow (E) \mid num$

Here "+" is a placeholder for any arithmetic operator with the same precedence as the addition operator: {+, -}, and "*" represents {*, /, %} operators. The grammar has left recursion, which must be eliminated before predictive or top-down parsing is done. A new production $E' \rightarrow E_1$ is added, where E' is the start symbol.

| **Productions** | **First** | **Follow** |
|---|---|---|
| $E_1 \rightarrow T_1\ E_2$ | ( num | ) $ |
| $E_2 \rightarrow + T_1\ E_2 \mid \varepsilon$ | + ε | ) $ |
| $T_1 \rightarrow F\ T_2$ | ( num | + ) $ |
| $T_2 \rightarrow * F\ T_2 \mid \varepsilon$ | * ε | + ) $ |
| $F \rightarrow (E_1) \mid num$ | ( num | * + ) $ |

From the above information, the LL(1) parsing table can be computed:

| | **+** | **mulOp** | **num** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|
| $E_1$ | | | $E_1 \rightarrow T_1\ E_2$ | $E_1 \rightarrow T_1\ E_2$ | | |
| $E_2$ | $E_2 \rightarrow + T_1\ E_2$ | | | | $E_2 \rightarrow \varepsilon$ | $E2 \rightarrow \varepsilon$ |
| $T_1$ | | | $T_1 \rightarrow F\ T_2$ | $T_1 \rightarrow F\ T_2$ | | |
| $T_2$ | $T_2 \rightarrow \varepsilon$ | $T_2 \rightarrow * F\ T_2$ | | | $T_2 \rightarrow \varepsilon$ | $T_2 \rightarrow \varepsilon$ |
| F | | | $F \rightarrow num$ | $F \rightarrow (E_1)$ | | |

Example: Leftmost derivation of the expression "3+4*5": $E' \Rightarrow E_1 \Rightarrow T_1E_2 \Rightarrow FT_2E_2 \Rightarrow 3T_2E_2 \Rightarrow 3E_2 \Rightarrow 3{+}T_1E_2 \Rightarrow 3{+}FT_2E_2 \Rightarrow 3{+}4T_2E_2 \Rightarrow 3{+}4{*}FT_2E_2 \Rightarrow 3{+}4{*}5T_2E_2 \Rightarrow 3{+}4{*}5E_2 \Rightarrow 3{+}4{*}5$

| # | Proc | Stack | Input | Production | # | Proc | Stack | Input | Production |
|---|---|---|---|---|---|---|---|---|---|
| 1) | - | $E_1\$$ | 3+4*5$ | $E_1 \rightarrow T_1E_2$ | 7) | 3+ | $FT_2E_2\$$ | 4*5$ | $F \rightarrow num$ (4) |
| 2) | - | $T_1E_2\$$ | 3+4*5$ | $T_1 \rightarrow FT_2$ | 8) | 3+ | $4T_2E_2\$$ | 4*5$ | $T_2 \rightarrow *FT_2$ |
| 3) | - | $FT_2E_2\$$ | 3+4*5$ | $F \rightarrow num$ (3) | 9) | 3+4 | $*FT_2E_2\$$ | *5$ | $F \rightarrow num$ (5) |
| 4) | - | $3T_2E_2\$$ | 3+4*5$ | $T_2 \rightarrow \varepsilon$ | 10) | 3+4* | $5T_2E_2\$$ | 5$ | $T_2 \rightarrow \varepsilon$ |
| 5) | 3 | $E_2\$$ | +4*5$ | $E_2 \rightarrow +T_1E_2$ | 11) | 3+4*5 | $E_2\$$ | $ | $E_2 \rightarrow \varepsilon$ |
| 6) | 3 | $+T_1E_2\$$ | +4*5$ | $T_1 \rightarrow FT_2$ | 12) | 3+4*5 | $ | $ | Accept |

**Recursive descent parsing**

Like predictive parsing, the grammar is modified to remove left recursion. Example:

E’ → E
E → T { addOp T }  // { … } means optional, and can be repeated, same as * in regexp
T → F { mulOp F }
F → (E) | num
addOp → + | -
mulOp → * | / | %

Class Token { … token (int or enum type) …}

Algorithm to evaluate infix expression:

```
E'() {
   rv = E();
   if q.peek() == $ then
         return rv;
   else
         Error
}

E() {
   rv = T();
   while(q.peek().token == addOp) {
         oper = q.remove();
         rv2 = T();
         rv = exec(oper, rv, rv2);
   }
   return rv;
}

T() {
   rv = F();
   while(q.peek().token == mulOp) {
         oper = q.remove();
         rv2 = F();
         rv = exec(oper, rv, rv2);
   }
   return rv;
}
```

```
F() {
   switch(q.peek().token) {
         case '(':
            q.remove();
            rv = E();
            if (q.peek() == ')')
                  q.remove();
            else
                  Error
            break;
         case num:
            rv=q.remove().value();
            break;
         default:
            Error
   }
   return rv;
}
```

## LR parsing

Bottom-up parsing. Generate rightmost derivation of input. Parsing table can be generated by software tools (called parser generators). Most programming languages are parsed using LALR(1) parsing (a simplified version of LR(1) parsing). LR Parsing table for the following context-free grammar:

Production Rules 0-6: $E' \to E$ $\quad E \to E + T$ $\quad E \to T$ $\quad T \to T*F$ $\quad T \to F$ $\quad F \to (E)$ $\quad F \to id$

| 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* $. Rightmost derivation generated by the parsing algorithm: $E \Rightarrow E + T \Rightarrow E + T*F \Rightarrow E + T*id \Rightarrow E + F*id \Rightarrow E + id*id \Rightarrow T + id*id + F + id*id \Rightarrow id + id*id$

| 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 |