Chapter 2 Stacks

数据科学与计算机学院　　黄方军

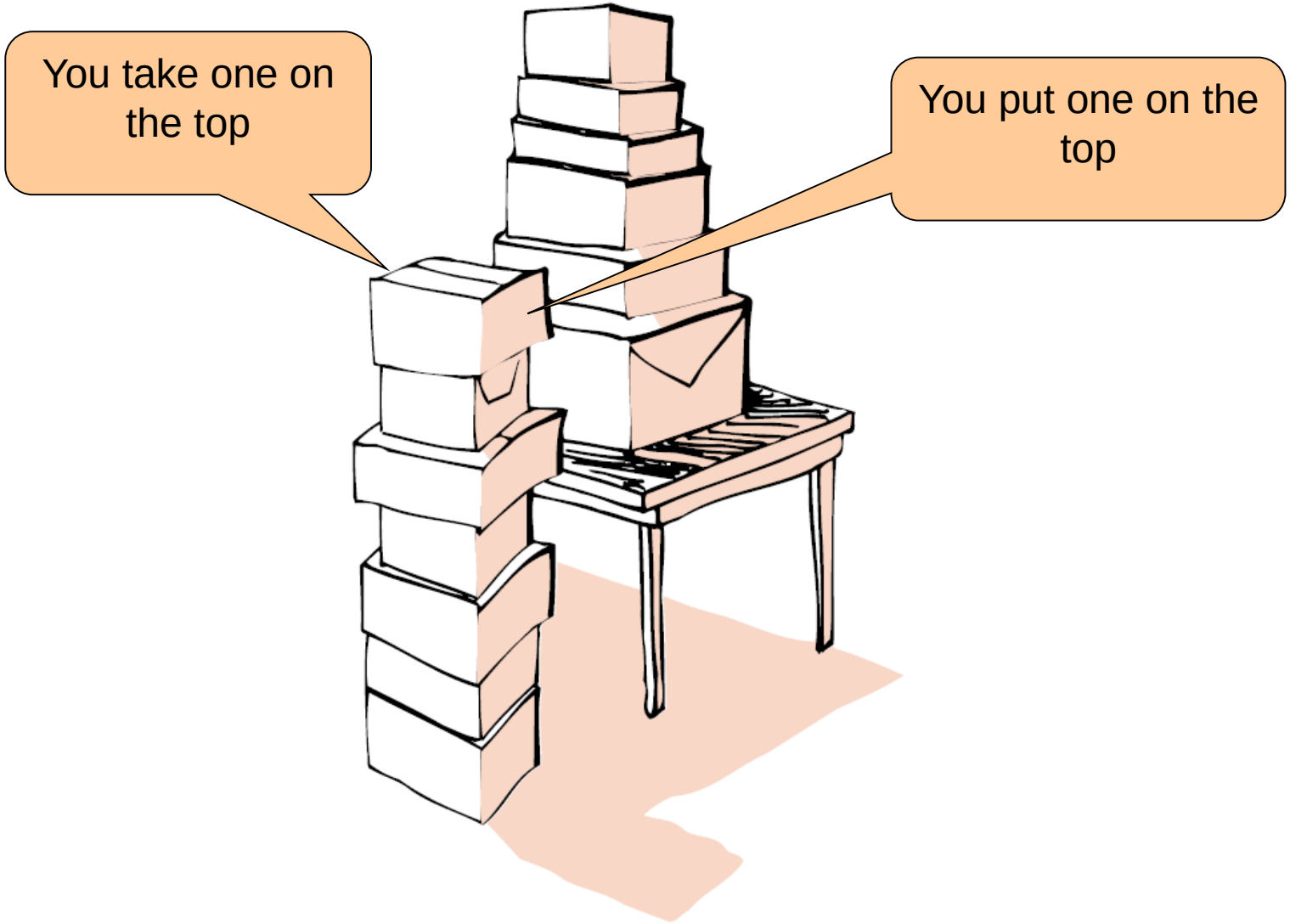
data_structures@163.com

东校区实验中心B502

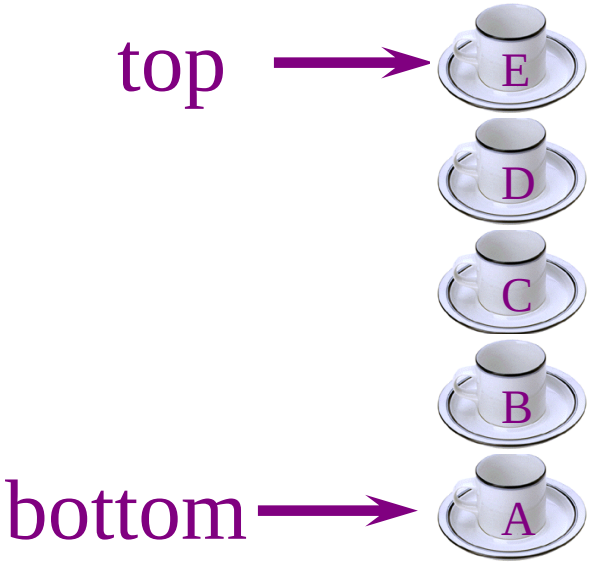
2.1.1 Lists and Arrays

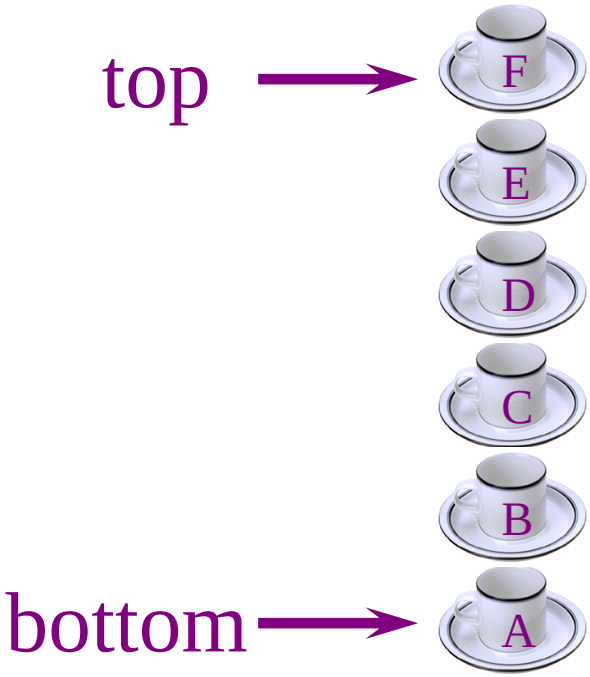
- A list is a dynamic data structure because its size is variable;
- An array is a static structure because it has a fixed size

2.1.2 Stacks



2.1.2 Stacks

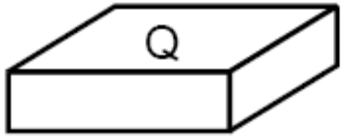




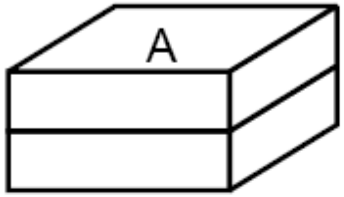
A stack is a LIFO list.

2.1.3 First Example: Reversing a List

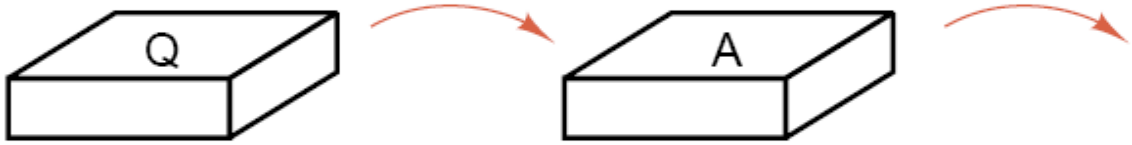
Push box Q onto empty stack:



Push box A onto stack:

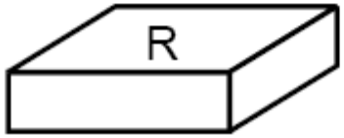


Pop a box from stack:



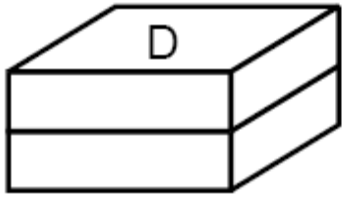
Pop a box from stack:

Push box R onto stack:

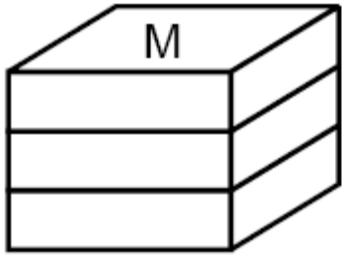


2.1.3 First Example: Reversing a List

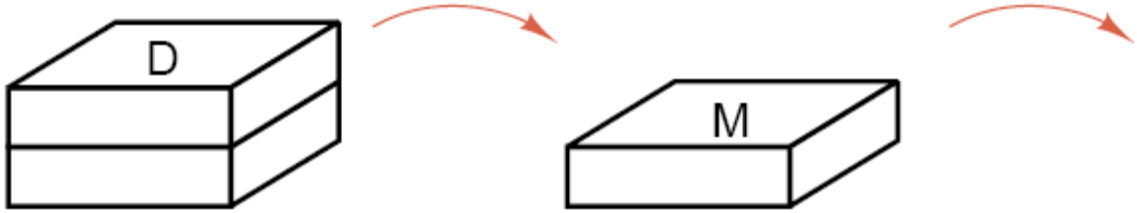
Push box D onto stack:



Push box M onto stack:

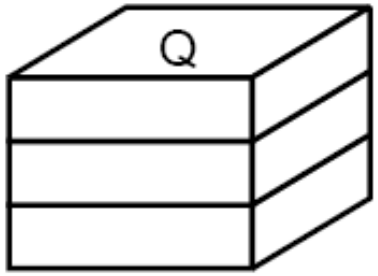


Pop a box from stack:



2.1.3 First Example: Reversing a List

Push box Q onto stack:



Push box S onto stack:

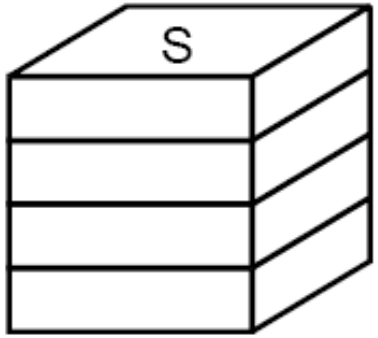


Figure 2.2. Pushing and popping a stack

2.1.3 First Example: Reversing a List

```
#include <stack>
int main()
{    int n, i;
     double item;
      stack<double> numbers;
     cin >> n;
     for (i=0; i<n; i++)
     {    cin >> item;
          numbers. push(item);  }
      while(!numbers.empty())
      {    cout<< numbers.top() << ‘ ‘ ;
           numbers.pop();   }
}
```

2.2 Implementation of Stacks

```
const int maxstack = 10;          //    small value for testing

class Stack {
public:
    Stack( );
    bool empty( ) const;
    Error_code pop( );
    Error_code top(Stack_entry &item) const;
    Error_code push(const Stack_entry &item);

private:
    int count;
    Stack_entry entry[maxstack];
};
```

2.2.1 Specification of Methods for Stacks

- Constructors
- Entry Types, Generics

  e.g., typedef char Stack_entry;
- Error Processing

  e.g., success, overflow, underflow
- Specification for methods

  e.g., pop, push, etc.

2.2.2 The class Specification

```
const int maxstack = 10;          //    small value for testing

class Stack {
public:
    Stack( );
    bool empty( ) const;
    Error_code pop( );
    Error_code top(Stack_entry &item) const;
    Error_code push(const Stack_entry &item);

private:
    int count;
    Stack_entry entry[maxstack];
};
```

2.2.3 Pushing , Popping, and Others

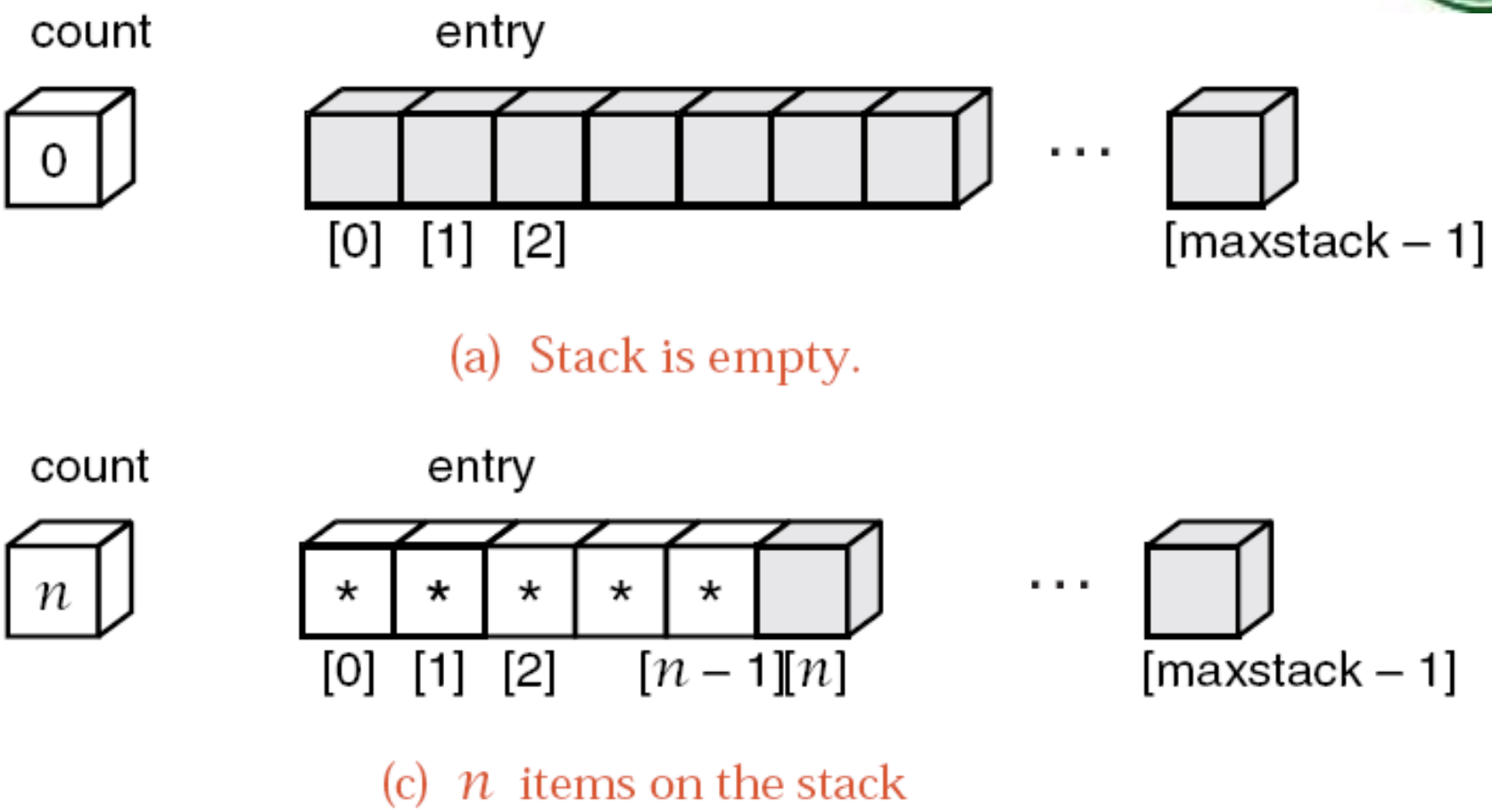


Fig. 2.4. Representation of data in a contiguous stack

2.2.3 Pushing , Popping, and Others

```
Error_code Stack::push(const Stack_entry &item)
/* Pre:   None.
   Post:  If the Stack is not full, item is added to the top of the Stack. If the Stack is
          full, an Error_code of overflow is returned and the Stack is left unchanged.
          */
{
   Error_code outcome = success;
   if (count >= maxstack)
      outcome = overflow;
   else
      entry[count++] = item;
   return outcome;
}
```

2.2.3 Pushing , Popping, and Others

```
Error_code Stack::pop()
/* Pre:   None.
   Post:  If the Stack is not empty, the top of the Stack is removed.  If the Stack is
          empty, an Error_code of underflow is returned.  */
{
   Error_code outcome = success;
   if (count == 0)
      outcome = underflow;
   else --count;
   return outcome;
}
```

2.2.3 Pushing , Popping, and Others

```
Error_code Stack :: top(Stack_entry &item) const
/* Pre:   None.
   Post:  If the Stack is not empty, the top of the Stack is returned in item. If the Stack
          is empty an Error_code of underflow is returned.  */
{
   Error_code outcome = success;
   if (count == 0)
      outcome = underflow;
   else
      item = entry[count − 1];
   return outcome;
}
```

Nothing is changed, including count.

2.2.3 Pushing , Popping, and Others

```
bool Stack::empty( ) const
/* Pre:   None.
   Post:  If the Stack is empty, true is returned. Otherwise false is returned.  */
{
   bool outcome = true;
   if (count > 0) outcome = false;
   return outcome;
}
```

How about the implementation using linked lists?

2.2.3 Pushing , Popping, and Others

```
Stack::Stack()
/* Post: The stack is initialized to be empty. */
{
    count = 0;
}
```

2.2.4 Encapsulation

- Public
- Private

2.3 Application: A desk calculator

- ? Denote an instruction to read an operand and push it onto the stack;
- +, -, *, and / represent arithmetic operations;
- = is an instruction to print the top of the stack (but not pop it off)

2.3 Application: A desk calculator

- ?a?b+=

  (a+b)

- ?a?b+?c?d+*=

  (a+b) * (c+d)

- ?a?b?c-=*?d+=

  (a*(b-c)) + d

2.3 Application: A desk calculator

```
double p, q;
  switch (command) {
  case '?':
      cout << "Enter a real number: " << flush;
      cin >> p;
      if (numbers.push(p) == overflow)
         cout << "Warning: Stack full, lost
                  number" << endl;
      break;
```

2.3 Application: A desk calculator

```
case '=':
   if (numbers.top(p) == underflow)
      cout << "Stack empty" << endl;
   else
      cout << p << endl;
   break;
```

2.3 Application: A desk calculator

```
case '+':
  if (numbers.top(p) == underflow)
    cout << "Stack empty" << endl;
  else {
    numbers.pop();
    if (numbers.top(q) == underflow) {
      cout << "Stack has just one entry" << endl;
      numbers.push(p);
    }
    else {
      numbers.pop();
      if (numbers.push(q + p) == overflow)
        cout << "Warning: Stack full, lost result" << endl;
    }
  }
  break;
```

2.4 Application: Bracket matching

{a = (1 + v(b[3 + c[4]])); ×

{ a = (b[0) + 1]; } ×

{()[()]} √

2.4 Application: Bracket matching

```
{
  Stack openings;
  char symbol;
  bool is_matched = true;
  while (is_matched && (symbol = cin.get()) != '\n') {
    if (symbol == '{' || symbol == '(' || symbol == '[')
      openings.push(symbol);
    if (symbol == '}' || symbol == ')' || symbol == ']') {
      if (openings.empty()) {
        cout << "Unmatched closing bracket " << symbol
             << " detected." << endl;
        is_matched = false;
      }
```

2.4 Application: Bracket matching

```
        else {
            char match;
            openings.top(match);
            openings.pop();
            is_matched = (symbol == '}' && match == '{')
                         || (symbol == ')' && match == '(')
                         || (symbol == ']' && match == '[');
            if (!is_matched)
                cout << "Bad match " << match << symbol << endl;
        }
    }
}
if (!openings.empty())
    cout << "Unmatched opening bracket(s) detected." << endl;
}
```

2.4 Application: Bracket matching

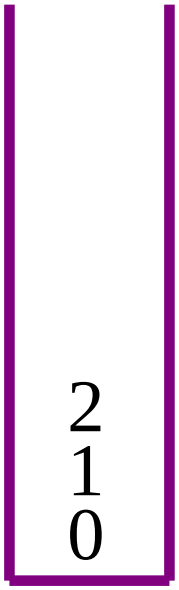
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis at 7 has no matching right parenthesis

2.4 Application: Bracket matching

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack

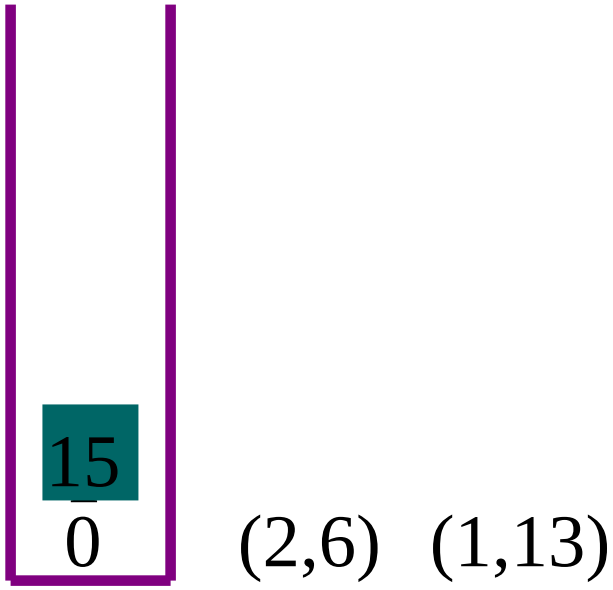
2.4 Application: Bracket matching

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



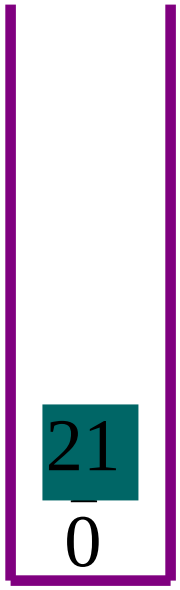
2.4 Application: Bracket matching

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



2.4 Application: Bracket matching

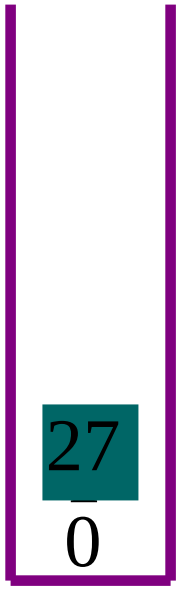
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



(2,6) (1,13) (15,19)

2.4 Application: Bracket matching

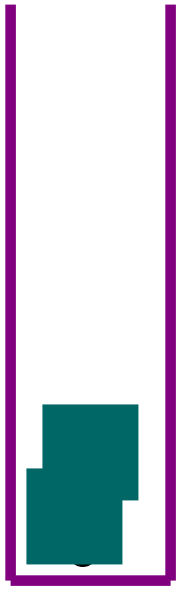
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



(2,6) (1,13) (15,19) (21,25)

2.4 Application: Bracket matching

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



(2,6) (1,13) (15,19) (21,25) (27,31) (0,32)

2.5 Application: Rearranging Railroad Cars