对集合中对象的表示方法将直接影响各种操作的效率。

链式数据结构是高效地实现背包、队列和栈的基础。

# API

* 一个无参的构造函数。
* 一个向集合中添加单个元素的方法。
* 一个测试集合是否为空的方法。
* 一个返回集合大小的方法。
* 一个能够删除集合中特定元素的方法。（Stack 和 Queue）

|  |  |
| --- | --- |
| 背包 | 不支持从中删除元素的集合。遍历时，元素的处理顺序不重要。例如，计算一组值的平均数、方差等。 |
| 先进先出队列 | FIFO，按照任务产生的顺序完成它们。任何服务型策略的基本原则都是公平，应该优先服务等待最久的人，如售票厅，收费站等，元素的处理顺序就是它们被添加到队列中的顺序。保存元素时要保存它们的相对顺序。 |
| 下压栈 | LIFO，后进先出。新邮件总是排在最前面。浏览器的回退按钮。元素的处理顺序与被压入的顺序正好相反。 |

# 栈用例：算术表达式求值

( 1 + ( ( 2 + 3 ) \* ( 4 \* 5 ) ) )

E.W.Dijkstra，20 世纪 60 年代，用两个栈，一个用于保存运算符，一个用于保存操作数：

* 将操作数压入操作数栈。
* 将运算符压入运算符栈。
* 忽略左括号。
* 在遇到有右括号时，弹出一个运算符，弹出所需数量的操作数，并将运算符和操作数的运算结果压入操作数栈。

import java.util.Scanner;

public class Evaluate {

public static void main(String[] args) {

Stack<String> ops = new Stack<String>(10);

Stack<Double> vals = new Stack<Double>(10);

Scanner sc = new Scanner(System.in);

// 在控制台输入 ( 1 + ( ( 2 + 3 ) \* ( 4 \* 5 ) ) )

while(sc.hasNext()){

String s = sc.next();

if(s.equals("("))                                         ;

else if(s.equals("+"))                         ops.push(s);

else if(s.equals("-"))                         ops.push(s);

else if(s.equals("\*"))                         ops.push(s);

else if(s.equals("/"))                         ops.push(s);

else if(s.equals("sqrt"))          ops.push(s);

else if(s.equals(")")) {

String op = ops.pop();

double v = vals.pop();

if(op.equals("+"))                 v = vals.pop() + v;

else if(op.equals("-"))         v = vals.pop() - v;

else if(op.equals("\*"))         v = vals.pop() \* v;

else if(op.equals("/"))         v = vals.pop() / v;

else if(op.equals("sqrt"))         v = Math.sqrt(v);

vals.push(v);

} else vals.push(Double.parseDouble(s));

}

// 按 Ctrl-Z 结束输入，继续程序

sc.close();

System.out.println(vals.pop());

}

}

# 集合数据类型的实现：算法 1.1 定容下压栈

**优点：**

每项操作的用时都与集合大小无关；

空间需求总是不超过集合大小乘以一个常数。

**缺点：**

某些 push() 和 pop() 操作会调整数组的大小，这项操作的耗时和栈大小成正比。

package com.feng.algrithms.ch01;

import java.util.Iterator;

public class FixedCapacityStack<Item> implements Iterable {

private Item[] a; // 栈元素

private int N; // 栈大小

public FixedCapacityStack(int cap) { a = (Item[]) new Object[cap]; }

public boolean isEmpty() { return N == 0; }

public int size() { return N; }

public void push(Item item) {

if (N == a.length)

resize(2 \* a.length);

a[N++] = item;

}

public Item pop() {

Item item = a[--N];

a[N] = null; // 避免对象游离：保存一个不需要的对象的引用称为游离。

// 当栈的大小为数组的四分之一时，将数组长度减半，这样在下次需要改变数组大小之前还能进行多次 push() 和 pop() 操作

// 栈的使用率永远不低于四分之一

if (N > 0 && N == a.length / 4)

resize(a.length / 2);

return item;

}

// 栈永远不会溢出

private void resize(int max) {

// 将大小为 N <= max 的栈移动到一个新的大小为 max 的数组中

Item[] temp = (Item[]) new Object[max];

for (int i = 0; i < N; i++) {

temp[i] = a[i];

}

a = temp;

}

@Override

public Iterator<Item> iterator() {

return new ReverseArrayIterator();

}

private class ReverseArrayIterator implements Iterator<Item> {

private int i = N;

@Override

public boolean hasNext() { return i > 0; }

@Override

public Item next() { return a[--i]; }

@Override

public void remove(){}

}

}

## 关于迭代

Stack<String> collection = new Stack<String>();

for(String s : collection)

System.out.println(s);

这里，foreach 语句只是 while 语句的一种简写方式，其本质与如下 while 语句是等价的：

Iterator<String> i = collection.iterator();

while(i.hasNext()){

String s = i.next();

Ssystem.out.println(s);

}

这段代码展示了一些在任意可迭代的集合数据类型中我们都要实现的东西：

* 集合数据类型必须继承 Iterable 接口，实现它的 iterator() 方法，返回一个合适的 Iterator 对象；
* 该 Iterator 对象是一个内部类的实例，该内部类继承 Iterator ，覆盖其 hasNext() 和 next() 方法——迭代器是一个实现了 hasNext() 和 next() 方法的类的对象。

# 链表

这是我们构造非 Java 直接支持的数据结构的第一个例子。

定义：链表是一种递归的数据结构，它或者为空（null），或者是指向一个结点（node）的引用，该结点含有一个泛型的元素和一个指向另一条链表的引用。

## 节点记录

在面向对象编程中，实现链表并不困难。我们首先用一个嵌套类来定义结点的抽象数据类型：

private class Node<Item> {

Item item;

Node next;

}

## 构造链表

public class App {

private static class Node<Item> {

Item item;

Node next;

}

public static void main(String[] args) {

Node first = new Node();

Node second = new Node();

Node third = new Node();

first.item = "to";

second.item = "be";

third.item = "or";

first.next = second;

second.next = third;

}

}

first、second 和 third 是三条链表。

first 表示的序列是 to、be、or。也可以用数组表示一系列元素：

String[] s = {"to", "be", "or"};

不同之处在于，在链表中向序列头尾插入元素或从链表头删除元素都更方便。

## 对链表的操作

* 在表头插入结点：该操作所需时间与链表长度无关。

Node oldfirst = first; // 使 oldfirst 指向 first 所指向的结点

first = new Node(); // 使 first 指向新的结点

first.item = "N"; // 初始化新结点的泛型元素

first.next = oldfirst; // 使新节点的 next 指向 oldfirst

* 在表尾插入结点：该操作所需时间与链表长度无关。

Node oldlast = last; // 使 oldlast 指向 last 所指向的节点

last = new Node(); // 使 last 指向新的结点

last.item = "question"; // 初始化新结点的泛型元素

last.next = null; // 使新节点的 next 指向 null

oldlast.next = last; // 使 oldlast.next 指向 last

* 从表头删除结点：该操作所需时间与链表长度无关。

first = first.next;

* 从表头之外其他位置删除指定结点
* 在表头尾之外的其他节点前插入一个新结点

以上两种操作都难以实现。例如，如何才能删除链表的尾节点呢，last 链接（链接表示对结点的引用）帮不上忙，因为我们需要将 last 之前的结点中的链接值改为 null。唯一的办法是遍历整条链表并找出指向 last 的结点。这种方法所需时间与链表长度成正比。实现任意插入和删除操作的标准解决方法是使用**双向链表**，其中每个结点都含有两个链接，指向不同的方向。

* 遍历

for (Node x = first; x != null; x = x.next) {

// 处理 x.item

}

# 算法 1.2 栈的实现

使用数组实现栈的缺点在于：某些 push() 和 pop() 操作会调整数组的大小，这项操作的耗时和栈大小成正比。使用链表实现栈可以避免这个问题，它将栈保存为一条链表，栈的顶部即为表头，实例变量 first 指向栈顶：

* 可以处理任意类型数据（使用了泛型）
* 所需空间总是与集合大小成正比
* push() 和 pop() 操作所需时间总是和集合大小无关

**链表是数组的一种重要替代方式。**这种代替方案已经有数十年的历史。

这份实现是我们对许多算法的实现的原型。它定义了链表数据结构并实现了供用例使用的方法 push() 和 pop()，仅用少了代码就取得了所期望的效果。算法和数据结构是相辅相成的。

package com.feng.algrithms.ch01;

import java.util.Iterator;

public class Stack<Item> implements Iterable<Item> {

private Node first = null; // 栈顶（最近添加的元素）

private int N = 0; // 元素数量

private class Node {

Item item;

Node next;

}

public boolean isEmpty() {

return first == null; // 或 N == 0

}

public int size() {

return N;

}

public void push(Item item) { // 在栈顶添加元素

Node oldfirst = first;

first = new Node();

first.item = item;

first.next = oldfirst;

N++;

}

public Item pop() { // 从栈顶删除元素

Item item = first.item;

first = first.next;

N--;

return item;

}

@Override

public Iterator<Item> iterator() { // 支持遍历

return new ListIterator();

}

private class ListIterator implements Iterator<Item> {

private Node current = first; // 遍历操作从栈顶开始

@Override

public boolean hasNext() {

return current != null;

}

@Override

public Item next() {

Item item = current.item;

current = first.next;

return item;

}

@Override

public void remove() {

}

}

}

# 算法 1.3 先进先出队列的实现

它将队列表示为一条从最早插入的元素到最近插入的元素的链表，实例变量 first 指向队列的开头，实例变量 last 指向队列的结尾。

* 可以处理任意类型数据（使用了泛型）
* 所需空间总是与集合大小成正比
* push() 和 pop() 操作所需时间总是和集合大小无关

package com.feng.algrithms.ch01;

import java.util.Iterator;

import edu.princeton.cs.algs4.StdIn;

import edu.princeton.cs.algs4.StdOut;

public class Queue<Item> implements Iterable<Item> {

private Node first = null; // 指向最早添加的结点的链接

private Node last = null; // 指向最近添加的结点的链接

private int N = 0; // 队列中的元素数量

private class Node {

Item item;

Node next;

}

private boolean isEmpty() {

return first == null;

}

private int size() {

return N;

}

public void enqueue(Item item) { // 入列：向队尾添加一个元素

Node oldlast = last;

last = new Node();

last.item = item;

last.next = null;

if (isEmpty()) {

// 至此，如果是首次入列操作，first 一定为 null，所以 isEmpty() 一定为 true

first = last; // 由于此时只有一个结点，所以让 first 也指向这个结点

} else {

oldlast.next = last;

}

N++;

}

public Item dequeue() { // 出列：从队列开头一个元素

Item item = first.item;

first = first.next; // 当结点数量由 1 变为 0 时，first 会变为 null

if (isEmpty()) {

// 当结点数量由 1 变为 0 时，first 为 null，isEmpty() 为真。此时队列中已经没有结点，所以 first 和 last 都应指向 null。

last = null;

}

N--;

return item;

}

@Override

public Iterator<Item> iterator() { // 支持遍历

return new ListIterator();

}

private class ListIterator implements Iterator<Item> {

private Node current = first;

@Override

public boolean hasNext() {

return current != null;

}

@Override

public Item next() {

Item item = current.item;

current = first.next;

return item;

}

@Override

public void remove() {

}

}

// 用例程序，输入 to be or not to - be - - that - - - is，按 Ctrl-Z 继续程序

public static void main(String[] args) {

Queue<String> q = new Queue<String>();

while (!StdIn.isEmpty()) {

String item = StdIn.readString();

if (!item.equals("-")) {

q.enqueue(item);

} else if (!q.isEmpty()) {

StdOut.print(q.dequeue() + " ");

}

}

StdOut.println("(" + q.size() + " left on queue.)");

}

}

算法 1.4 背包

用链表实现背包秩序将 Stack 中的 push() 改名为 add()，并去掉 pop() 的实现即可。