数据结构与算法

第 4 讲：字符串

韩文弢

清华大学

2024—2025 学年度春季学期

本讲内容

4.1 问题引入 2
4.2 字符串的概念 4
4.3 字符串的实现 11
4.4 字符串拓展内容 20
4.5 字符串的应用 23
4.6 小结 33

4.1 问题引入

4.1.1 字符串处理的需求

日常生活中，很多地方要处理一串由基本单元组成的内容。

图 4.1　文本编辑

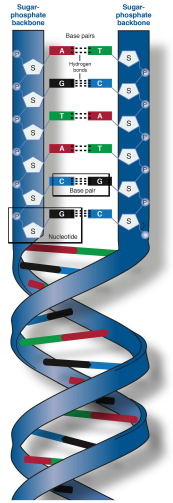


图 4.2　基因序列

如何用计算机来表示和处理这样的内容?

4.2 字符串的概念

4.2.1 字符串的定义

定义 4.1 字符串 S 是一个有穷的字符序列，由零个或多个字符组成。每个字符都属于同一个有穷的字符集合 Σ。零个字符的字符串称为**空串**，空串不包含任何字符。

记字符串 $S = s_0 s_1 \cdots s_{n-1}$ 或 $\emptyset$，其中 $n \geq 0$ 为字符串 S 的长度，$s_i \in \Sigma (0 \leq i < n)$ 是字符串 S 的第 i 个字符。

$\emptyset$ 表示空串，空串的长度为 0。

4.2.2 子串的定义

定义 4.2 字符串中任意个连续的字符组成的子序列称为该字符串的**子串**。

例. $S =$ “abcd” 是长度为 4 的字符串（习惯上用引号把字符串的内容括起来）。

则 S 的所有子串包括 “”（空串，即 $\emptyset$） “a” “b” “c” “d” “ab” “bc” “cd” “abc” “bcd” “abcd”（本身）。

4.2.3 字符串的抽象数据类型

伪码 4.1: 字符串的 ADT

元素：$D = \{s_i \mid s_i \in \Sigma, i = 0, 1, 2, ..., n-1\}$，$\Sigma$ 是字符全集，$n \geq 0$ 是字符串长度

关系：$R = \{(s_i, s_{i+1}) \mid s_i, s_{i+1} \in \Sigma, i = 0, 1, 2, ..., n-2\}$

基本操作：

Initialize(S): 初始化一个空的字符串 S

Destroy(S): 销毁字符串 S

IsEmpty(S): 返回字符串 S 是否为空串

Length(S): 返回字符串 S 的长度

Compare(S, T): 比较两个字符串 S 和 T 的大小，返回 -1、0 或 1

Substring(S, p, l): 返回字符串 S 中从位置 p 开始、长度为 l 的子串

Insert(S, p, T): 将字符串 T 插入到字符串 S 的位置 p

Remove(S, p, l): 删除字符串 S 中从位置 p 开始的 l 个字符

Concatenate(S, T): 连接字符串 S 和 T

Search(S, T): 返回字符串 T 在字符串 S 中首次出现的位置，若没有出现则返回 $\emptyset$

4.2.4 字符串比较

两个字符串比较大小采用**字典序**比较方式。

定义 4.3 (字典序) 两个字符串 $S = s_0 s_1 \cdots s_{n-1}$ 和 $T = t_0 t_1 \cdots t_{m-1}$ 比较大小，$S < T$ 成立当且仅当

1. 存在下标 k 满足对所有 $0 \leq i < k$ 有 $s_i = t_i$，且 $s_k < t_k$；或
2. $n < m$，且对所有 $0 \leq i < n$ 有 $s_i = t_i$。

例. “abc” < “abd”，“ab” < “abc”。

4.2.5 字符串查找

在一个字符串中查找另一个字符串是否出现是一种常用操作。

定义 4.4 在字符串 S 中找出与字符串 T 相等的子串，称为字符串的**模式匹配**（pattern matching），也称为查找或定位操作。其中，字符串 S 称为**目标串**，字符串 T 称为**模式串**。若在字符串 S 中找到与字符串 T 相等的子串则匹配**成功**，否则匹配**失败**。

4.2.6 模式匹配的形式化定义

定义 4.5 设字符串 $S = s_0 s_1 \cdots s_{n-1}$，字符串 $T = t_0 t_1 \cdots t_{m-1}$。若存在 $p(0 \le p \le n-m)$，使得对所有 $0 \le i < m$ 都满足 $s_{p+i} = t_i$，则匹配成功，称 p 是一个**有效位移**。

例. $S =$“abbaba”，$T =$“aba”，$p = 3$ 是一个有效位移。

4.3 字符串的实现

4.3.1 字符串的存储方式

- 顺序存储：使用顺序表进行存储，可以随机访问指定位置的字符，插入和删除操作较慢。
- 链式存储：使用链表（或树状）结构进行存储，可以支持快速的插入和删除操作，但不支持随机访问。

课内只讲解字符串的顺序存储方式。

4.3.2 字符串的顺序存储方式

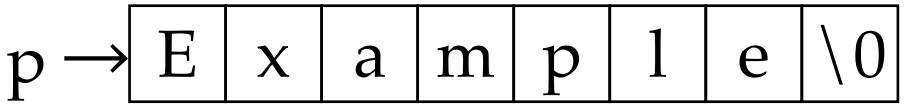
伪码 4.2: 字符串的顺序存储实现

String:

> data: 指向字符串数据的指针，数据连续存储
>
> size: 当前字符串的长度
>
> capacity: 分配的容量

4.3.3 另一种实现方式：C 风格字符串

C 语言使用另一种实现方式：以 '\0'（ASCII 码值为 0 的字符）结尾的字符指针（类型 char*）。



- 优点：额外的空间开销小
- 缺点：操作繁琐，需要程序员自己分配足够的空间，容易出错

4.3.4 字符串：比较

伪码 4.3: 字符串的字典序比较

```
Compare(S, T):
    n ← Length(S)
    m ← Length(T)
    for i ← 0 to min(n, m) − 1
        if S[i] < T[i]
            return −1
        else if S[i] > T[i]
            return 1
    if n < m
        return −1
    else if n > m
        return 1
    return 0
```

4.3.5 字符串：模式匹配

伪码 4.4: 朴素的字符串模式匹配算法

```
Search(S, T):
1   n ← Length(S)
2   m ← Length(T)
3   for i ← 0 to n − m
4       for j ← 0 to m − 1
5           if S[i + j] ≠ T[j]
6               break
7       if j = m   # m 个字符均匹配上
8           return i
9   return ∅
```

4.3.6 朴素模式匹配算法的效率

设目标串 S 的长度为 n，模式串 T 的长度为 m，则朴素模式匹配算法最坏情况下的时间复杂度为 $O(mn)$。

思考：什么时候朴素模式匹配算法的复杂度达到最坏情况?

例如：$S =$ “aaaaaaab”，$T =$ “aab”

如何改进？利用模式串本身的特点减少失配后的工作量，参见 KMP 算法，最坏情况时间复杂度为 $O(m + n)$。

但是实践中朴素算法一般效率也很高，且实现简单，系数小，被广泛使用。

4.3.7 C++ 中的字符串 string

C++ 标准库提供 `string` 类来表示字符串。

- 在头文件 `<string>` 中，底层是 `basic_string` 模板类，`string` 是 `basic_string<char>` 的别名。
- 采用类似 `vector` 的方式来管理存储空间。
- 提供了丰富的操作接口，重载了赋值、连接、比较等运算符。
- 定义了常量 `string::npos` 来表示最大长度、匹配失败等含义，可作为一些函数的参数，也是一些函数可能的返回值。
- 详细接口见 https://en.cppreference.com/w/cpp/string/basic_string。

4.3.8 修改操作的效率问题

C++

```cpp
#include <iostream>
#include <string>

int main() {
  const int n = 1000000;
  std::string s;
  for (int i = 0; i < n; i++) {
    s = s + ' ';
  }
  std::cout << s.length() <<
  std::endl;
}
```

C++

```cpp
#include <iostream>
#include <string>

int main() {
  const int n = 1000000;
  std::string s;
  for (int i = 0; i < n; i++) {
    s += ' ';
  }
  std::cout << s.length() <<
  std::endl;
}
```

比较这两段程序的执行时间： 左侧程序约 10 秒，右侧程序约 3 毫秒。 原因：左侧程序时间复杂度是 $O(n^2)$，右侧是 $O(n)$。

4.4 字符串拓展内容

4.4.1 只读视图 string_view

效率原则：在使用字符串时，要尽量避免创建新的字符串。

为了高效处理子串相关的问题，C++17 提供了只读视图 `string_view`。

- `string_view` 本身不管理存储空间，而是指向一个 `string` 中的部分内容。
- 不能通过 `string_view` 来修改内容，但是可以修改它所指的范围。
- 其他接口与 `string` 几乎一样，详细接口见 https://en.cppreference.com/w/cpp/string/basic_string_view。

4.4.2 输入输出

使用 C++ 标准库进行字符串相关的输入时有两种接口：

1. `istream` 重载的提取操作符 `>>`，在输入时以空白字符[1]为分隔符，一般用于输入单词。
2. `<string>` 头文件中提供的全局函数 `getline`，用于按整行进行输入。

[1]空白字符包括空格、制表符、换行符等。

4.5 字符串的应用

4.5.1 表达式求值

问题. 给出一个由数、运算符和小括号组成的表达式，其中数由 0 到 9 的数字组成；运算符有加减乘除（除法结果取整），先乘除后加减；小括号可以嵌套。计算该表达式的值，注意表达式可能会不合法，要能报错。

例如：

- `1+1` $\longrightarrow$ `2`
- `(1+5)/2` $\longrightarrow$ `3`
- `(8+0)/(9-(2*3+1))` $\longrightarrow$ `4`
- `(1+2` $\longrightarrow$ 报错

4.5.2 表达式求值：解题思路

使用第 2 讲提到的思路一：找到整个表达式优先级最低的运算符，将问题转化为对运算符两边的子表达式求值，然后计算最终结果。子表达式的求值通过递归来求解。

需要进一步考虑的问题：

- 如何寻找优先级最低的运算符？ 考虑每个位置当前括号嵌套层数，只需要考虑在 0 层的运算符，分情况讨论什么时候当前运算符是目前看到的优先级最低的运算符
- 如果 0 层没有运算符如何处理？ 此时表达式的形式是 (...)，去掉最外层括号递归计算里面的子表达式
- 既没有运算符也没有括号怎么处理？ 是纯数字，将字符串转化成整型数值

4.5.3 表达式求值：参考程序

C++

```cpp
#include <iostream>
#include <optional>
#include <string>
#include <string_view>

// 计算两个数的四则运算
std::optional<int> Calculate(int lhs, char op, int rhs) {
  switch (op) {
    case '+': return lhs + rhs;
    case '-': return lhs - rhs;
    case '*': return lhs * rhs;
    case '/':
```

4.5.3 表达式求值：参考程序

```
13          if (rhs != 0) {
14            return lhs / rhs;
15          } else {
16            return {};  // 被零除
17          }
18        default:
19          return {};  // 不支持的运算
20      }
21  }
22
23  // 计算一个表达式
24  std::optional<int> Evaluate(std::string_view expr) {
```

4.5.3 表达式求值：参考程序

```
25    if (expr.empty()) {
26      return {};  // 非法表达式：空串
27    }
28    int level = 0;  // 当前括号嵌套层数
29    std::size_t pos = std::string::npos;
30    for (std::size_t i = 0; i < expr.length(); i++) {
31      char ch = expr[i];
32      if (ch == '(') {
33        ++level;
34      } else if (ch == ')') {
35        if (--level < 0) {
36          return {};  // 非法表达式：括号不匹配
```

4.5.3 表达式求值：参考程序

```
37          }
38        } else if (level == 0) {
39          if (ch == '+' || ch == '-') {
40            pos = i;  // 如果是加减号，一定比前面出现的优先级低
41          } else if ((ch == '*' || ch == '/') &&
42                     (pos == std::string::npos ||
43                      expr[pos] == '*' || expr[pos] == '/')) {
44            pos = i;  // 如果是乘除号，只比前面出现的乘除号优先级低
45          }
46        }
47      }
48      if (level != 0) {
```

4.5.3 表达式求值：参考程序

```
49      return {};  // 非法表达式：括号不匹配
50    }
51    if (pos != std::string::npos) {  // 找到了最低优先级的运算符
52      auto left = Evaluate(expr.substr(0, pos));
53      if (!left) {
54        return {};  // 左子表达式求值错误
55      }
56      auto right = Evaluate(expr.substr(pos + 1));
57      if (!right) {
58        return {};  // 右子表达式求值错误
59      }
60      return Calculate(*left, expr[pos], *right);
```

4.5.3 表达式求值：参考程序

```
61    } else if (expr.front() == '(') {  // (xxx) 的形式
62      return Evaluate(expr.substr(1, expr.length() - 2));
63    } else {  // 数字
64      return std::stoi(std::string(expr));
65    }
66    return {};  // 其他错误
67  }
68
69  int main() {
70    std::string expr;
71    std::getline(std::cin, expr);
72    auto result = Evaluate(expr);
```

4.5.3 表达式求值：参考程序

```
73    if (result) {
74      std::cout << "Result: " << *result << std::endl;
75    } else {
76      std::cout << "Invalid expression" << std::endl;
77    }
78  }
```

4.6 小结

4.6.1 本讲小结

- 字符串的概念
- 字符串的实现
- C++ 中字符串相关的类：`string` 和 `string_view`
- 字符串的应用：表达式解析求值