栈与队列1

栈的概念

栈是最常用也是最常考的数据结构之一，其是很多场景的合适抽象。

栈是一个抽象的线性逻辑结构，其只有一端开口，这个开口可以添加值，即push，也可以在容器非空的时候查看开口数值，即top或者peek（C++里是top，Java里是peek），还可以在容器非空的时候将开口数值删除，即pop。开口的地方被称为“栈顶”，另一端则被称为“栈底”。所有的操作只能在栈顶进行。由于这个特殊性质，栈被称为是先进后出的，即LIFO或者FILO，即Last in first out或者First in last out。

栈的实现，可以用数组或者单链表。数组在尾端添加、删除都是非常快的，适合做栈顶；而单链表的表头进行添加、删除是非常快的，适合做栈顶。具体实现这里省略，大家可以自行实现。

例题：https://blog.csdn.net/qq_46105170/article/details/113798817

栈的应用

栈在操作系统中无处不在，其与函数调用有着深刻的关系。同时，栈与表达式解析和递归，也有着密不可分的联系。具体如下：

1. 函数调用与调用栈

在程序执行过程中，每次函数调用都需要保存当前的执行状态，以便在函数执行完毕后恢复。这些执行状态包括函数的参数、局部变量和返回地址等。调用栈就是用于存储这些信息的结构：

- **函数调用**：当一个函数被调用时，其活动记录（Activation Record）被压入调用栈。
- **函数返回**：函数执行完毕后，其活动记录被弹出，控制权返回给调用该函数的位置。

这种机制使得程序能够正确地管理多层次的函数调用，支持递归等复杂操作。

2. 表达式求值与语法解析

栈在编译器和解释器中用于表达式的解析和求值：

- **中缀转后缀**：将中缀表达式转换为后缀表达式（逆波兰表达式）时，运算符和操作数被压入和弹出栈，以确定正确的计算顺序。
- **表达式求值**：在求值后缀表达式时，操作数被压入栈，当遇到运算符时，弹出相应数量的操作数进行计算，结果再压入栈中。

3. 深度优先搜索（DFS）与回溯算法

在图论算法中，栈用于实现深度优先搜索：

- **路径记录**：当前访问的节点被压入栈，表示正在探索的路径。
- **回溯操作**：当无法继续前进时，从栈中弹出节点，回到上一个状态，尝试其他路径。

表达式求值

栈在算法中的最重要的作用之一，就是表达式求值。

例题：https://blog.csdn.net/qq_46105170/article/details/106229090

给定一个只含数字、小括号和加减乘除的表达式，其中可能存在负数，求该表达式的值。

这题有标准算法。我们首先要规定一下符号的“优先级”，注意这里左括号我们也要当成运算符来处理，左括号优先级最低，接着是加减，它们两个优先级相同并且都高于左括号优先级，优先级最高的是乘除。接着：

1. 开两个栈，一个是数字栈，只存数字，另一个是符号栈，只存左括号或四则运算符号。
2. 遍历表达式，遇到数字，截取之，然后push进数字栈；
3. 遇到左括号，将其push进符号栈；
4. 遇到右括号，只要符号栈的栈顶不是左括号，那么就则每次pop两个数字，pop一个符号，进行计算之后再将答案push回数字栈。重复操作直到符号栈栈顶是左括号为止，接着将左括号pop；
5. 遇到符号，如果符号栈为空，则push其进符号栈；如果符号栈不空，则只要符号栈栈顶的符号优先级大于（此处大于等于也是对的）当前遇到的符号，就进行计算（同4，pop两个数pop一个符号等等）。重复操作直到符号栈空，或者符号栈的栈顶优先级小于了当前符号。接着将当前符号push进符号栈；
6. 表达式遍历完之后，如果符号栈不空，则继续进行计算（同4，pop两个数pop一个符号等等）；
7. 数字栈栈顶即为答案。

代码如下：

```
class Solution {
 public:
  int calculate(string t) {
    string s;
    // 预处理一下t，主要是负号前加个0
    for (int i = 0; i < t.size(); i++) {
      if (t[i] == ' ') continue;
      if (t[i] == '-' && (s.empty() || s.back() == '(')) s += '0';
      s += t[i];
    }
    unordered_map<char, int> mp{
        {'(', 0}, {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}};
    stack<int> stk;
    stack<char> ops;
    for (int i = 0; i < s.size(); i++) {
```

```
      if (s[i] == '(') ops.push('(');
      else if (isdigit(s[i])) {
        int x = 0, j = i;
        while (j < s.size() && isdigit(s[j])) x = x * 10 + (s[j++] - '0');
        stk.push(x);
        i = j - 1;
      } else if (s[i] == ')') {
        while (ops.top() != '(') calc(stk, ops);
        ops.pop();
      } else {
        while (ops.size() && mp[ops.top()] >= mp[s[i]]) calc(stk, ops);
        ops.push(s[i]);
      }
    }

    while (ops.size()) calc(stk, ops);
    return stk.top();
  }

  void calc(stack<int>& stk, stack<char>& ops) {
    int b = stk.top(); stk.pop();
    int a = stk.top(); stk.pop();
    char op = ops.top(); ops.pop();
    switch (op) {
      case '+': stk.push(a + b); break;
      case '-': stk.push(a - b); break;
      case '*': stk.push(a * b); break;
      case '/': stk.push(a / b); break;
    }
  }
};
```

时空复杂度是$O(n)$，n为表达式长度。

表达式求值的状态机视角

我们假设某个表达式只含数字和加减乘除，并且不会出现除以0等不合法的操作。这种题目有一个“状态机”的思路来做。考虑一个数对$(0,0)$作为起始状态，对于中间状态(a,b)，每一步做如下操作：

1. 如果是$+x$，状态变为$(a+b,x)$；
2. 如果是$-x$，状态变为$(a+b,-x)$；
3. 如果是$\times x$，状态变为(a,bx)；
4. 如果是$/x$，状态变为$(a,b/x)$。

开头数字之前的运算符视为$+$。举例：$2+3\times4-5\times2$

状态变化是：

$(0,0) \to (0,2) \to (2,3) \to (2,12) \to (14,-5) \to (14,-10)$，最后答案就是两维相加，答案为$4$。读者可以验证这种方法是正确的。

如果含括号的话，表达式求值，除了要用状态机，还需要用递归。但思路是和上面差不多的，只不过遇到括号的时候，需要开一个“副本”，从$(0,0)$开始计算一下括号里的表达式的值。参考 https://blog.csdn.net/qq_46105170/article/details/115291782。代码如下：

```cpp
#include <iostream>
using namespace std;

string s;

void update(int &a, int &b, int x, char op) {
  switch (op) {
    case '+': a += b, b = x; break;
    case '-': a += b, b = -x; break;
    case '*': b *= x; break;
    case '/': b /= x; break;
  }
}

int dfs(int &idx, int a, int b) {
  char op = '+';
  while (idx < s.size() && s[idx] != ')') {
    char ch = s[idx];
    if (!isdigit(ch) && ch != '(') {
      op = ch;
      idx++;
    }
    if (ch == '(') {
      idx++;
      int x = dfs(idx, 0, 0);
      idx++;
      update(a, b, x, op);
    }
    if (isdigit(ch)) {
      int j = idx;
      while (j < s.size() && isdigit(s[j])) j++;
      int x = stoi(s.substr(idx, j - idx));
      update(a, b, x, op);
      idx = j;
    }
  }
  return a + b;
}
```

```
39
40  int main() {
41    cin >> s;
42    int idx = 0;
43    printf("%d\n", dfs(idx, 0, 0));
44  }
```

队列的概念

队列也是最常用也是最常考的数据结构之一。

队列是一个抽象的线性逻辑结构，其有两端开口，分别叫队头和队尾。队尾只能添加值，即push，也可以在容器非空的时候查看队尾的数值，即back；队头只能删除值，还可以在容器非空的时候查看队头的数值，即front。所有的操作只能在两端进行。由于这个特殊性质，栈被称为是先进先出的，即LILO或者FIFO，即Last in last out或者First in first out。

队列的实现，可以用循环数组或者双链表。数组在尾端添加、删除都是非常快的，但是在头部添加删除则很慢，可以用循环数组的方式，只需要用两个指针标记一下队头和队尾就行了，这样就不用挪动元素；而双链表的表头、尾进行添加、删除都是非常快的，一个做队头一个做队尾就行。具体实现这里省略，大家可以自行实现。

例题：https://blog.csdn.net/qq_46105170/article/details/113801397

队列的应用

队列在主要是在广度优先搜索（BFS）中应用很多，在图论算法中，广度优先搜索使用队列来记录需要访问的节点。通过队列，算法能够按照层次顺序遍历节点，广泛应用于最短路径搜索、连通性检测等。

队列在操作系统中也应用很多，但是和算法的关系相对于栈的应用小很多，这里省略。

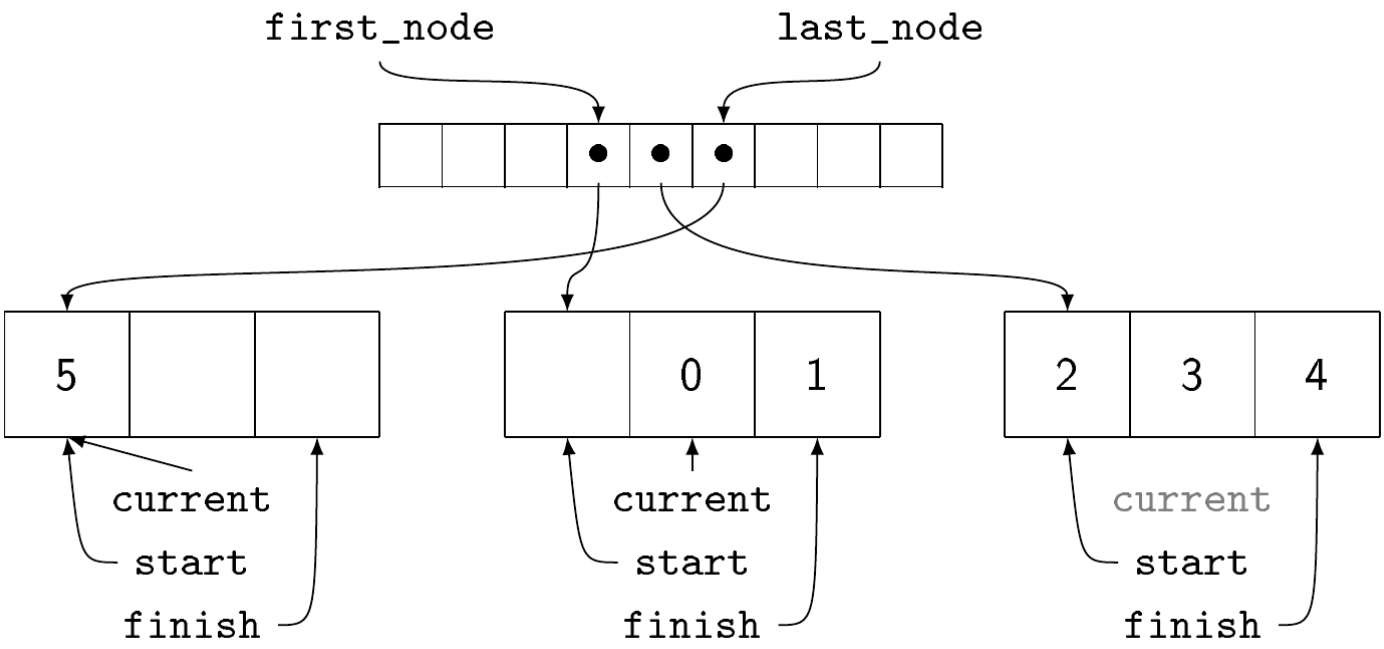
双端队列

顾名思义，双端队列就是两端都可以添加、删除、查看元素的线性数据结构。栈和队列都是双端队列的特例。在算法中，双端队列主要出现在单调队列和01图最短路的求解中，其他地方用的不多。以后都会涉及。

C++ STL中的栈、队列、双端队列

C++ STL里的栈和队列都是通过双端队列实现的。而C++的双端队列的实现非常复杂，但是非常经典，其兼顾了时间空间的效率。与Java的Deque不同的是，C++的deque还支持像数组那样的随机访问，可以说是功能相当强大。

```
std::deque<int> dq{0, 1, 2, 3, 4, 5};
```



下面将详细解释 `std::deque` 在 C++ 中的实现原理。以下来自chatgpt。

1. 总体结构

`std::deque` 的内部结构可以视为一块 **映射表（map）**，指向一系列 **固定大小的缓冲区（block 或称作缓冲区块）**。这些缓冲区连续地存储元素，但整体上不要求所有元素在内存中连续。

- **映射表（map）**：这是一个指针数组，每个指针指向一个缓冲区块。
- **缓冲区块（block）**：每个块包含固定数量的元素，块的大小通常是固定的（例如 512 或 1024 个元素），这取决于实现。

这种结构类似于 **分段式连续存储**，既避免了大规模连续内存分配的麻烦，又能够支持高效的随机访问。

2. 插入和删除操作的效率

- **两端高效插入/删除**：由于映射表可以在两端扩展，当需要在队列两端插入或删除元素时，只需在映射表的两端增加或移除块的指针，无需移动大量元素。
- **中间插入/删除**：在队列中间插入或删除元素时，可能需要移动相关的元素，但由于元素被分散在多个块中，实际需要移动的元素数量相对较少。

3. 随机访问的实现

`std::deque` 支持常数时间的随机访问，这是通过双重索引实现的：

- **第一层索引（块索引）**：通过计算元素的全局索引除以每个块的大小，确定元素所在的块。
- **第二层索引（块内偏移）**：通过计算元素的全局索引模上块大小，确定元素在块内的偏移位置。

这种方式使得随机访问的时间复杂度为 O(1)，尽管元素不在物理内存中连续存储。

4. 与 `std::vector` 和 `std::list` 的比较

- `std::vector`：底层是动态数组，所有元素在内存中连续存储，支持高效的随机访问和尾部插入/删除，但在头部插入/删除元素效率低下，因为需要移动大量元素。
- `std::list`：底层是双向链表，元素不连续存储，支持在任意位置高效地插入/删除，但随机访问的效率低下（O(n) 时间复杂度）。
- `std::deque`：结合了两者的优点，支持两端高效的插入/删除操作，同时提供常数时间的随机访问。

5. 内存管理

- **内存分配**：`std::deque` 通过分配多个小的缓冲区块，避免了分配一大片连续内存的需求。这在处理大量元素时，更加节省内存并减少内存碎片。
- **块的重用**：当元素被删除后，相应的块可能被保留以供将来使用，减少频繁的内存分配和释放操作。

6. 具体实现细节

以下是 `std::deque` 实现的关键细节：

- **块大小（Buffer Size）**：块的大小通常是编译器或标准库实现中预先定义的，可能根据元素类型的大小进行调整，以优化内存使用和对齐。
- **映射表的扩展**：当需要在两端插入元素且当前映射表已满时，映射表本身会进行动态扩展，类似于 `std::vector` 的容量增长，但增长频率较低。
- **指针的高效更新**：由于映射表存储的是块的指针，插入和删除操作只需更新映射表和块内的指针，而不需要移动实际的元素数据。

7. 操作的时间复杂度

- **随机访问**：O(1)
- **两端插入/删除**：摊销 O(1)
- **中间插入/删除**：O(n)，但由于块的存在，实际移动的元素可能少于 `std::vector`。

8. 代码示意

以下是一个简化的 `std::deque` 结构示意：

```
[Map (Pointer Array)]
|     |     |     |     |     |
 v     v     v     v     v     v
[Block][Block][Block][Block][Block]

每个 Block 包含固定数量的元素。
```

当进行元素访问时：

1. **计算块索引**：`block_index = index / block_size`
2. **计算块内偏移**：`offset = index % block_size`
3. **访问元素**：`element = map[block_index][offset]`

9. 实现的差异性

需要注意的是，`std::deque` 的具体实现可能因编译器和标准库的不同而有所差异：

- **GNU 标准库（libstdc++）**：采用分段数组的实现方式，块大小和映射表的增长策略可能不同。
- **LLVM 标准库（libc++）**：实现细节可能有所调整，以优化特定场景下的性能。

作业

所有例题