

数据结构实验 (3)

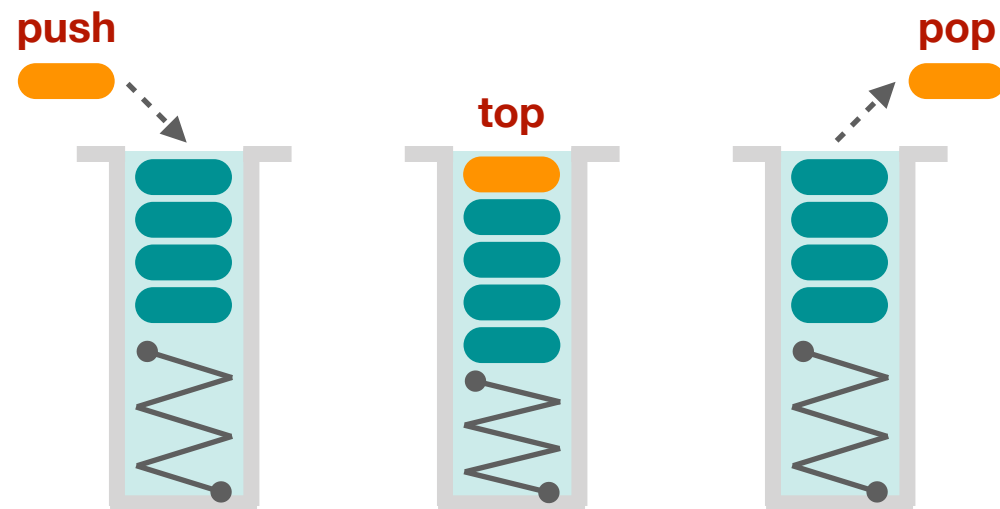
栈和队列

目录

- 栈与队列
 - 栈与队列的定义与实现
 - 栈与队列的应用
 - 函数栈与括号匹配问题
 - 队列与轮训调度
 - 深度优先与广度优先遍历算法
 - 栈与队列的扩展
 - 优先队列
 - 栈 + get_max()
 - 使用栈模拟队列

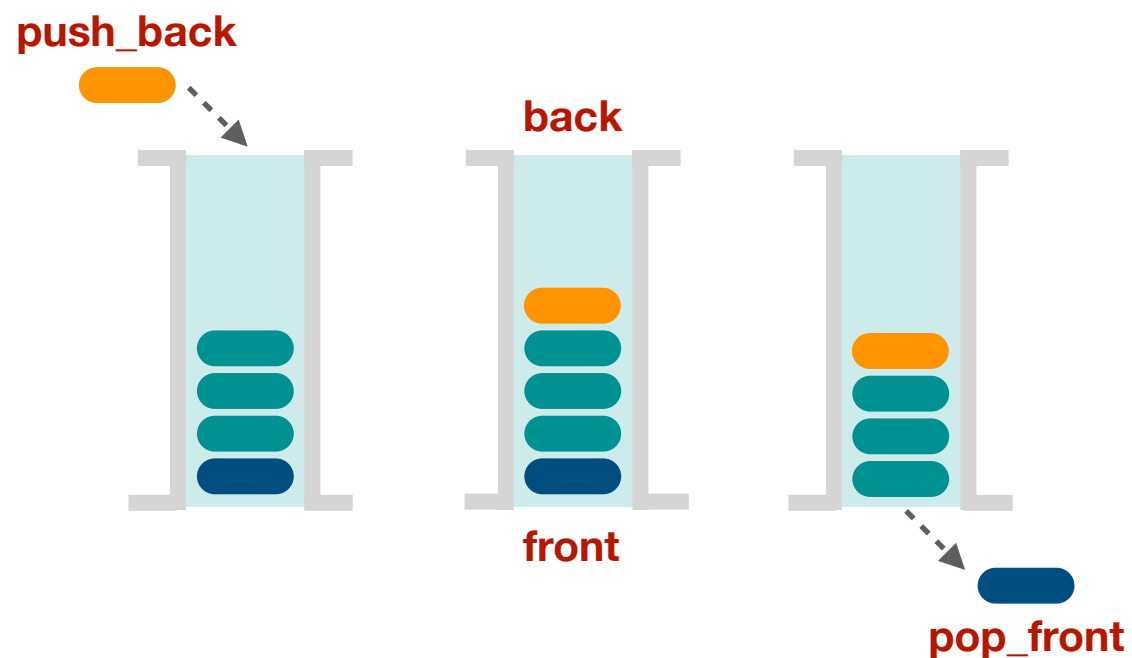
- 栈与队列的定义与实现

- 栈将数据互相垛（stack, vt.& vi. 叠、垛）在一起；
- 新的数据可以垛（push）在栈顶，也可以从栈顶取走（pop）数据；
- 能获取栈顶的元素（top）
- 能获取最大元素吗？(max)



- 栈与队列的定义与实现

- 队列将数据前后排（queue, vi. 排队等候）在一起；
- 新的数据可以排（**push_back**）在队列末尾，也可以从队列开头取走（**pop_front**）数据；
- 能获取队列开头（**front**）和（**back**）的元素



目录

- 栈与队列
 - 栈与队列的定义与实现
 - 栈与队列的应用
 - 函数栈与括号匹配问题
 - 队列与轮训调度
 - 深度优先与广度优先遍历算法
 - 栈与队列的扩展
 - 优先队列
 - 栈 + get_max()
 - 使用栈模拟队列

- 栈与队列的定义与实现
 - 栈与括号匹配问题

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

1) 传统递归思路:

I. 问题边界: 无括号的表达式是匹配的

II. 递归策略: 表达式 匹配, 仅当 (表达式) 匹配

这只是表达时匹配的充分条件

2) 传统思路存在反例: $(()) () = (()) ()$

- 栈与队列的定义与实现
 - 栈与括号匹配问题

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

消去紧邻的 匹配括号 不影响剩余表达式的判断

1) 改进递归思路:

I. 问题边界: 无括号的表达式是匹配的

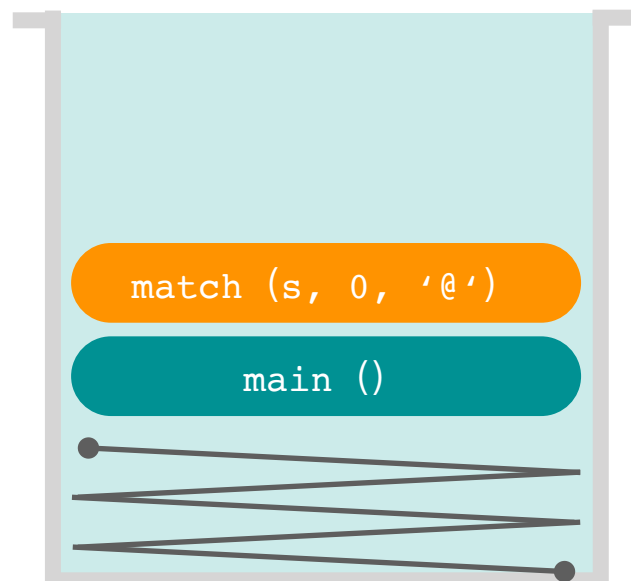
II. 递归策略: 表达式A () 表达式B 匹配, 仅当 表达式A 表达式B 匹配

III. 遍历表达式, 遇到左括号时, 进入子问题: 寻找紧邻的右括号

- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"

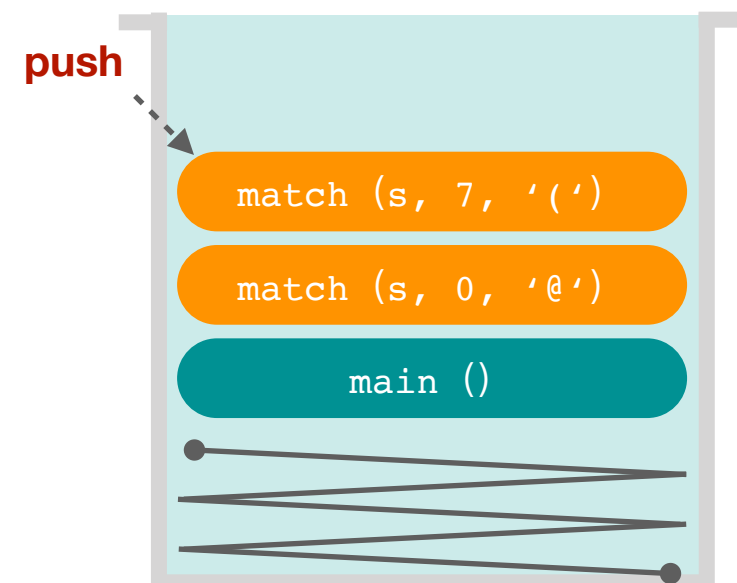

↑



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"

find:), " { if (x * (y + z[1]) < 137) { x = 1; } }"



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

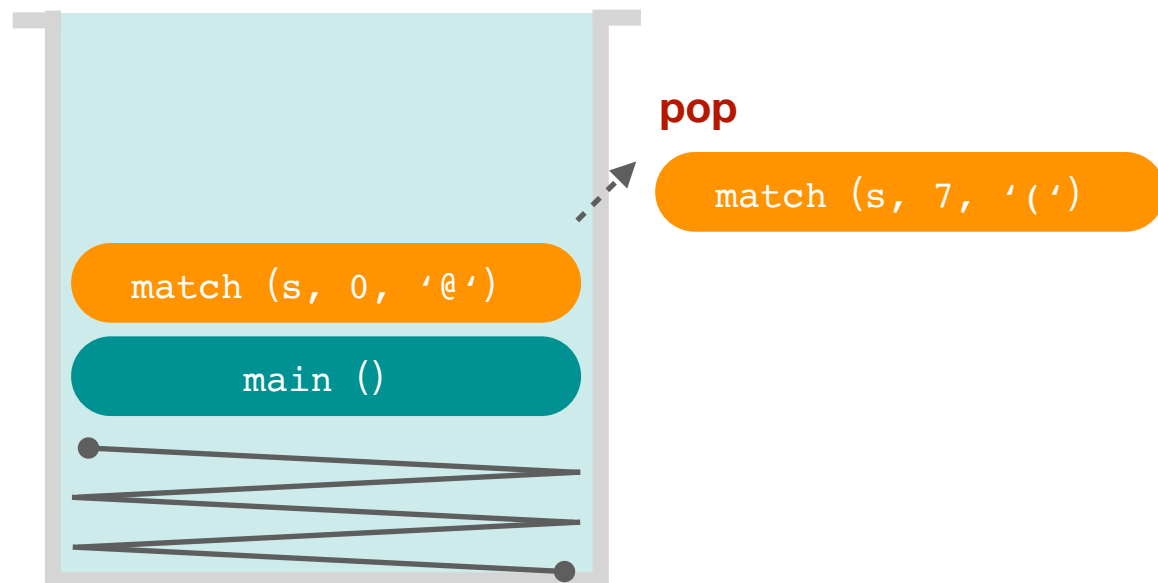
match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"

↑

find:), ") { if (x * (y + z[1]) < 137) { x = 1; } }"

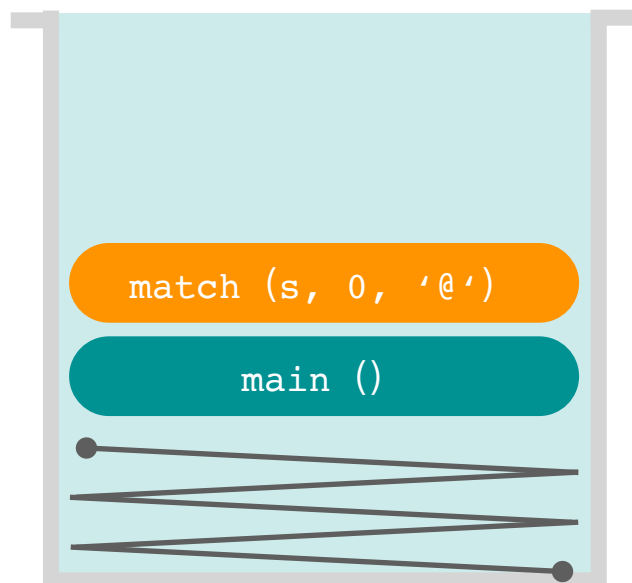
↑

匹配成功!



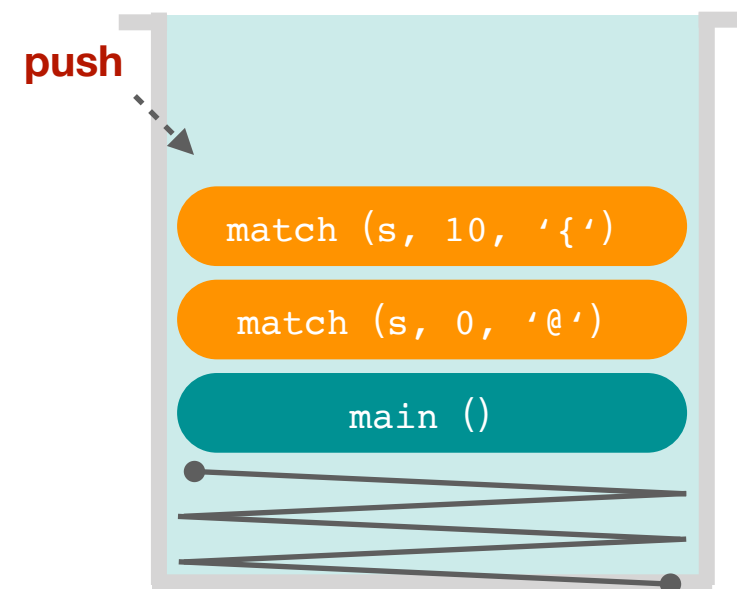
- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

```
match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"  
      ↑  ↑  
find:  ), ")" { if (x * (y + z[1]) < 137) { x = 1; } }"
```



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

```
match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"
      ↑
find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"
      ↑
```



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

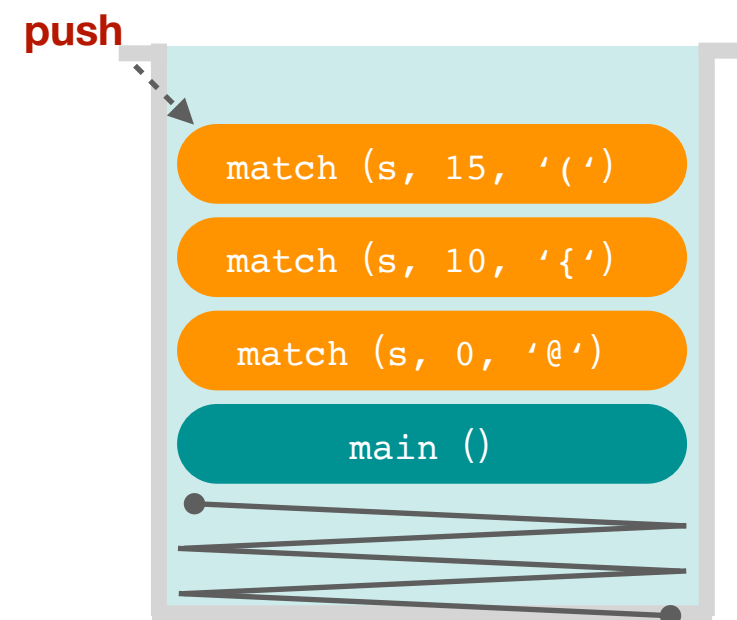
match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



find:), "x * (y + z[1]) < 137) { x = 1; } }"



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



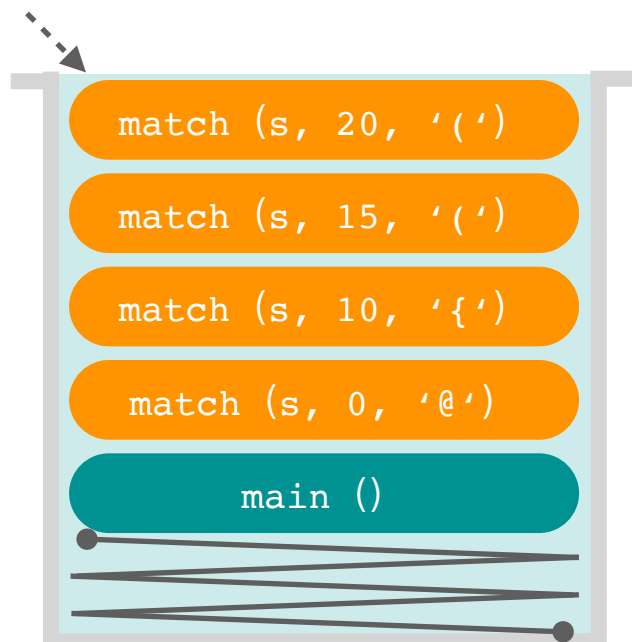
find:), "x * (y + z[1]) < 137) { x = 1; } }"



find:), "y + z[1]) < 137) { x = 1; } }"



push



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

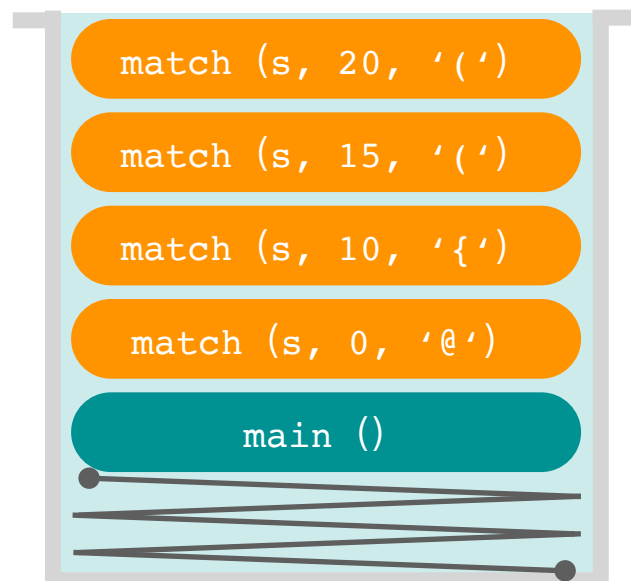
match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"

find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"

find:), "x * (y + z[1]) < 137) { x = 1; } }"

find:), "y + z[1]) < 137) { x = 1; } }"

匹配成功!



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



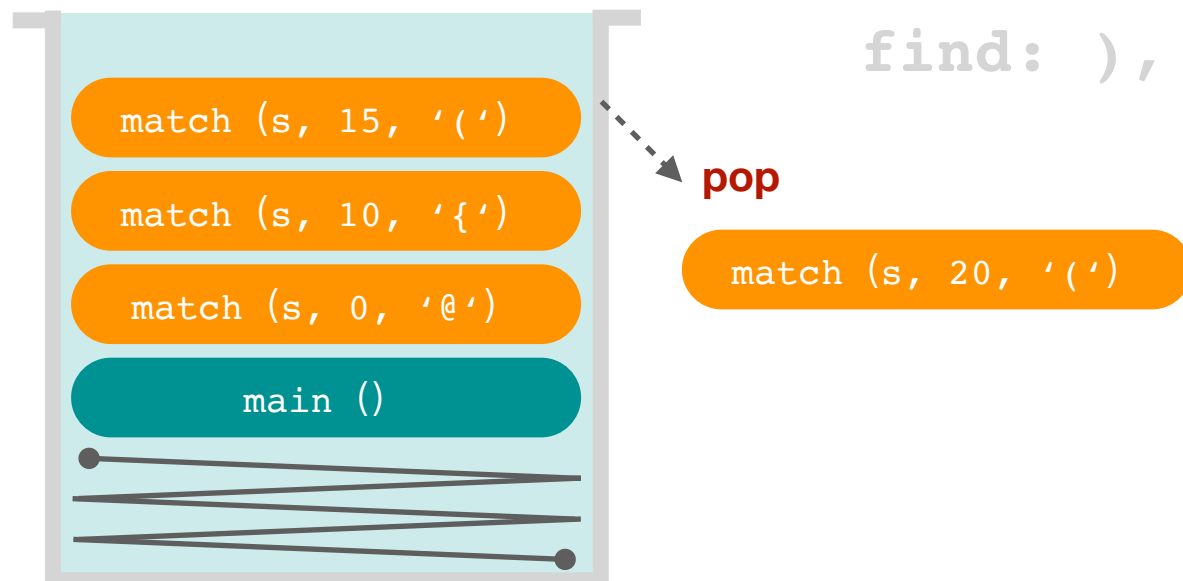
find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



find:), "x * (y + z[1]) < 137) { x = 1; } }"



find:), "y + z[1]) < 137) { x = 1; } }"



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



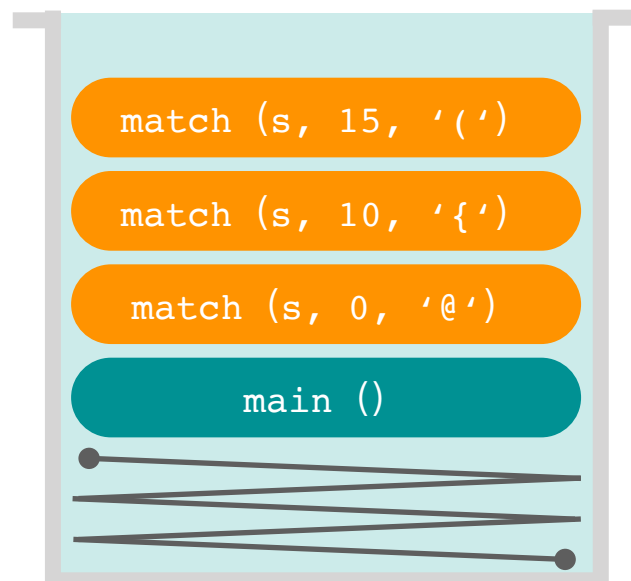
find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



find:), "x * (y + z[1]) < 137) { x = 1; } }"



匹配成功!



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

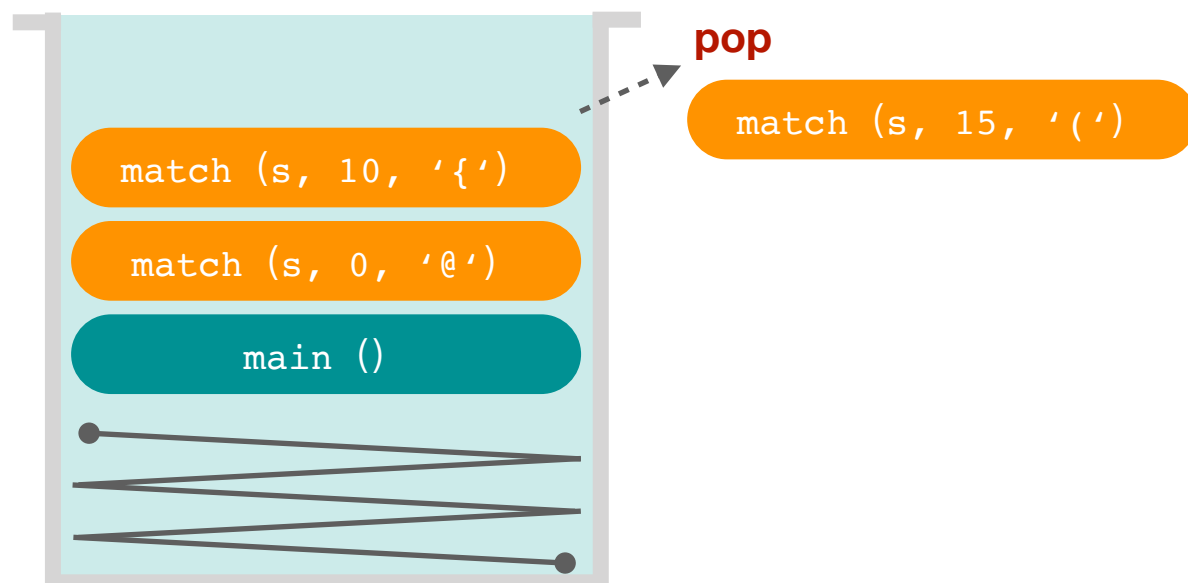
match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



find:), "x * (y + z[1]) < 137) { x = 1; } }"



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



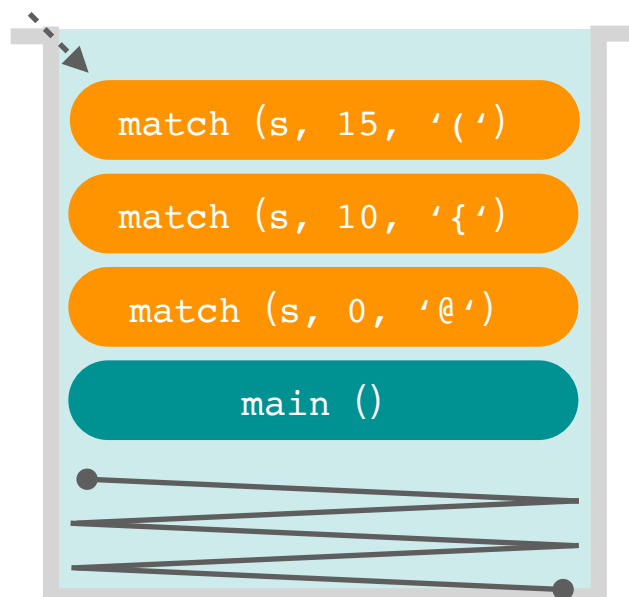
find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



find: }, "x = 1; } }"



push



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



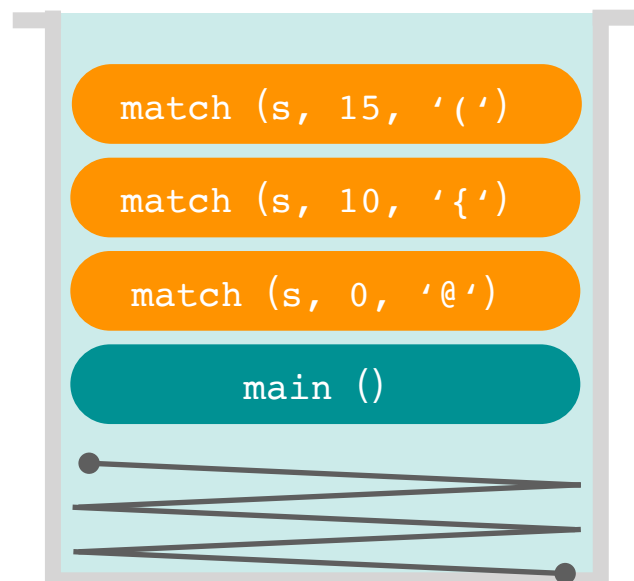
find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



find: }, "x = 1; } }"



匹配成功!



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

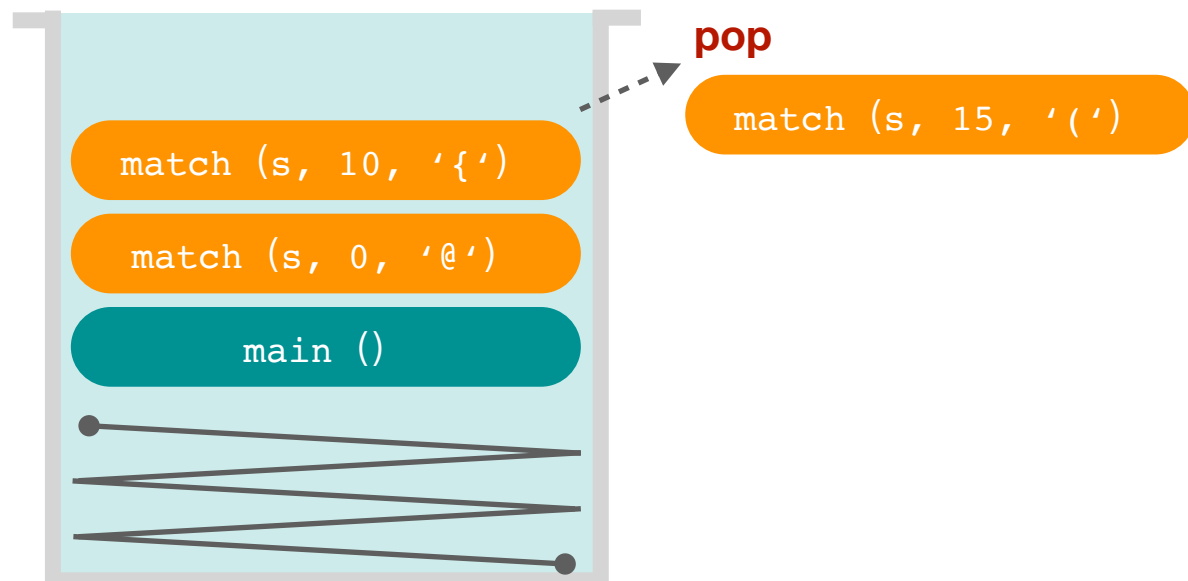
match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



find: }, "x = 1; } }"



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

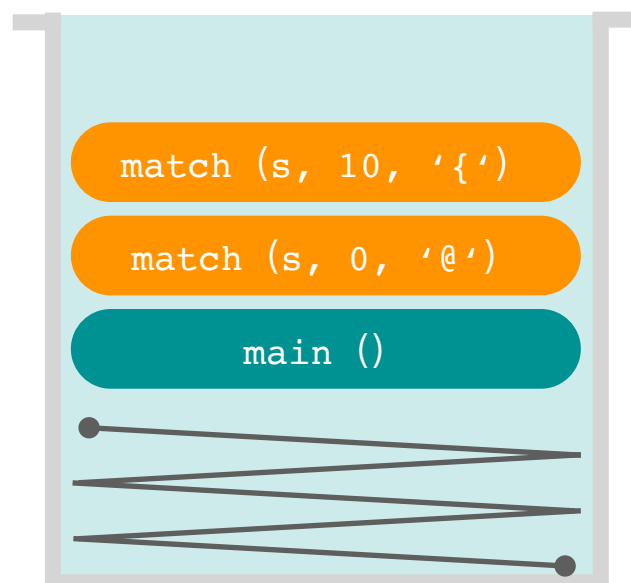
match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"



find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



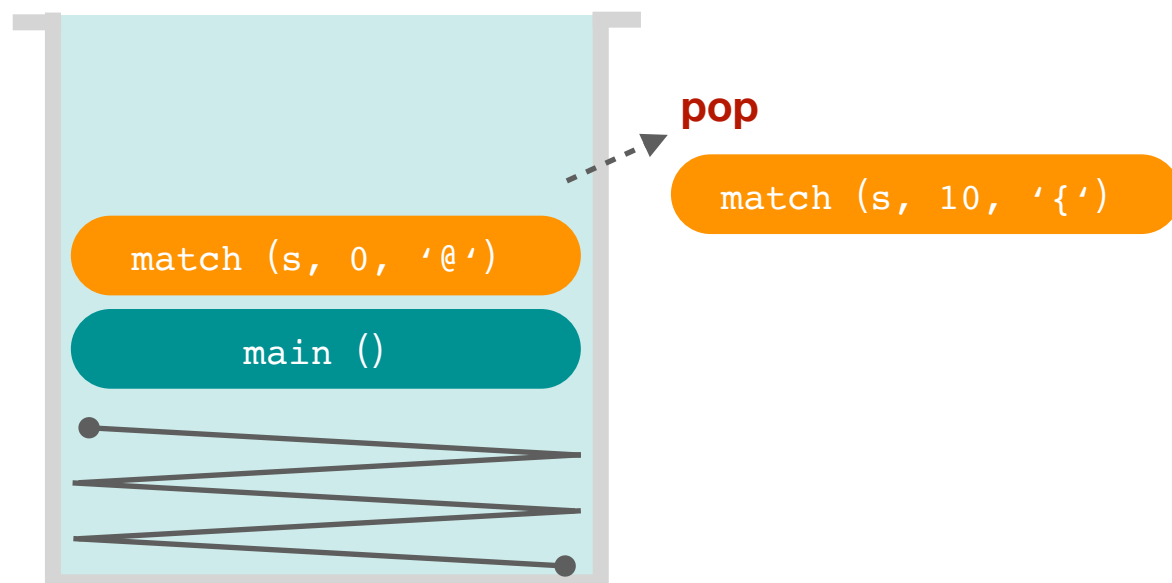

匹配成功!



- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

match: "int foo() { if (x * (y + z[1]) < 137) { x = 1; } }"

find: }, "if (x * (y + z[1]) < 137) { x = 1; } }"



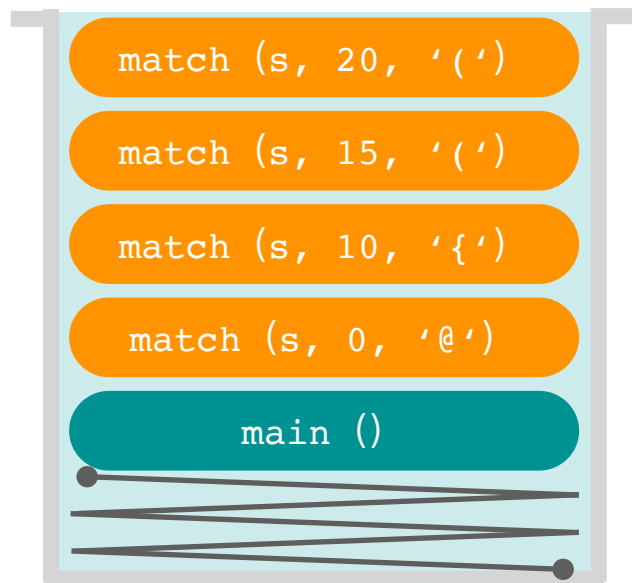
- 栈与队列的定义与实现
 - 栈与括号匹配问题
 - 回顾递归思想

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

算法：

- 1) 定义问题 `match()`，遍历在字符串，并判断括号是否匹配
 - I. 问题边界：已经遍历到字符串结尾，结束，并返回**匹配成功**；
 - II. 遍历字符串，遇到左括号 `(`, `[` 或 `{`，进入子问题 2)，寻找对应的右括号；若寻找成功，则继续遍历，否则，**匹配失败**；
- 2) 定义子问题 `find()`，遍历在字符串，并寻找匹配的括号：`(=>)`，`[=>]`，`{ => }`
 - 1) 问题边界：已经遍历到字符串结尾，结束，并返回**匹配失败**；
 - 2) 遍历字符串，遇到对应右括号，则返回**匹配成功**，及当前遍历坐标；若遇到左括号 `(`, `[` 或 `{`，则进入子问题 2)，寻找对应右括号；

- 栈与队列的定义与实现
- 栈与括号匹配问题
 - 递归与函数栈



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

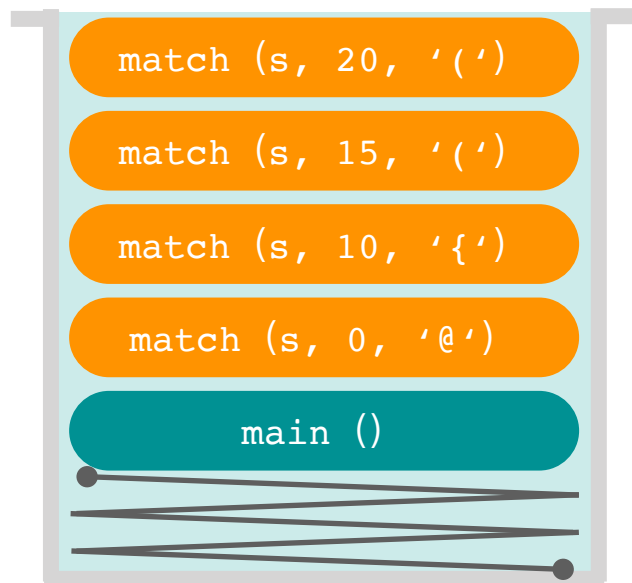
int match(const std::string& str, int pos, char c) {
    for (int i = pos; i < str.length(); ++i) {
        int j = 0;
        switch (str[i]) {
            // 遍历到左括号, 则递归解决子问题
            case '[':
            case '{':
            case '(':
                j = match(str, i+1, str[i]);
                if (j > 0) {
                    i = j;
                    continue;
                } else {
                    return -i;
                }
            // 遍历到右括号, 则判断是否属于匹配的括号
            case ']':
                if (c == '[') { return i; } else { return -i; }
            case '}':
                if (c == '{') { return i; } else { return -i; }
            case ')':
                if (c == '(') { return i; } else { return -i; }
        }
    }
    // 用 '@' 标记是最外层的遍历
    if (c == '@') {
        return str.length();
    } else {
        return -str.length();
    }
}

int main(int, char**) {
    std::string st = "{ if (x * (y + z[1]) < 137) { x = 1; } }";
    std::cout << match(st, 0, '@') << std::endl;
}
```

- 栈与队列的定义与实现

- 栈与括号匹配问题

- 递归与函数栈



* 动机：编程语言的函数站，为**隐式**维护的栈，需花费额外的空间、时间

- 额外开销的大小取决于**递归深度**

* 方法：

- **显式地**维护调用栈
- 将递归算法改写为遍历+栈

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

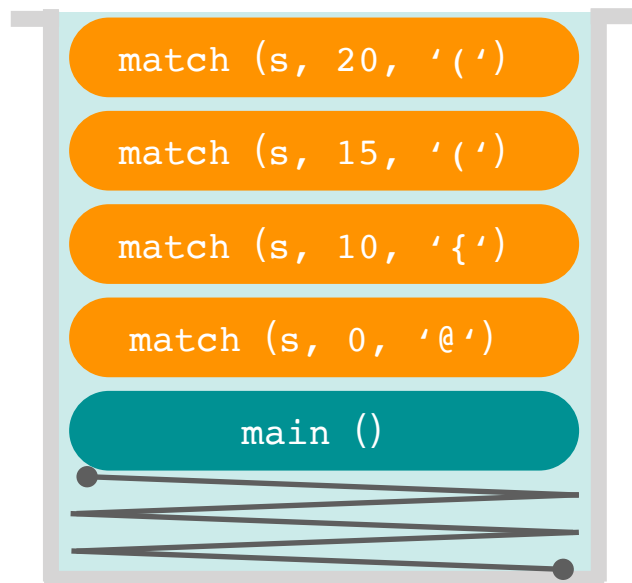
```
int match(const std::string& str, int pos, char c) {
    for (int i = pos; i < str.length(); ++i) {
        int j = 0;
        switch (str[i]) {
            // 遍历到左括号，则递归解决子问题
            case '[':
            case '{':
            case '(':
                j = match(str, i+1, str[i]);
                if (j > 0) {
                    i = j;
                    continue;
                } else {
                    return -i;
                }
            // 遍历到右括号，则判断是否属于匹配的括号
            case ']':
                if (c == '[') { return i; } else { return -i; }
            case '}':
                if (c == '{') { return i; } else { return -i; }
            case ')':
                if (c == '(') { return i; } else { return -i; }
        }
    }
    // 用 '@' 标记是最外层的遍历
    if (c == '@') {
        return str.length();
    } else {
        return -str.length();
    }
}

int main(int, char**) {
    std::string st = "{ if (x * (y + z[1]) < 137) { x = 1; } }";
    std::cout << match(st, 0, '@') << std::endl;
}
```

- 栈与队列的定义与实现

- 栈与括号匹配问题

- 递归与函数栈



* 动机：编程语言的函数站，为**隐式**维护的栈，需花费额外的空间、时间

- 额外开销的大小取决于**递归深度**

* 方法：

- **显式地**维护调用栈
- 将递归算法改写为遍历+栈

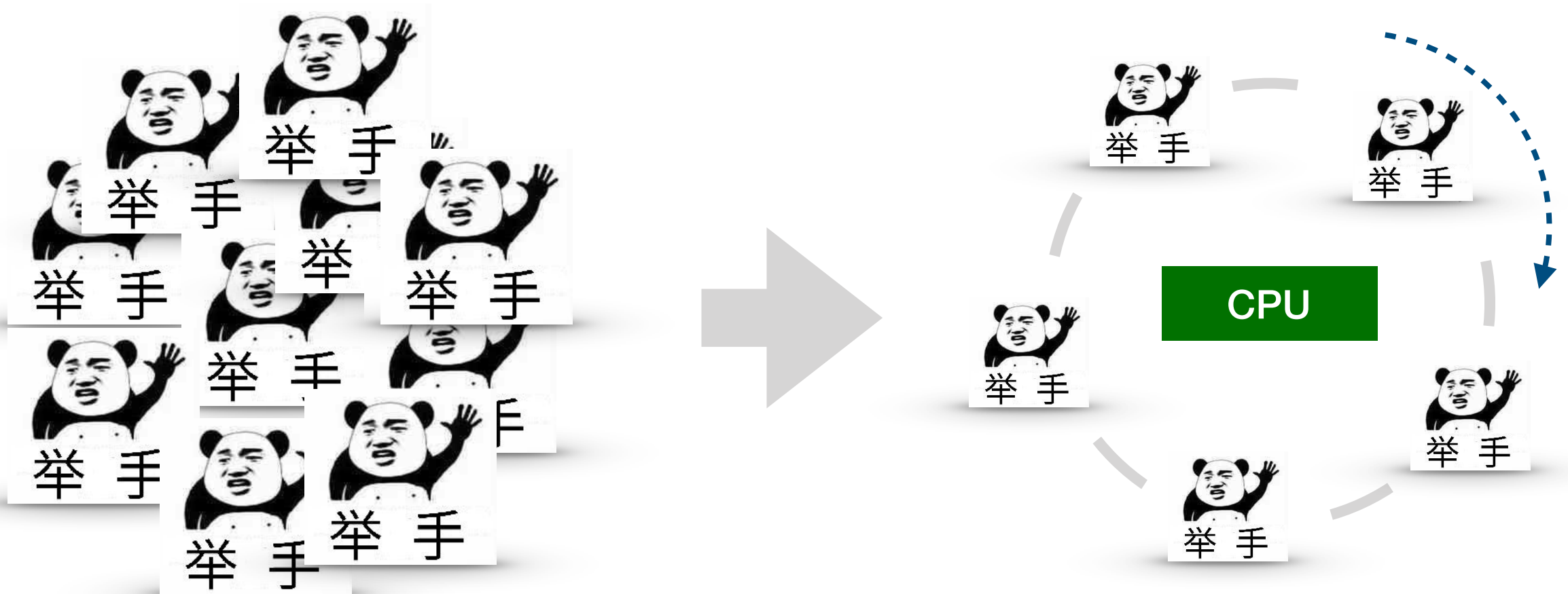
```
#include <string>
#include <stack>

bool match_stack(const std::string& str) {
    std::stack<char> st;
    for (char ch : str) {
        switch (ch) {
            case '[':
            case '{':
            case '(':
                st.push(ch);
                continue;
            case ']':
                if (st.size() > 0 && st.top() == '[') {
                    st.pop();
                    continue;
                } else {
                    return false;
                }
            case '}':
                if (st.size() > 0 && st.top() == '{') {
                    st.pop();
                    continue;
                } else {
                    return false;
                }
            case ')':
                if (st.size() > 0 && st.top() == '(') {
                    st.pop();
                    continue;
                } else {
                    return false;
                }
        }
    }
    if (st.size() == 0) {
        return true;
    } else {
        return false;
    }
}
```

目录

- 栈与队列
 - 栈与队列的定义与实现
 - 栈与队列的应用
 - 函数栈与括号匹配问题
 - 队列与轮训调度
 - 深度优先与广度优先遍历算法
 - 栈与队列的扩展
 - 优先队列
 - 栈 + get_max()
 - 使用栈模拟队列

- 栈与队列的应用 (2)
 - 队列与多线程编程
 - 多个使用者共享同一资源时，如何进行资源分配？
 - 例如：多线程程序在单一 CPU 上的执行；多个 IO 请求在磁盘执行；多个 Web 服务程序相应网络请求



- 栈与队列的应用 (2)

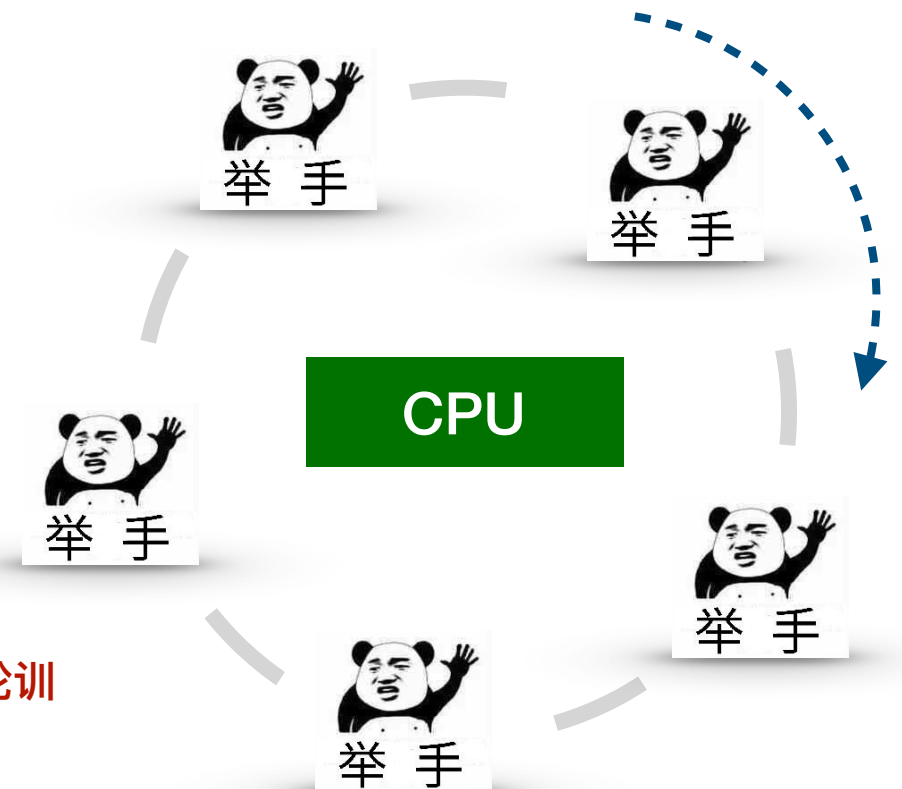
- 队列与多线程编程

问题：如果有多个资源可以被共享？
多个 CPU 执行程序？

- 多个使用者共享同一资源时，如何进行资源分配？

- 例如：多线程程序在单一 CPU 上的执行；多个 IO 请求在磁盘执行；多个 Web 服务程序相应网络请求

```
// 轮询调度算法(Round-Robin Scheduling)
// https://en.wikipedia.org/wiki/Round-robin\_scheduling
RoundRobin {
    // 初始化队列，并将共享资源的所有使用者插入队列中
    Queue q(clients);
    // 反复执行任务，直到在服务终止
    while !serviceTerminated() {
        client = q.pop(); // 删除并获取队列首的使用者
        serve(client.task); // 执行任务
        q.push(client); // 将使用者放入队列末尾，等待下一次轮训
    }
}
```

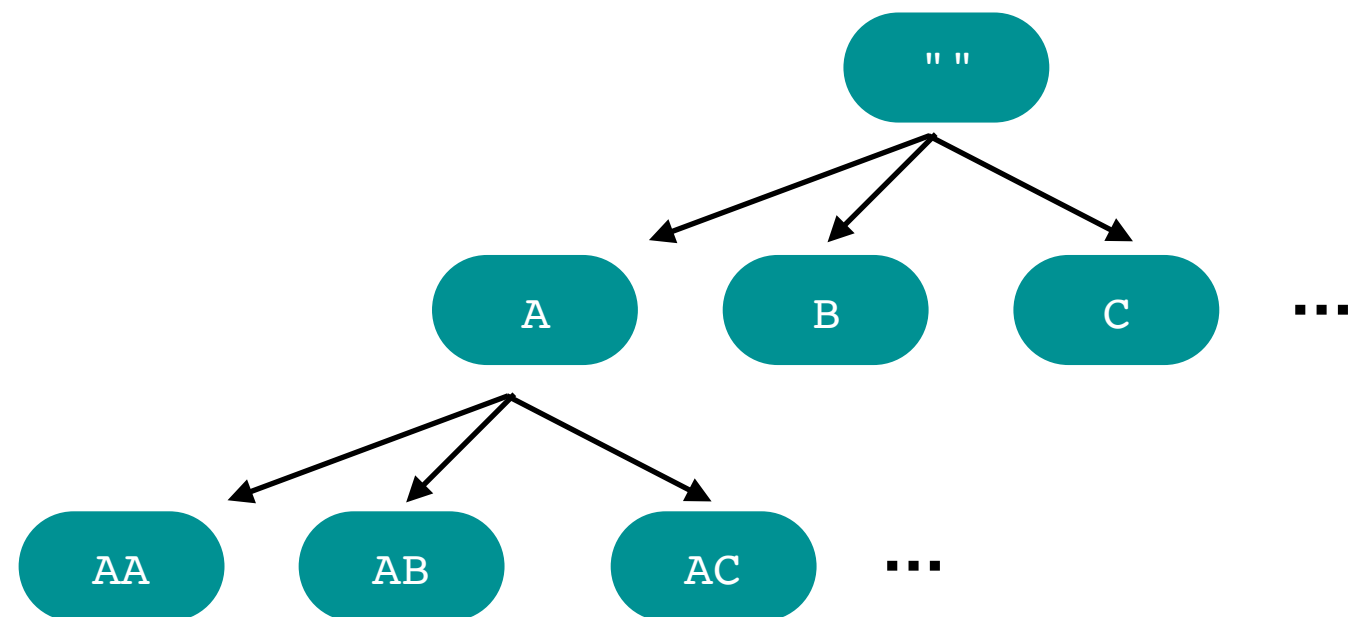


目录

- 栈与队列
 - 栈与队列的定义与实现
 - 栈与队列的应用
 - 函数栈与括号匹配问题
 - 队列与轮训调度
 - 深度优先与广度优先遍历算法
 - 栈与队列的扩展
 - 优先队列
 - 栈 + get_max()
 - 使用栈模拟队列

- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

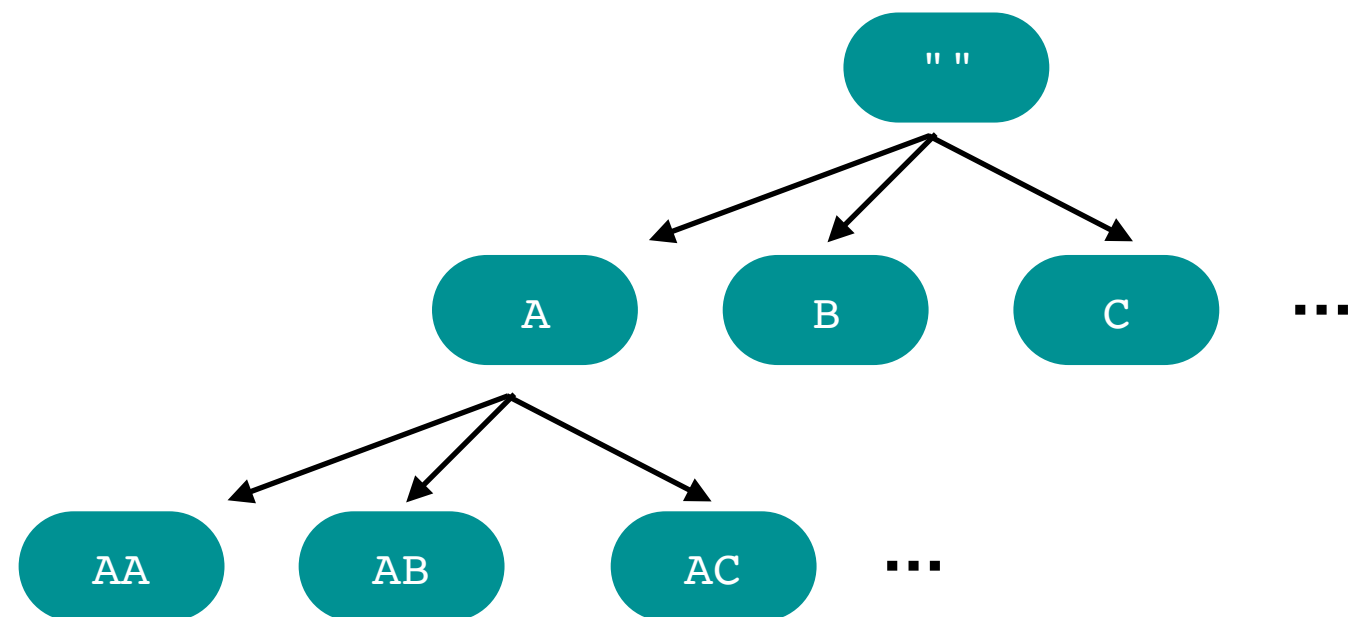
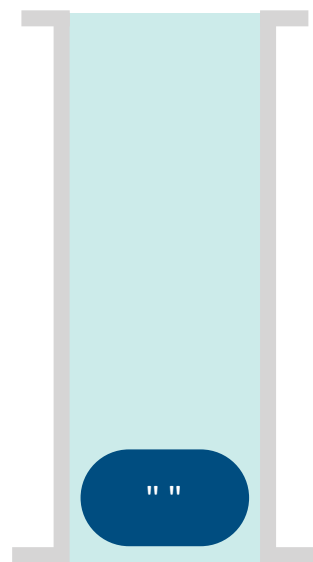
"" | A | B | C | ... | AA | AB | AC | ... | BA | BB | BC | ... | CA | CB | ...



- 栈与队列的应用 (3)

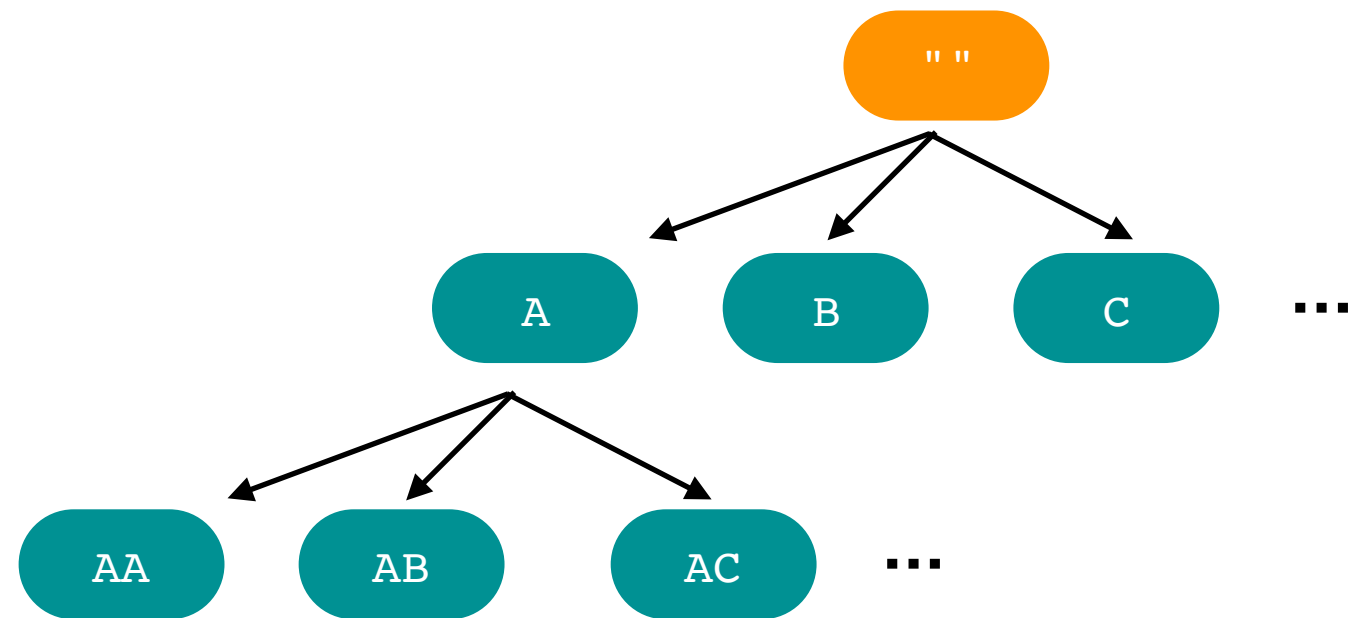
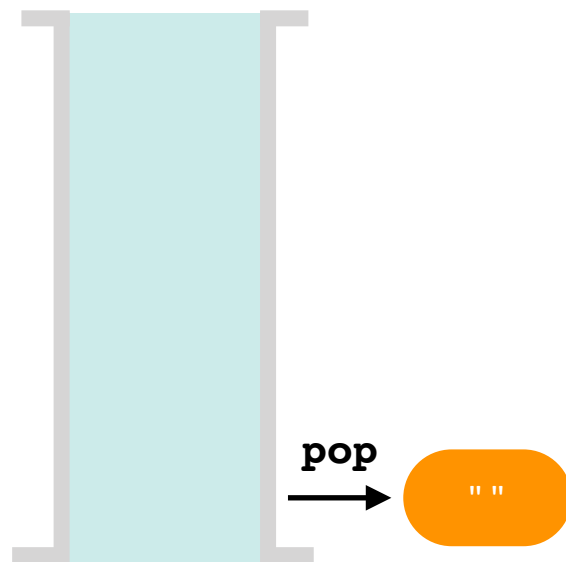
- 字典生成问题：生成不超过指定长度的所有字母串

"" | A | B | C | ... | AA | AB | AC | ... | BA | BB | BC | ... | CA | CB | ...

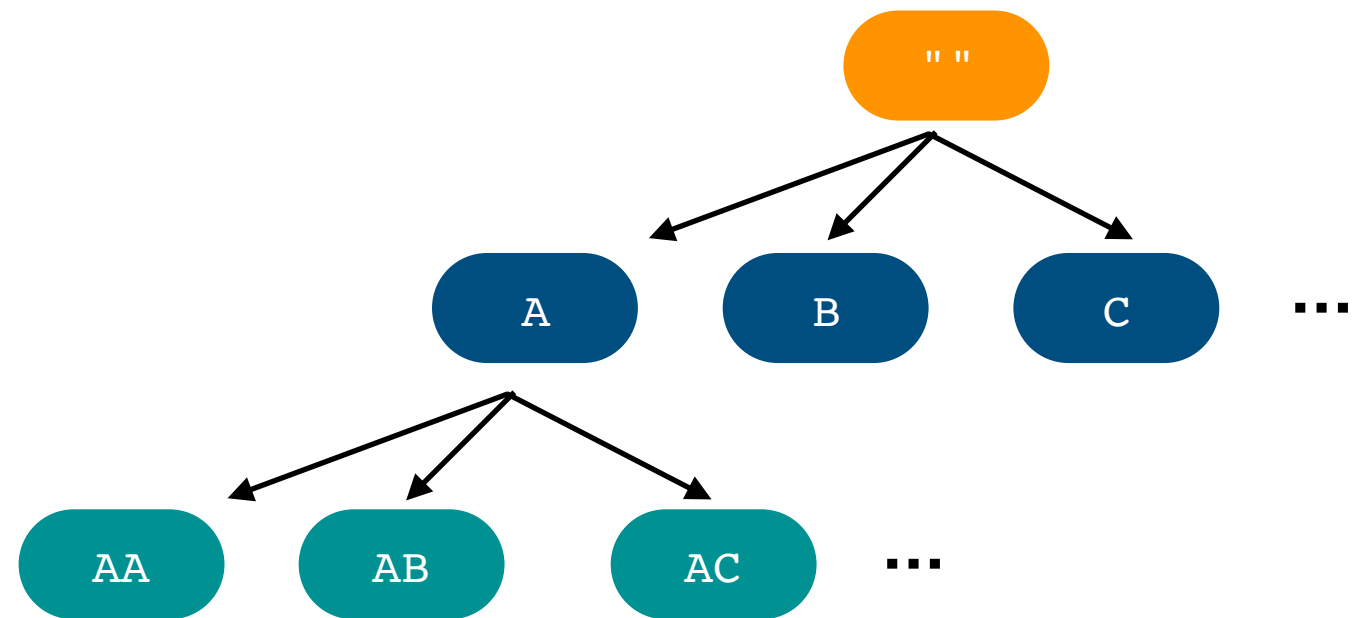
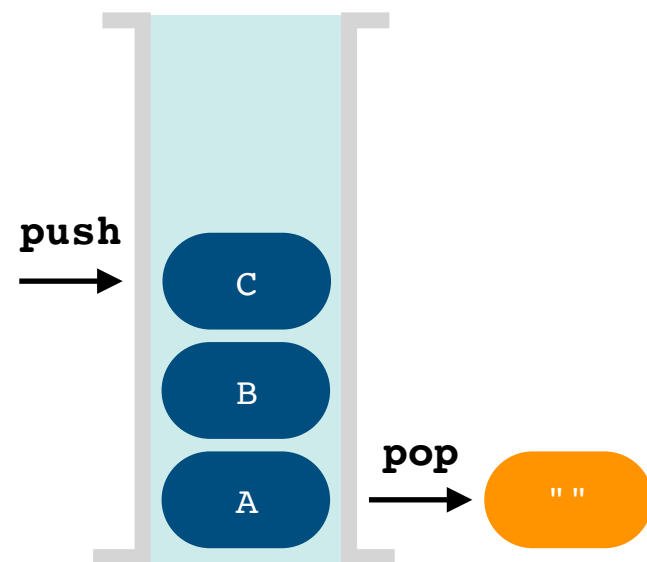


- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

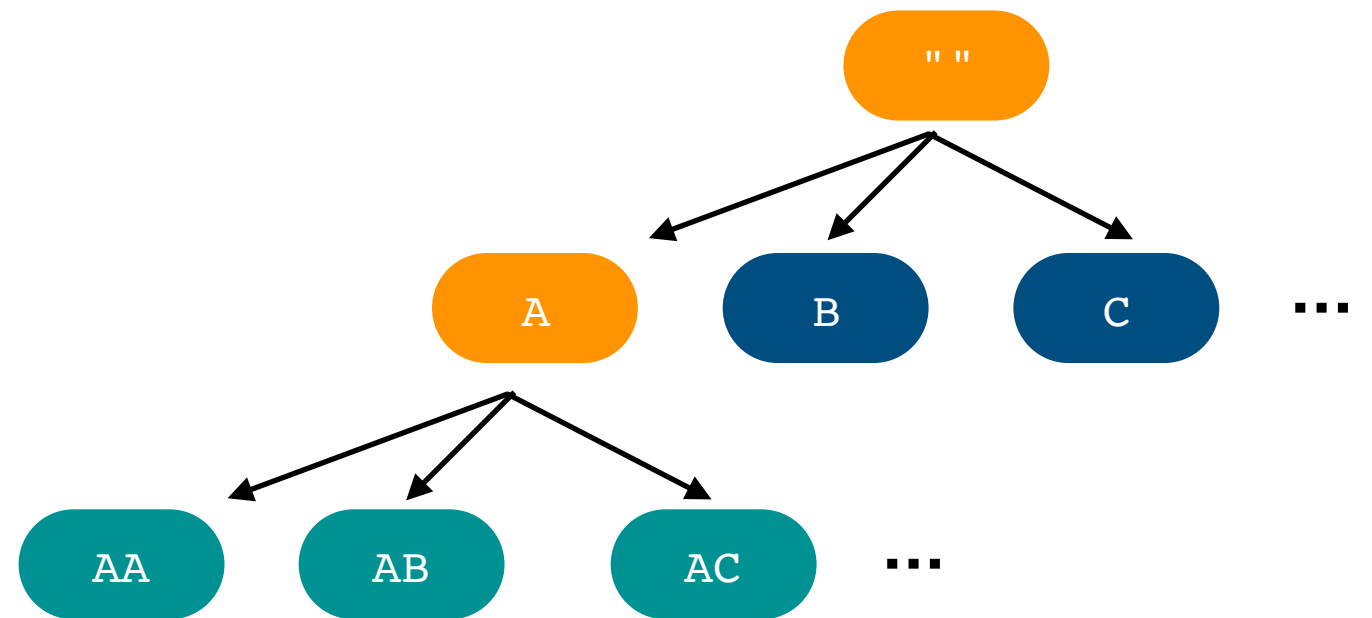
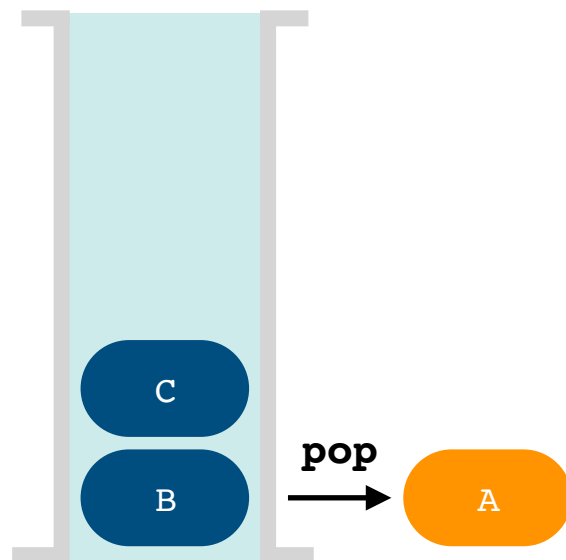
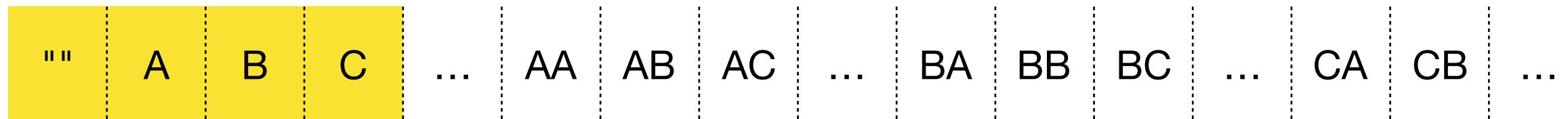
"" A B C ... AA AB AC ... BA BB BC ... CA CB ...



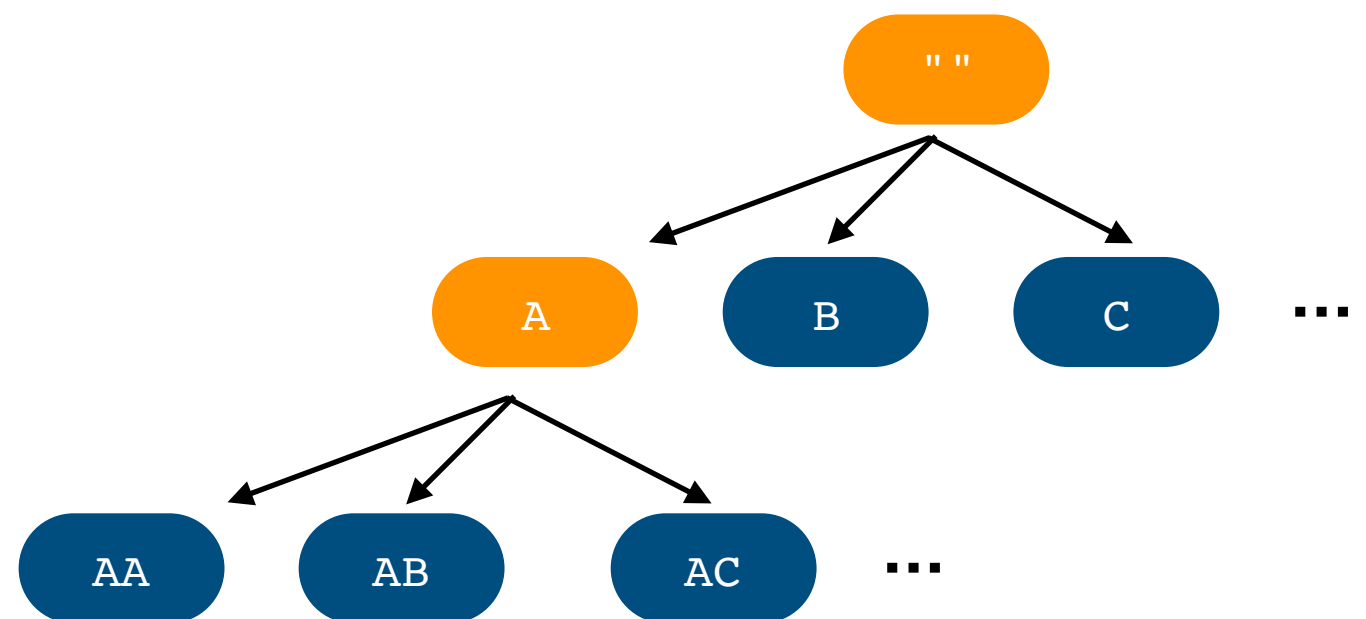
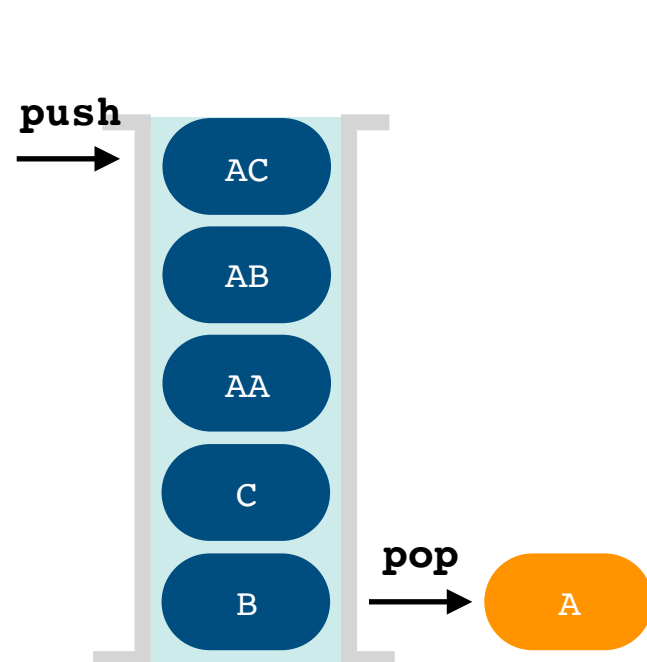
- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串



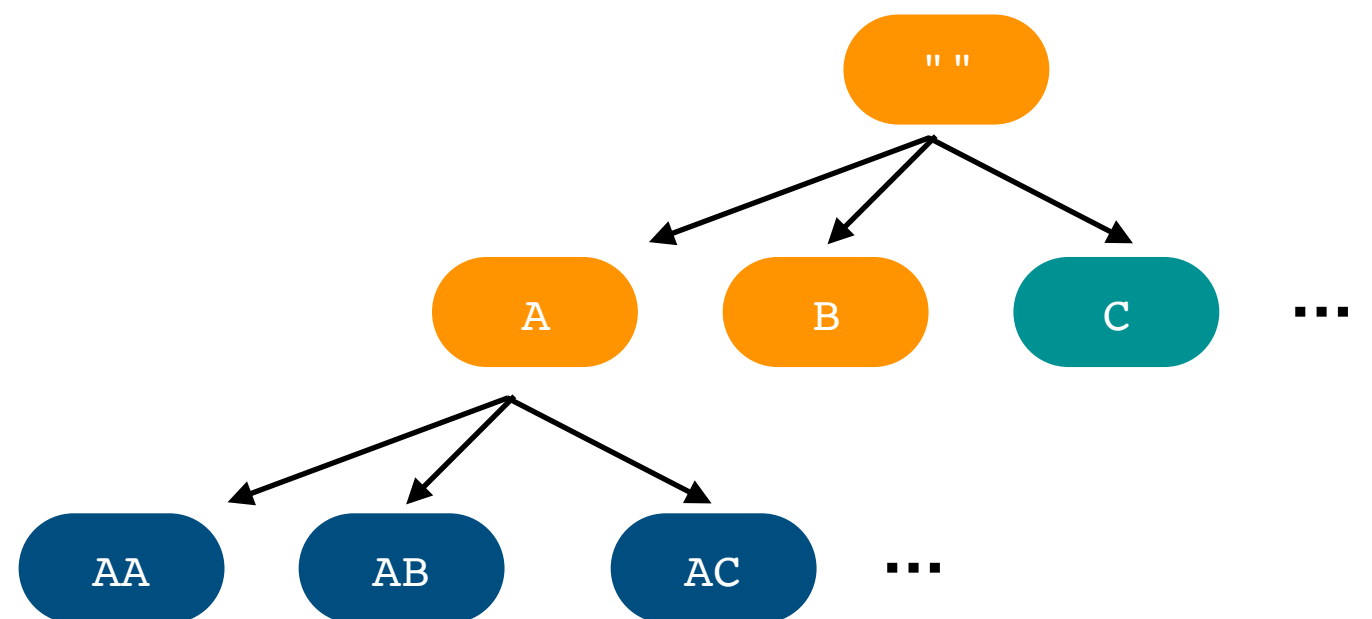
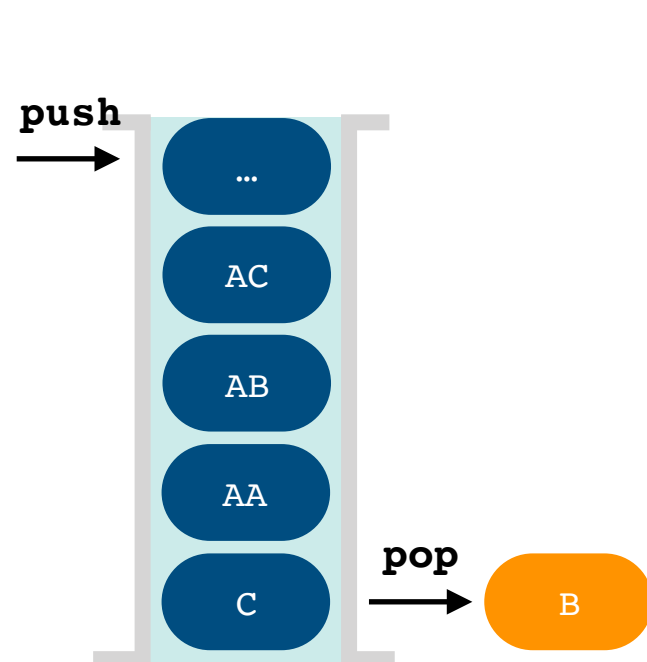
- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串



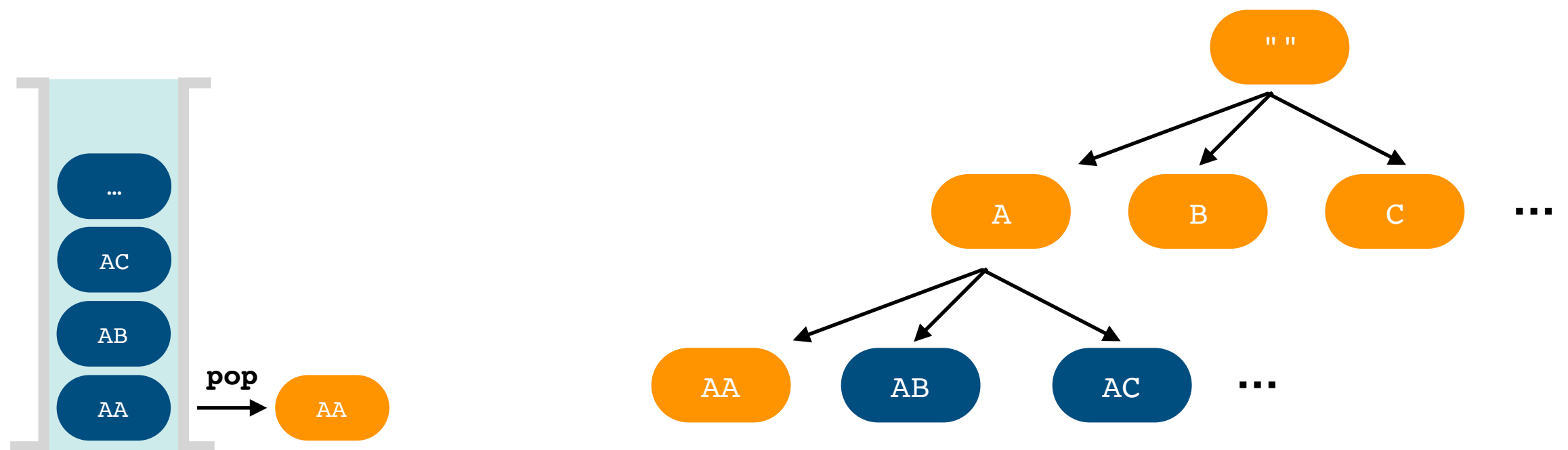
- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串



- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串



- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串



AA 已经达到指定长度限制，不再往队列中插入

- 栈与队列的应用 (3)

- 字典生成问题：生成不超过指定长度的所有字母串

" " | A | B | C | ... | AA | AB | AC | ... | BA | BB | BC | ... | CA | CB | ...

// 字符串生成算法

```
StringGenerate(candidates, length_limit) {
```

```
    // 初始化队列
```

```
    Queue q();
```

```
    // 将长度为0的空字符串插入队列
```

```
    q.push("");
```

```
    // 若队列不为空，则不断使用队首元素生成新的字符串，并插入队列
```

```
    while q.size() > 0 {
```

```
        // 删除并获取队列首字符串
```

```
        str = q.pop();
```

```
        print(str);
```

```
        if str.length() < length_limit {
```

```
            // 若字符串长度符合要求，则遍历所有候选字母
```

```
            for char in candidates {
```

```
                // 生成新的字符串，并插入队列
```

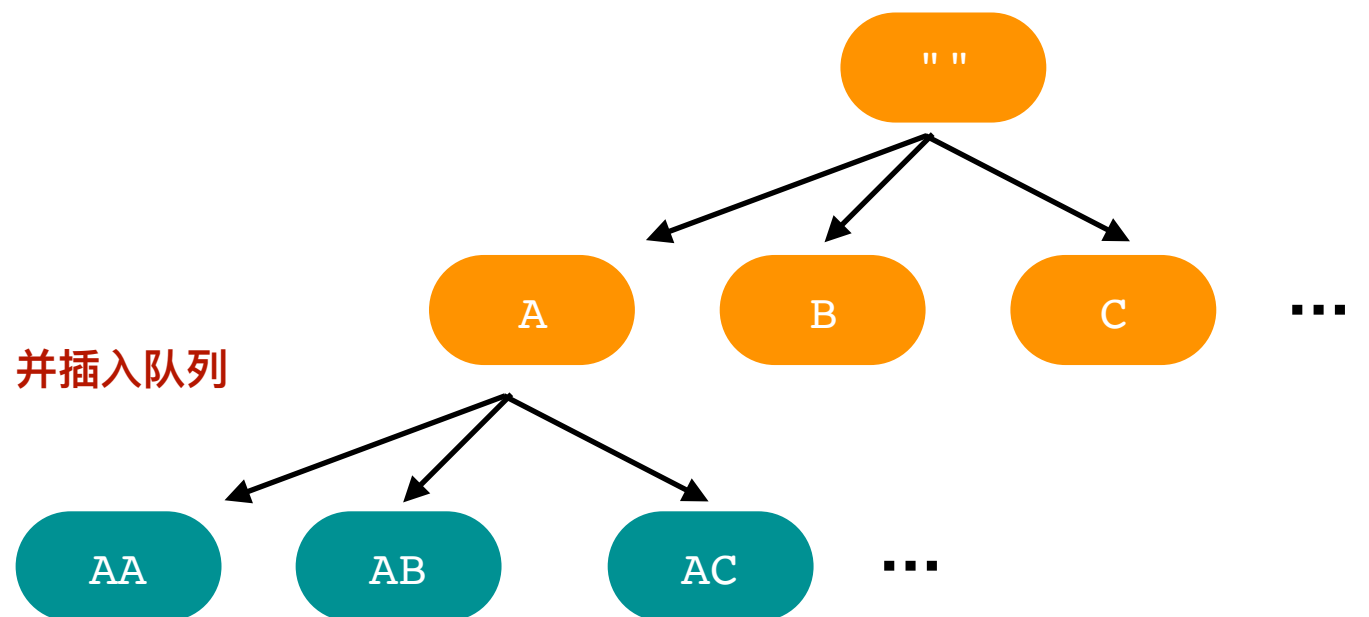
```
                q.push(str + c);
```

```
            }
```

```
        }
```

```
    }
```

```
}
```



- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

""	A	B	C	...	AA	AB	AC	...	BA	BB	BC	...	CA	CB	...
----	---	---	---	-----	----	----	----	-----	----	----	----	-----	----	----	-----

```
// 字符串生成算法
StringGenerate(candidates, length_limit) {
    // 初始化队列
    Queue q();
    // 将长度为0的空字符串插入队列
    q.push("");
    // 若队列不为空，则不断使用队首元素生成新的字符串，并插入队列
    while q.size() > 0 {
        // 删除并获取队列首字符串
        str = q.pop();
        print(str);
        if str.length() < length_limit {
            // 若字符串长度符合要求，则遍历所有候选字母
            for char in candidates {
                // 生成新的字符串，并插入队列
                q.push(str + c);
            }
        }
    }
}
```

广度优先搜索算法 (Breadth-First Search):

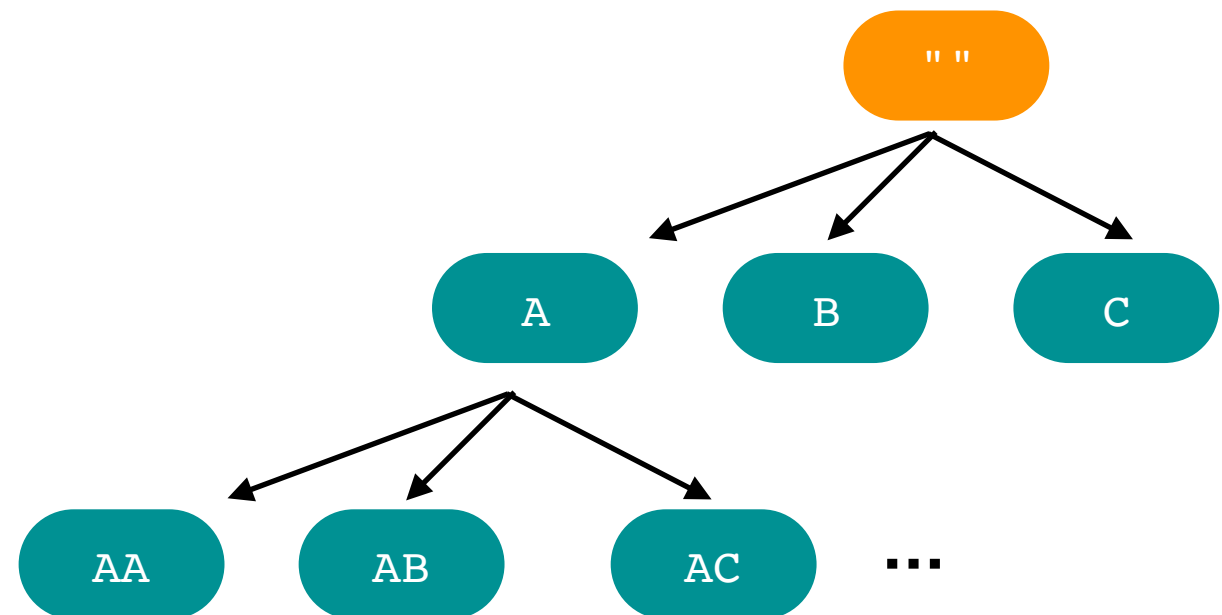
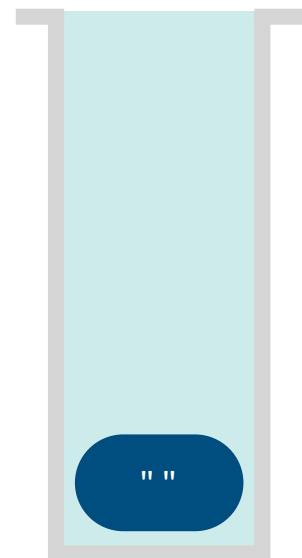
1. 使用队列，按插入队列的先后顺序依次遍历所有状态
2. 常用于寻找**最优解**，例如地图路径搜索

LeetCode 127. <Word Ladder>

<https://leetcode.com/problems/word-ladder/>

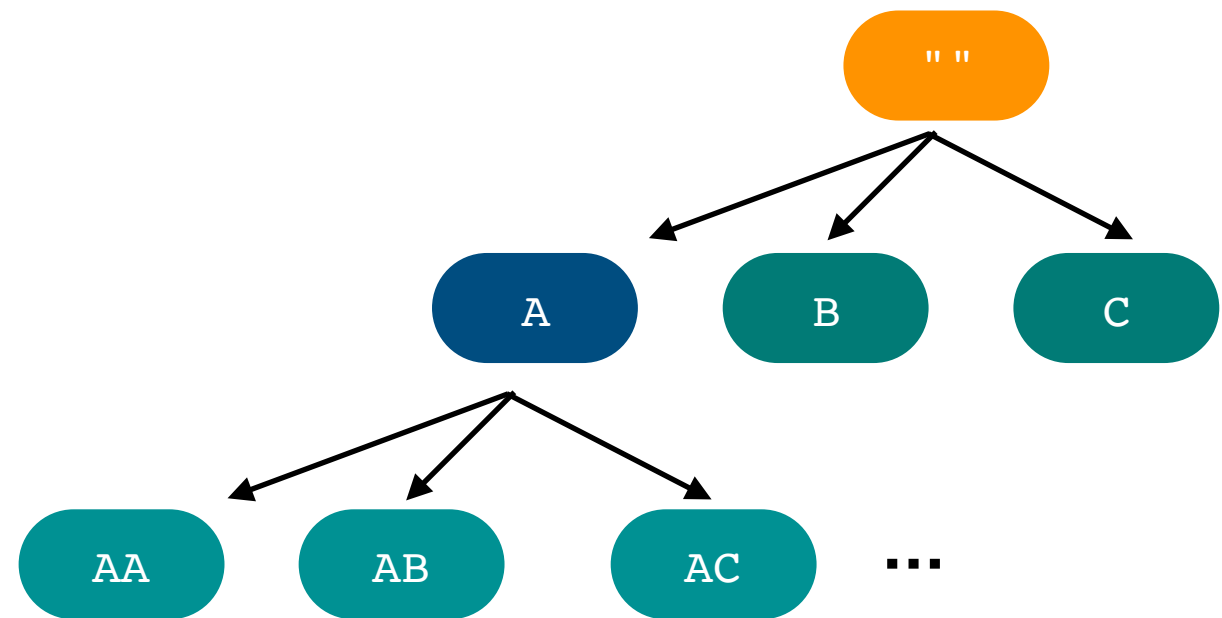
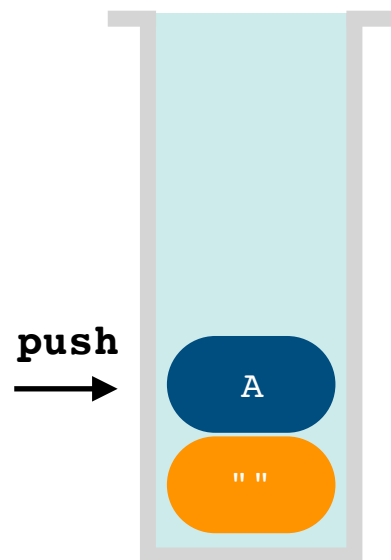
- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

" " A B C ... AA AB AC ... BA BB BC ... CA CB ...



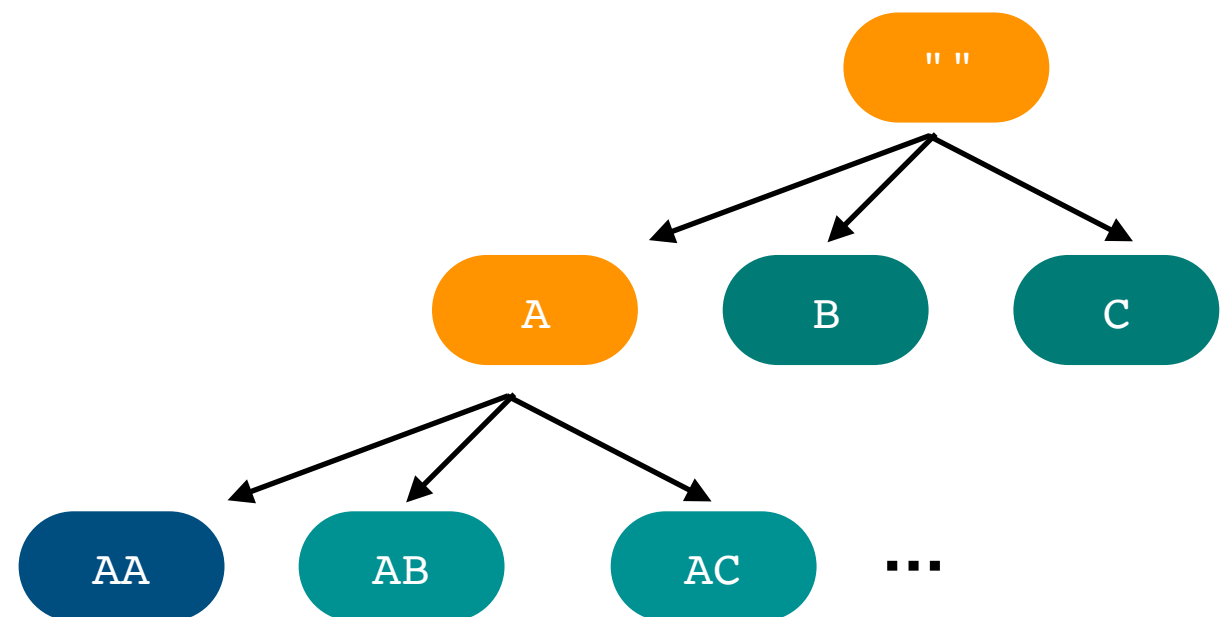
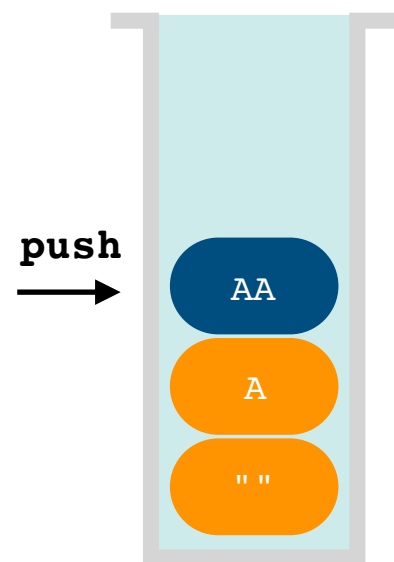
- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

"" A B C ... AA AB AC ... BA BB BC ... CA CB ...



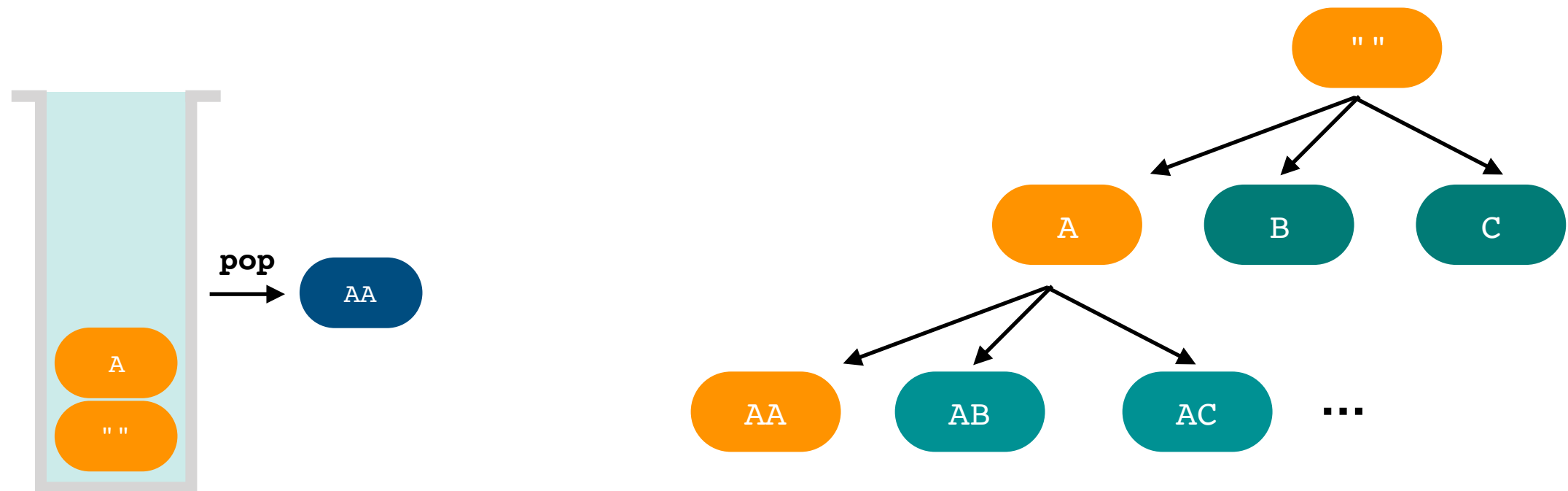
- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

"" A B C ... AA AB AC ... BA BB BC ... CA CB ...



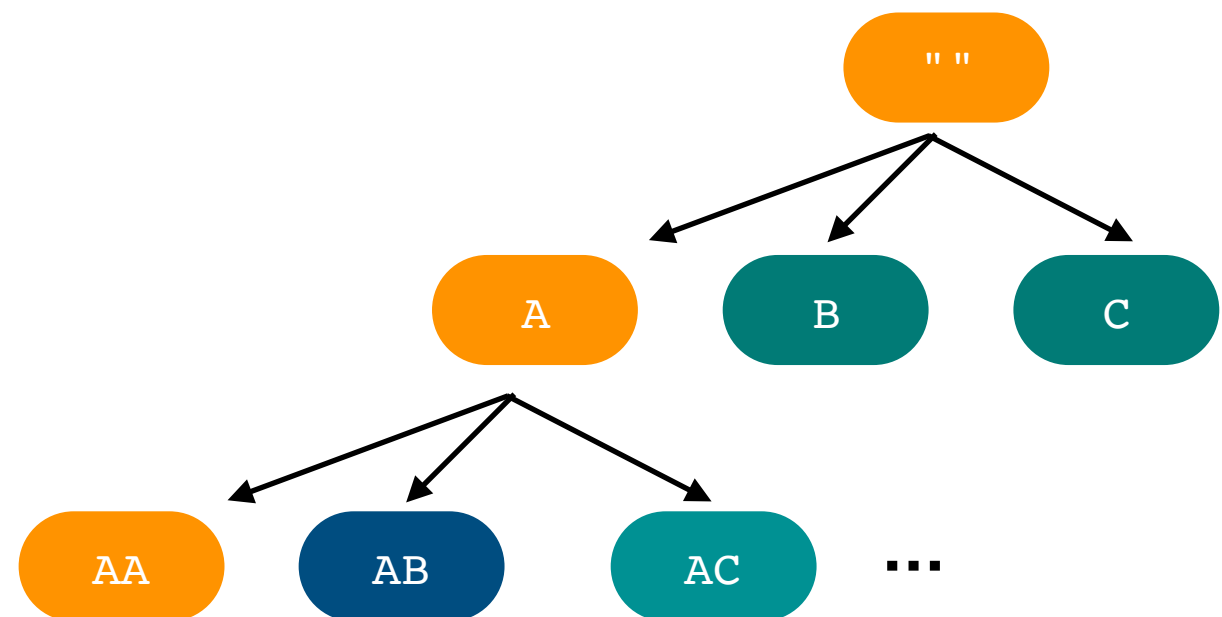
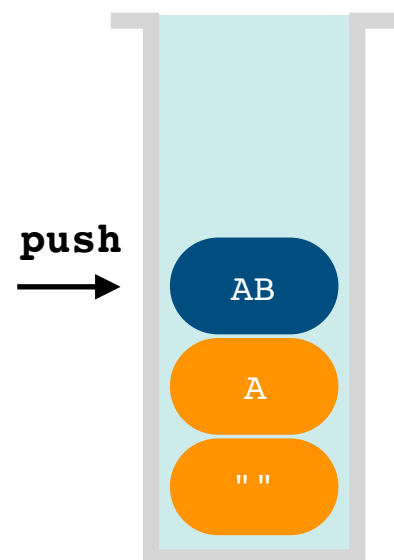
- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

"" A B C ... AA AB AC ... BA BB BC ... CA CB ...



- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

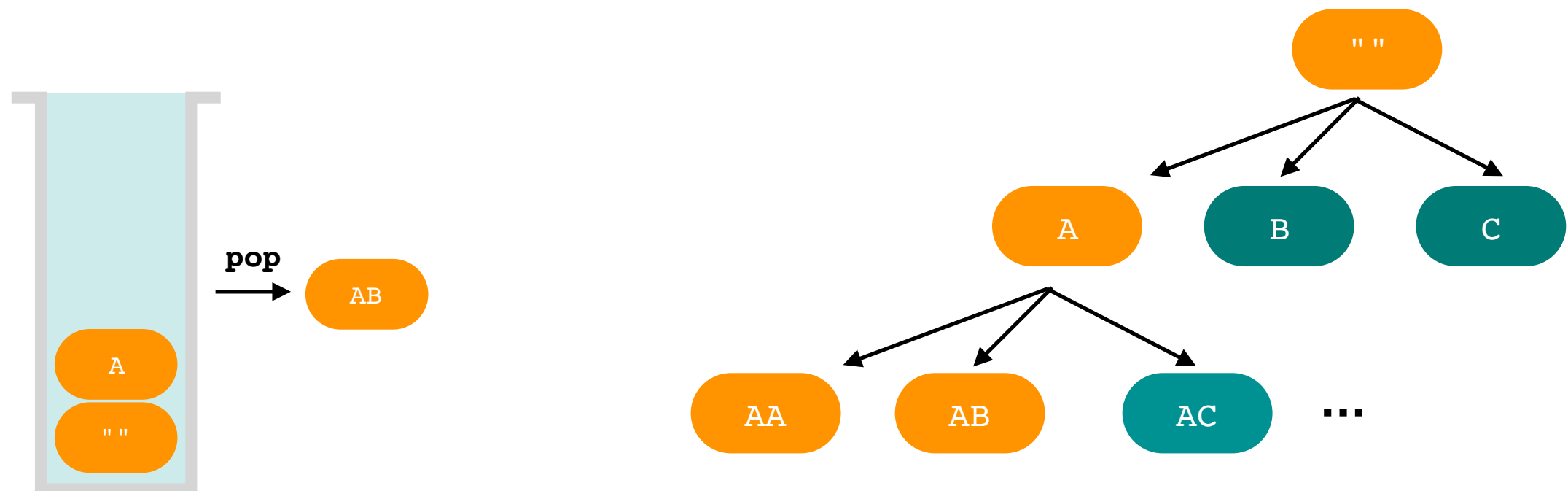
" " A B C ... AA AB AC ... BA BB BC ... CA CB ...



AA 已经达到指定长度限制，不再往栈中插入

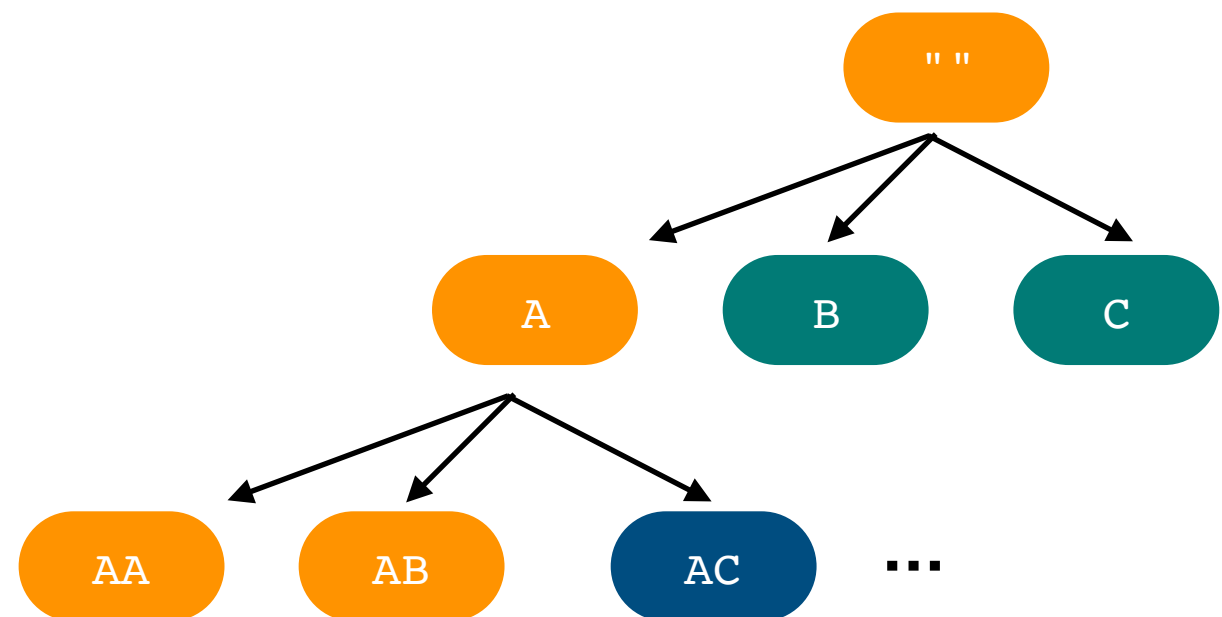
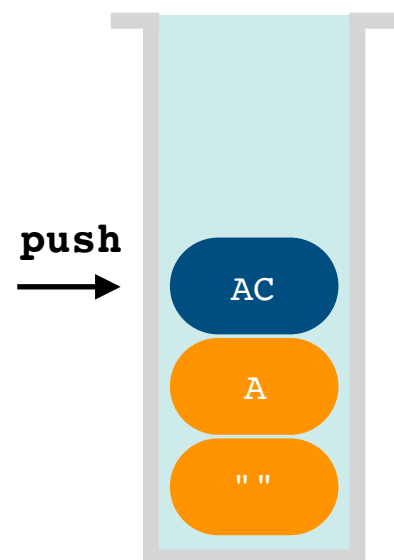
- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

"" A B C ... AA AB AC ... BA BB BC ... CA CB ...



- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

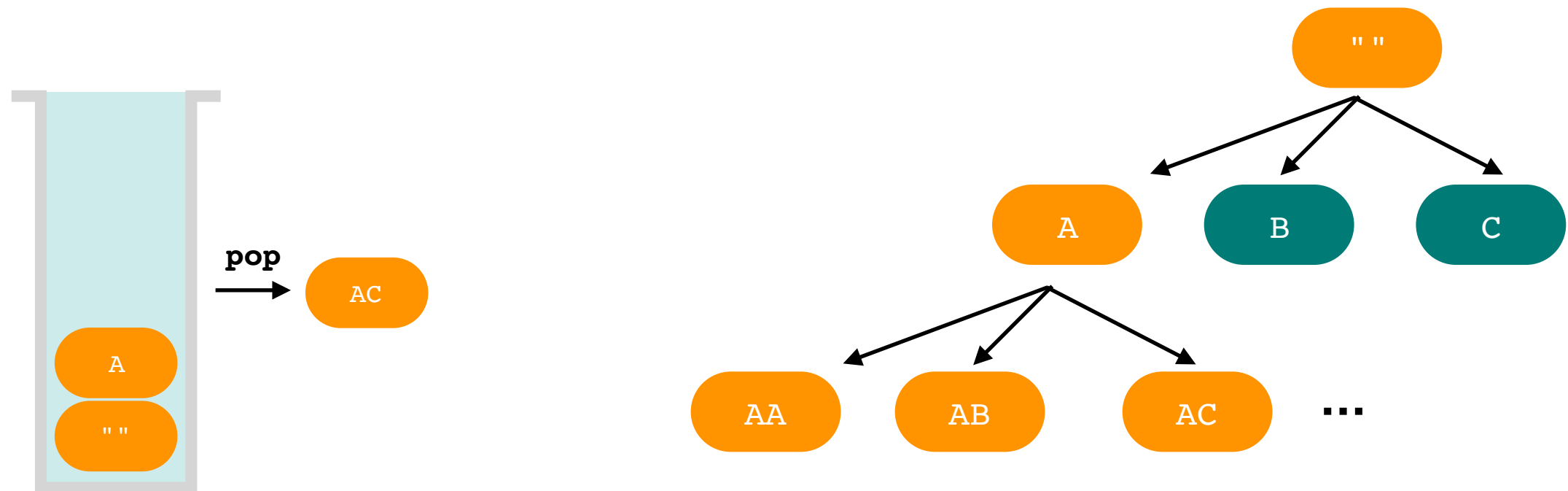
" " A B C ... AA AB AC ... BA BB BC ... CA CB ...



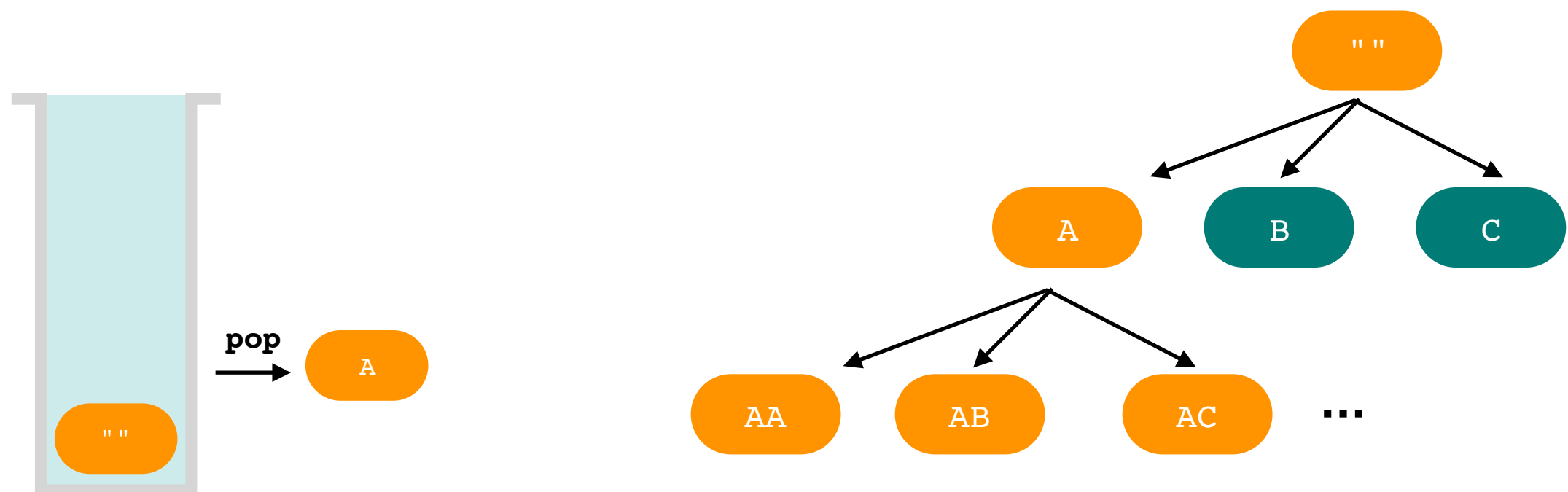
AB 已经达到指定长度限制，不再往栈中插入

- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

"" A B C ... AA AB AC ... BA BB BC ... CA CB ...

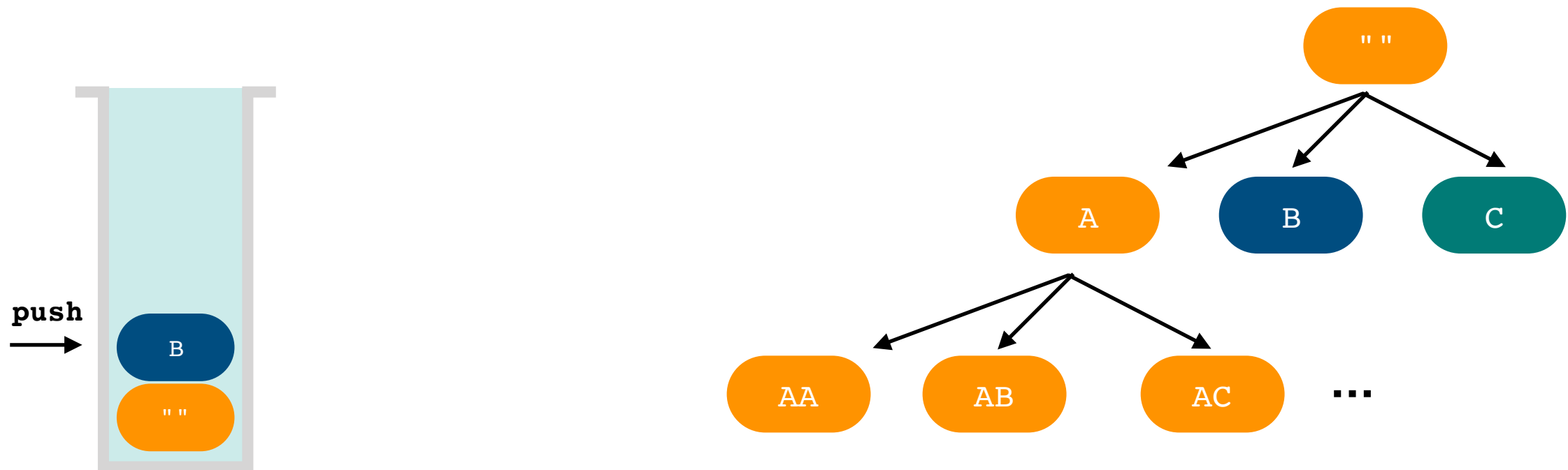


- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

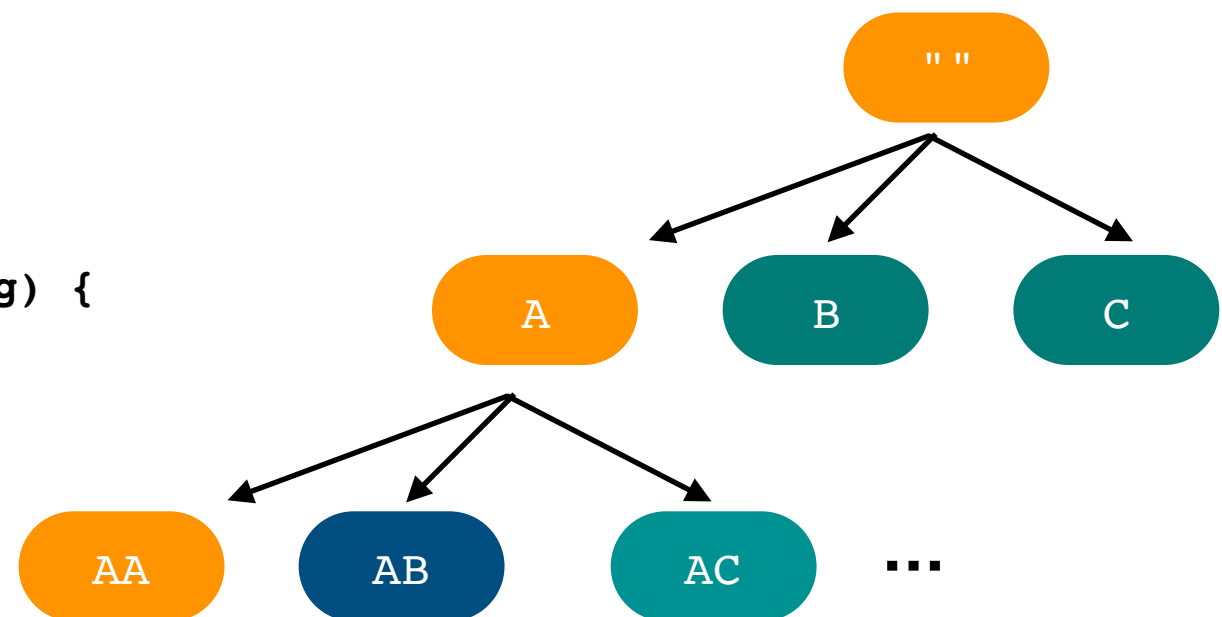


AC 已经达到指定长度限制，不再往栈中插入

- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串



- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串



```
// 字符串生成算法
StringGenerate(candidates, length_limit, string) {
    // 输出生成的字符串
    print(string);
    if string.length() >= length_limit {
        // 到达边界条件
        return;
    }
    for char in candidates {
        // 生成新的字符串，并插入队列
        StringGenerate(candidates, length_limit, string + char);
    }
}
```


- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串



```
// 字符串生成算法
StringGenerate(candidates, length_limit, string) {
    // 输出生成的字符串
    print(string);
    if string.length() >= length_limit {
        // 到达边界条件
        return;
    }
    for char in candidates {
        // 生成新的字符串，并插入队列
        StringGenerate(candidates, length_limit, string + char);
    }
}
```

深度优先搜索算法 (Depth-First Search):

1. 使用栈（或函数栈）；
2. 用途广泛，用于在各种问题的解决空间中进行遍历；
3. 内存显著低于广度优先搜索

- 栈与队列的应用 (3)
 - 字典生成问题：生成不超过指定长度的所有字母串

广度优先搜索算法 (Breadth-First Search) :

1. 使用队列，按插入队列的先后顺序依次遍历所有状态；
2. 常用于寻找**最优解**，例如地图路径搜索；
3. 时间~最优解出现时刻；
4. 空间~存储遍历的所有状态；

VS

深度优先搜索算法 (Depth-First Search) :

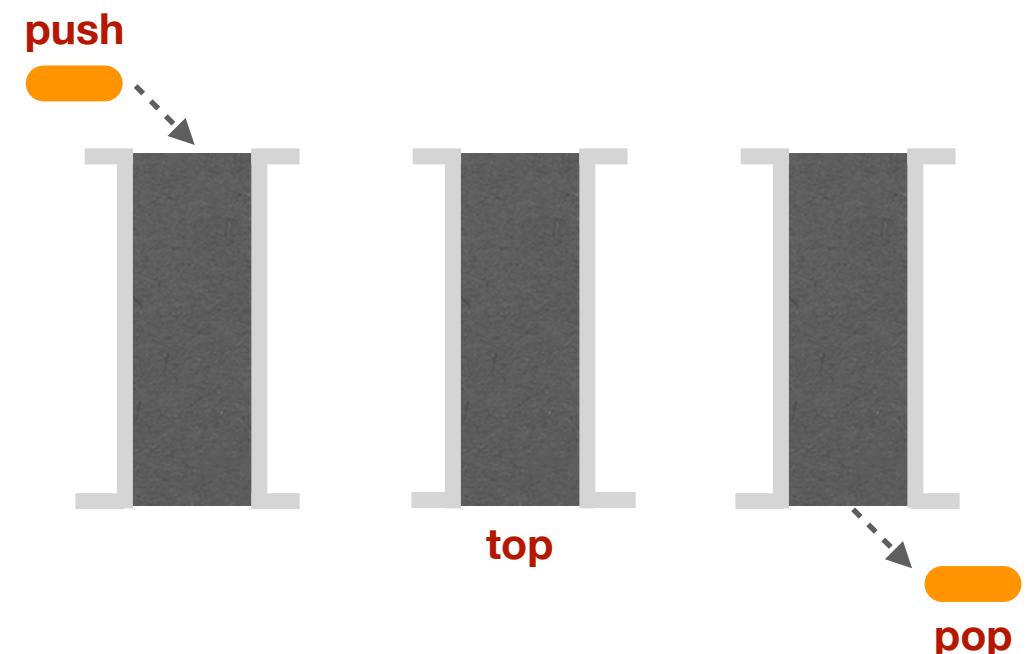
1. 使用栈（或函数栈）；
2. 用途广泛，用于在各种问题的解决空间中进行遍历；
3. 时间~遍历整个问题空间；
4. 空间~几乎无开销；

一定有一个平衡点：迭代加深深度优先搜索算法

目录

- 栈与队列
 - 栈与队列的定义与实现
 - 栈与队列的应用
 - 函数栈与括号匹配问题
 - 队列与轮训调度
 - 深度优先与广度优先遍历算法
 - 栈与队列的扩展
 - 优先队列
 - 栈 + get_max()
 - 使用栈模拟队列

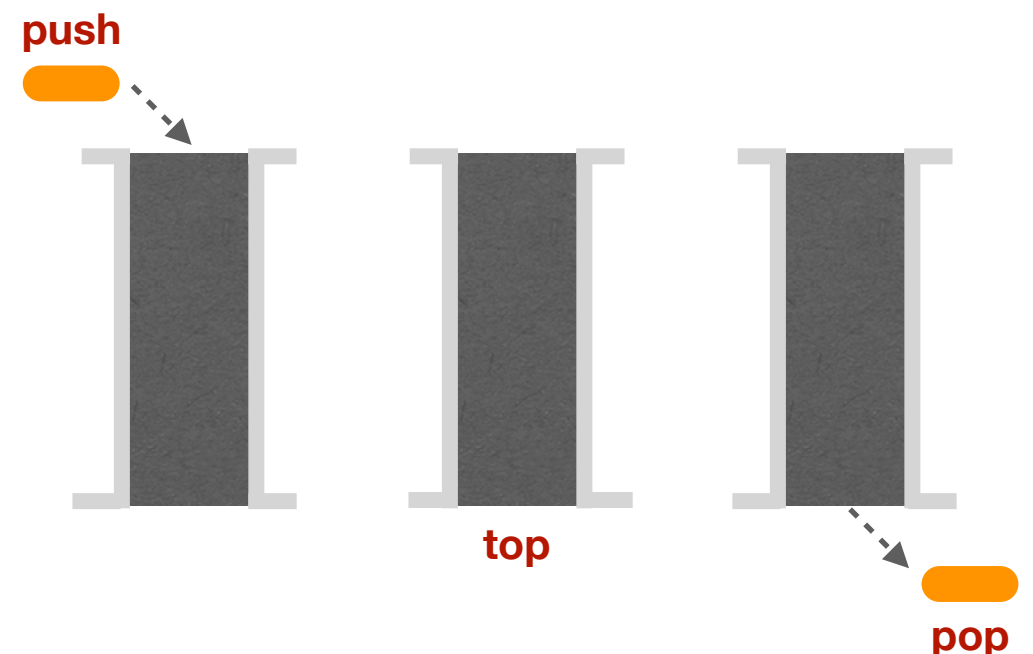
- 栈与队列的扩展 (3)
 - 优先队列 (Priority Queue) 是一种抽象数据类型 (ADT) 。
 - 可以向优先队列中插入元素 (push) ; 每个元素都有优先级, 而优先级高 (或者低) 的将会先出队 (pop) ; 同时也能获取优先队列中优先级最高 (或者最低) 的元素 (top)
 - 抽象的栈 (Stack) 和队列 (Queue) 都属于一种优先队列
 - 栈: 以元素插入次序作为优先级, 插入越早, 优先级越高
 - 队列: 以元素插入次序作为优先级, 插入越早, 优先级越低



- 栈与队列的扩展 (3)
 - 优先队列 (Priority Queue) 是一种抽象数据类型 (ADT) 。
 - 可以向优先队列中插入元素 (push) ; 每个元素都有优先级, 而优先级高 (或者低) 的将会先出队 (pop) ; 同时也能获取优先队列中优先级最高 (或者最低) 的元素 (top)
 - 抽象的栈 (Stack) 和队列 (Queue) 都属于一种优先队列
 - 栈: 以元素插入次序作为优先级, 插入越早, 优先级越高
 - 队列: 以元素插入次序作为优先级, 插入越早, 优先级越低

后续学习的各种数据结构和算法, 会研究如何高效实现各种优先队列, 找到队列中最高优先级的元素, 并将它从队列中删除。例如:

- 1) 排序问题: 最大堆、最小堆 (Heap)
- 2) 最短路径问题: Dijkstra 算法, 其实是广度优先搜索中, 将队列替换为以路径长度作为优先级的优先队列



- 栈与队列的扩展 (3)

- 优先队列应用举例

LeetCode 703. <Kth Largest Element in a Stream>

<https://leetcode.com/problems/kth-largest-element-in-a-stream/>

- 查找整数序列中第K大的元素

算法:

1) 定义优先队列, 以元素大小作为优先级, 值较小的元素优先级较高:

1) `push(k)` - 插入元素

2) `pop()` - 删除队列中最小的元素

3) `size()` - 返回队列中元素数目

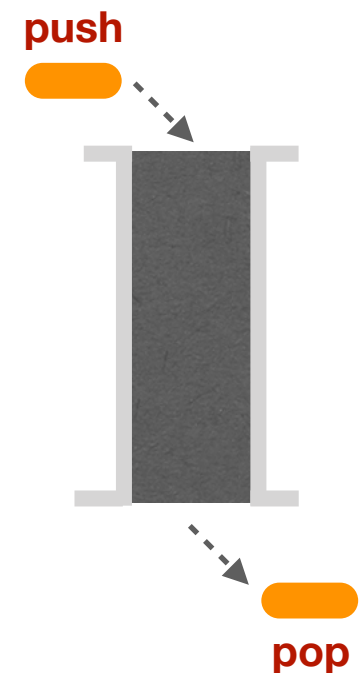
4) `top()` - 返回队列中最小的元素

2) 遍历整数序列

1) 将序列元素 `push()` 进优先队列

2) 若队列中的元素数目 `size() <= k`, 则重复 1); 否则, `pop()` 删除优先级最高的元素 (值最小的元素)

3) 返回队列中优先级最高的元素 (值最小的元素)



`pop()` 返回优先队列中最小的元素

- 栈与队列的扩展 (3)

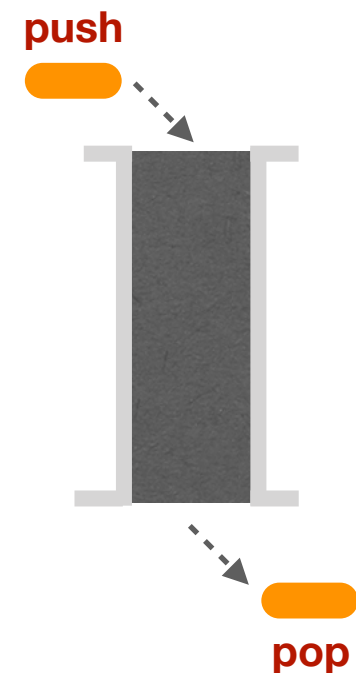
- 优先队列应用举例

LeetCode 703. <Kth Largest Element in a Stream>

<https://leetcode.com/problems/kth-largest-element-in-a-stream/>

- 查找整数序列中第K大的元素

```
int find_kth_max(const std::vector<int>& arr, int k) {  
    // 定义一个以元素大小作为优先级的优先队列，返回队列中最小的元素  
    MinHeap<int> h;  
    for (int element : arr) {  
        // 将新元素插入优先队列  
        h.push(element);  
        // 若优先队列中元素数目超过 k，则删除队列中优先级最高的元素（最小的元素）  
        if (h.size() > k) {  
            h.pop();  
        }  
    }  
    // 返回队列中优先级最高的元素（k个元素中最小的元素，即第k大的元素）  
    return h.top();  
}
```



pop() 返回优先队列中最小的元素

目录

- 栈与队列
 - 栈与队列的定义与实现
 - 栈与队列的应用
 - 函数栈与括号匹配问题
 - 队列与轮训调度
 - 深度优先与广度优先遍历算法
- 栈与队列的扩展
 - 优先队列
 - 栈 + get_max()
 - 使用栈模拟队列

- 栈与队列的扩展 (2)
- 实现一个栈，使其支持 `get_max()` 操作，获取栈内元素中的最大值（或最小值）

下一节课

- 栈与队列的扩展 (3)
- 使用栈模拟队列操作

下一节课

扩展练习

LeetCode 127. <Word Ladder>

<https://leetcode.com/problems/word-ladder/>

LeetCode 703. <Kth Largest Element in a Stream>

<https://leetcode.com/problems/kth-largest-element-in-a-stream/>