ESTR 2102 数据结构

> 基于 C 语言

①②③④⑤⑥⑦⑧⑨⑩

operands 操作数（被操作的数）

compatible 兼容

arithmetic 算数

rear （队列的） 尾

buffer area 缓冲区

adjacent 邻近的

vertex 顶点

deduce 推导

sparse 稀疏的

lexicographical 字典的

auxiliary 辅助的

manipulation 操作

deteriorate 恶化

1. C 语言 Review

++a：前置自增，先自增再使用.

a++：后置自增，先使用再自增.

`scanf()`

> Reading input from user

```c
#include <stdio.h>

int main(void){

    int num1, num2;

    printf("Enter two integers: \n");
    scanf("%d%d", &num1, &num2);

    printf("num1 = %d, num2 = %d\n", num1, num2);

    return 0;
}
```

user 可以选择两种输入方法：

```
123 456
```

或

```
123
456
```

中间的空格和换行打多少个都可以，只要把两个数字分开即可

Type Conversion

Rule: An operand with "simpler" data type will be converted to a more "expressive/complex" data type.

> 隐式类型转换，会保留两个操作数中较复杂（更多Byte）的类型

但是，5/2 相当于 2，因为两个都是 int，余数被抹除

Explicit Type Conversion (Casting)

> 显式类型转换

```
(new_type) operand
```

```
double d = 4.2;
int y = (int) d; // y becomes 4, no warning
int x = d;       // x becomes 4, compiler warns
                 // because of missing type casting
```

Operators

Basic arithmetic operators

> `+`, `-`, `*`, `/`, `%`

Relational Operators

> `<`, `<=`, `==`, `>=`, `>`, `!=`

Logical Operators

> `&&`, `||`, `!`

Assignment Operators

> `=`, `+=`, `-=`, `*=`, `/=`, `&=`

`&=`： 按位与赋值运算符。

示例

```
#include <stdio.h>

int main() {
    int a = 6;   // 二进制为 110
    int b = 3;   // 二进制为 011

    a &= b;      // 相当于 a = a & b

    printf("a 的值是 %d\n", a); // 输出结果为 2，二进制为 010

    return 0;
}
```

`&`：逐位比较两个数的二进制位，只有两个位都为1时结果才为1。

这种运算符常用于需要对变量的特定位进行清零或掩码操作的场景。

Increment and Decrement Operators

> `++`, `--`

The increment operator `++` can be placed in either prefix or postfix position, with different results.

`k = ++i`： prefix, 前缀，先加后赋值

`k = i++`： postfix, 后缀，先赋值后加

Bitwise Operators

> `&`, `|`, `~`, `^`

Bitwise AND (`&`)

Result is `1` if both corresponding bits are `1`, otherwise `0`.

```
int a = 5;  // Binary: 0101
int b = 3;  // Binary: 0011
int result = a & b;  // Result: 0001 (1 in decimal)
```

Bitwise OR (`|`)

Result is `1` if at least one corresponding bit is `1`.

```
int a = 5;  // Binary: 0101
int b = 3;  // Binary: 0011
int result = a | b;  // Result: 0111 (7 in decimal)
```

Bitwise NOT (`~`)

Result is `1` for each `0` and `0` for each `1`.

```
int a = 5;  // Binary: 0101
int result = ~a;  // Result: 1010 (in binary, -6 in decimal due to two's complement)
```

Bitwise XOR (`^`)

Result is `1` if the corresponding bits are different.

```
int a = 5;  // Binary: 0101
int b = 3;  // Binary: 0011
int result = a ^ b;  // Result: 0110 (6 in decimal)
```

These operators are useful for tasks like setting, clearing, or toggling specific bits.

Comma Operator, Parentheses, Conditional Operator, Member Operator, Pointer Operators, ...

If Statement

```
if (expr)
    statement;
next_statement;

if (expr)
    statement_1;
else
    statement_2;
next_statement;
```

> expr 为 true （返回 1）就执行

An else statement attaches to the nearest if that has not been paired with an else

Indentation and Compound Statement

在 C 中，Indenting code does not affect a program. It only makes it easier to read. 如果 if 后有多个语句要执行，需要 Compound Statement

```
if (x>0){
    printf("A");
    printf("B");
}
printf("C")
```

Common Mistakes

Using `=` instead of `==` as equality operator

```
if (a = 0) // 赋值语句返回等号右边的值，所以 a = 0 这个表达式总是返回 0 (False)
    some_statement;
```

Placing `;` after the condition of an if statement

```
if (a != 0);
    some_statement;
```

> 会直接结束这个 if

Conditionally performing 1 of N tasks

```
if (expr1)
    statement1;
else if (expr2)
    statement2;
else if (expr3)
    statement3;
```

is the same as

```
if (expr1)
    statement1;
else
    if (expr2)
        statement2;
    else
        if (expr3)
            statement3;
```

Looping

```
while (condition)
    statement1;
statement2;
```

```
for (init; condition; update)
    statement1;
statement2;
```

do-while loop

> 少用

Structures

```
struct struct_name {
    type1 member1;
    type2 member2a, member2b;
    ...
    typeN memberN;
};
```

> This only defines a new data type called `struct struct_name`

For example:

```
struct student {
    char id[11];
    char name[40];
    double gpa;
}; // 定义的时候不能赋值
```

To define a structure variable:

```
struct student Peter;
```

> 或者叫创建 student 的一个实例

也可以在创建的同时初始化成员的值

```
struct student Peter = {"10", "Peter", 4.0};
```

定义结构体的同时可以直接创建一个变量（实例）：

```
struct student {
    char id[11];
    char name[40];
    double gpa;
} Peter;
```

这相当于先定义结构体，再创建实例，因此即使定义的同时创建实例，该结构体也已经存在，之后可以创建其他实例：

```
#include <stdio.h>

int main() {
    struct student {
        char id[11];
        char name[40];
        double gpa;
    } Peter;

    struct student Steve = {"10", "Steve", 4.0};

    printf("id: %s\n", Steve.id);
    printf("name: %s\n", Steve.name);
    printf("gpa: %.1f\n", Steve.gpa);
    return 0;
}
```

注意：C 语言中，字符数组（char）可以在声明时使用初始化列表进行赋值

```
char name[40] = "Peter"; // 初始化赋值
```

一旦数组被初始化，就不能使用 `=` 为整个数组重新赋值。需要使用函数如 `strcpy` 来修改数组内容。

```
#include <stdio.h>

int main() {

    struct student {
        char id[11];
        char name[40];
        double gpa;
    } Peter;

    strcpy(Peter.name, "Peter");
    printf("name is %s\n", Peter.name);

    return 0;
}
```

Variable Scope

> 变量作用域

Variable are accessible only within the block in which they are declared.

Local Scope

> 局部作用域

```
void foo(int p){
    int q;
    ...
}
```

> p and q are only accessible inside foo().

```
int main(void){
    int x;
    ...
    if(...){
        int y;
        ...
    }
    return 0;
}
```

> x is only accessible inside main().
>
> y is only accessible within the if-block
>
> Accessing an identifier outside its scope will result in a compile-time error.

Global Scope (File Scope)

> Variables that are not declared inside of any function are commonly known as global variable. They are accessible anywhere in the same file.

```
int universe; // global

void foo(){
    printf("%d\n", universe);
    universe++;
}

int main(void){

    universe = 1;
    foo();
    printf("%d\n", universe);

    return 0;
}
```

Masking

> An identifier declared inside a block **masks** or **overshadows** the same identifier declared outside the block.

```
int bar = 0;

void foo(){
    bar = 1; // refer to the glocal "bar"
}

int main(void){
    int bar = 3;
    foo();
    printf("%d\n", bar);
    bar++; // refer to the "bar" declared in main()
    {
        int bar = 3;
        printf("%d\n", bar); // refer to the "bar" declared in the current block
    }
```

```
    bar--; // refer to the "bar" declared in main()
    printf("%d\n", bar); // refer to the "bar" declared in main()
    return 0;
}
```

输出:

```
3
3
3
```

Global variable is a powerful tool available in C.

However, we should **NOT** use it in general.

When there is something wrong with the value of a local variable, we can easily look for the bug in its scope.

The value of a *global variable* is hard to tell and predict because it can be modified anywhere in any order!

Instead, we should use parameters and return values to exchange information between functions.

> 把实参传给函数，函数用形参在内部处理，就避免了全局变量的使用。

Passing Arrays to Functions

```
void printArray(int array[], int arraySize) {
    int i;
    for (i = 0; i < arraySize; i++)
        printf("%d ", array[i]);
    printf("\n");
}

int main(void) {
    int A[5] = { 1, 2, 3, 4, 5 }, B[10] = { 0 };

    printArray(A, 5);   // Output "1 2 3 4 5 "
    printArray(B, 10);  // Output "0 0 0 0 0 0 0 0 0 0 "
    printArray(A, 3);   // Output "1 2 3 "

    return 0;
}
```

Pointer

```
// Version 1
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

// Version 2
void swap(int *a, int *b) {
    int *tmp = a; // 这里甚至可以直接用整数变量 tmp，但不建议这么使用。可以使用的原因只是指针和整数变量恰好占用相同的字节数
    a = b;
    b = tmp;
}

int main(void) {
    int x = 5, y = 2;
    swap(&x, &y);
    return 0;
```

```
}
```

> *号有两种用法。
>
> 在函数声明里，swap(int *a, int *b) 中 `*` 用于声明 `a` 和 `b` 为指向 int 的指针（a 和 b 存地址）. 在本例中，`a` 指向 `x`，`b` 指向 `y`. 换句话说，它只是声明参数是指针类型。
>
> 第二种用法，在函数体内，`*a` 和 `*b` 是解引用操作（a 和 b 存地址，解引用后变成地址中存的值），用于访问和修改指针所指向的值。注意，如果对某个地址中的值进行修改，意味着该地址中的值发生了改变，后续查询该地址就会显示修改后的值。
>
> Version 2，在 swap 函数执行后，`a` 指向 `y`, `b` 指向 `x`, 但是 `x` 和 `y` 的值没有交换。Version 1 可以正常实现交换功能。

函数指针

```
qsort(arr, n, sizeof(int), compare);
```

此处 `compare` 是一个函数

完整代码

```c
#include <stdio.h>
#include <stdlib.h>

int compare(const void * a, const void * b) {
    return ( *(int*)a - *(int*)b );
}

int main() {
    int arr[] = {10, 5, 15, 12, 90, 80};
    int n = sizeof(arr)/sizeof(arr[0]), i;
    qsort (arr, n, sizeof(int), compare);
    for (i=0; i<n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Type Conversation

Explicit Type Conversion (Casting)

> 强制类型转换

Syntax

```
(new_type) operand
```

> Note that not every type conversion is possible.

Increment Operator

```
a = 1;
b = a++;
```

> a = 2, b = 1. 先赋值再更新

```
a = 1;
b = ++a;
```

> a = 2, b = 2. 先更新再赋值

`{}`: compound statement

Common Mistakes

`==` 不要写成 `=`

`if` 后面不要加 `;`

Structure

```
struct student {
    char id[11];
    char name[40];
    double gpa;
};
```

> 写在 `main` 外面
>
> 定义了一种新的数据类型 `struct student`

To define a structure variable:

```
struct student peter;
```

> 写在 `main` 里面

2. 渐进分析

Data Elements（数据元素） 和 Data Items（数据项）

> Data Element：数据基本单位，可包含多个数据项。通常由多个特征或属性组成。
>
> Data Item：数据最小单元，不能进一步分割。是数据元素的组成部分。

例：

一个「学生」数据元素可包含多个属性，如姓名、性别、出生日期、入学年份和 GPA。

`{姓名: "张三", 性别: "男", 出生日期: "2000-01-01", 入学年份: 2018, GPA: 3.5}`

数据项：在上述「学生」数据元素中，每个属性就是一个数据项。

> 姓名（"张三"）
>
> 性别（"男"）
>
> 出生日期（"2000-01-01"）

> 入学年份（2018）
>
> GPA（3.5）

Logical Structure

> Linear structures (Linear List, Stack, Queue) and Non-linear structures (Tree, Graph)

Physical Structure

> Sequential storage structure and Linked storage structure (using pointer)

Sequential storage structure：

优点：可以直接定位（公式计算）

缺点：需要连续的内存，不便于拓展

Linked storage structure：

优点：便于拓展（延长链）

缺点：搜索某个元素，需要从头开始定位

Programming and Algorithm

Complexity

> Space Complexity：storage spaces
>
> Time Complexity：number of steps
>
> 重点考虑时间复杂度

Asymptotic Analysis

> Worst case：O
>
> Average case：Θ
>
> Best case：Ω
>
> 重点考虑 Worst case

Big-O Analysis

An algorithm takes $\frac{1}{2}n^2 - \frac{1}{2}n + 10$ steps to run. We say that it is $O(n^2)$.

> 考虑 leading term，主导项

根据数学定义，$O(g(n))$ 中的 $g(n)$ 就是去掉系数的主导项

> 由于定义中 c 的存在，不用考虑系数

small o 的限制更强

2.1 符号

> 摘自 ENGG 2440 离散数学. 数据结构只考 Big-O.

2.1.1 Big-O

表示函数增长的上界.

$$f(x) = O(g(x)) \quad if \quad \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad f(x) \leq cg(x)$$

> 虽然一个函数可以有多个 Big-O，但通常选择**最紧的上界**，以提供更精确的增长描述.
>
> 注意：c 大于 0.

2.1.2 Big-Ω

表示函数增长的下界.

$$f(x) = \Omega(g(x)) \quad if \quad \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad f(x) \geq cg(x)$$

Theorem 1

$f(x) = O(g(x))$ is the same as $g(x) = \Omega(f(x))$.

$$\begin{aligned} f(x) = O(g(x)) &\Rightarrow \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad f(x) \leq cg(x) \\ &\Leftrightarrow \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad g(x) \geq \frac{1}{c} f(x) \end{aligned}$$

By taking $c' = \frac{1}{c} > 0,\ x_0' = x_0 \geq 0$, we have $g(x) = \Omega(f(x))$.

2.1.3 Θ

同时满足 O 和 Ω 的条件，表示函数增长的确切阶.

$$f(x) = \Theta(g(x)) \quad if \quad f(x) = O(g(x)) \quad and \quad g(x) = O(f(x)).$$

> $f(x) = \Theta(g(x)) \nRightarrow f(x) = g(x)$. It just means that
> $\exists\, c_1, c_2 > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad c_1 g(x) \leq f(x) \leq c_2 g(x)$
>
> 只能说明 $f(x)$ 和 $g(x)$ 同阶.

2.1.4 Small-o

表示严格上界（严格更高阶），比 Big-O 更严格的界定.

$$f(x) = o(g(x)) \quad if \quad \lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$$

等价于

$$f(x) = o(g(x)) \quad if \quad \forall\, c > 0,\ \exists\, x_0 > 0\ such\ that\ 0 \leq f(x) < cg(x)\ for\ all\ x \geq x_0.$$

> 注意同阶不足以满足任意 c，$g(x)$ 要比 $f(x)$ 高阶才满足.

$$\begin{aligned} &f(x) = o(g(x)) \Rightarrow f(x) = O(g(x)) \\ &f(x) = O(g(x)) \nRightarrow f(x) = o(g(x)) \end{aligned}$$

2.1.5 Small-ω

表示严格下界（严格更低阶），比 Big-Ω 更严格的界定.

$$f(x) = \omega(g(x)) \quad if \quad \lim_{x \to \infty} \frac{g(x)}{f(x)} = 0.$$

等价于

$$f(x) = \omega(g(x)) \quad if \quad \forall\, c > 0,\ \exists\, x_0 > 0\ such\ that\ 0 \leq cg(x) < f(x)\ for\ all\ x \geq x_0.$$

> 注意同阶不足以满足任意 c，$g(x)$ 要比 $f(x)$ 低阶才满足.

$$\begin{aligned} &f(x) = \omega(g(x)) \Rightarrow f(x) = \Omega(g(x)) \\ &f(x) = \Omega(g(x)) \nRightarrow f(x) = \omega(g(x)) \end{aligned}$$

2.2 性质

> Properties for Asymptotic Analysis
>
> 数据结构不考.

- 传递性（Transitivity）

> If $f(n) = \Pi(g(n))$ and $g(n) = \Pi(h(n))$, then $f(n) = \Pi(h(n))$, where $\Pi = O,\ o,\ \Omega,\ \omega,\ or\ \Theta$.

- 加法法则（Rule of sums）

> $f(n) + g(n) = \Pi(max\{f(n), g(n)\})$, where $\Pi = O, \Omega$ or Θ.

- 乘法法则（Rule of products）

> If $f_1(n) = \Pi(g_1(n))$ and $f_2(n) = \Pi(g_2(n))$, then $f_1(n)f_2(n) = \Pi(g_1(n)\ g_2(n))$, where $\Pi = O,\ o,\ \Omega,\ \omega$, or Θ.

- 转置对称性（Transpose symmetry）

> $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
>
> $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

- 自反性（Reflexivity）

> $f(n) = \Pi(f(n))$, where $\Pi = O,\ \Omega$, or Θ.

- 对称性（Symmetry）

> $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

2.3 例子

We will care about the following functions that appear often in data structures:

常数	对数	linear	n log n	二次	polynomial	指数
$O(1)$	$O(log\ n)$	$O(n)$	$O(n\ log\ n)$	$O(n^2)$	$O(n^k)$	$O(a^n)$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$$
$$\sum_{i=1}^{n} 4^i = 4 \cdot \frac{1-4^n}{1-4} = O(4^n)$$
$$\frac{log(n^2+1)}{log(n)} \approx \frac{2log(n)}{log(n)} = 2 = O(1)$$
$$\sum_{i=1}^{n} \frac{1}{i} = H_n \approx ln(n) + \gamma = O(log\ n)$$

Example: Running Time Analysis

> Count the total number of operations and deduce the big-O.

```
int sum(int n){                 // no operations
    int i, psum = 0;            // 1 assignment
    for(i=0; i<=n; i++)         // 1 assignment, n+2 checking, n+1 increment, n+1 assignment
        psum += i * i * i;      // 2 multi, 1 plus, 1 assignment
    return psum;                // 1 return
}
```

Total:

$$\begin{aligned} & 1 + (1 + (n+2) + (n+1) + (n+1)) + (2+1+1)(n+1) + 1 \\ = \ & 7n + 11 \\ = \ & O(n) \end{aligned}$$

Example: Constant Time $O(1)$

```
int multiple(int n){
    int product = n*n*n;
    return product;
}
```

Example: Linear Time $O(n)$

```
double mean(int n, double *values){
    double sum = 0;
    int i;
    for(i=0; i<n; i++)
        sum += values[i];
    return sum / n;
}
```

Example: Quadratic Time $O(n^2)$

```
struct matrix{
    int n, m;
    double **mat;
};

void scale(struct matrix *M, double factor){
    int i, j;
    for(i=0; i<M->n; i++)
        for(j=0; j<M->m; j++)
            M->mat[i][j] = M->mat[i][j] * factor;
}
```

Example: logarithm Time $O(log\ n)$

> 二分搜索

```
int binarySearch(int arr[], int size, int x){
    int first, last;
    first = 0; last = size - 1;
    while (first <= last) {
        int middle = (first + last) / 2;
        if (arr[middle] < x) first = middle + 1;
        else if (x < arr[middle]) last = middle - 1;
        else return middle;
    }
    return -1; // Not found
}
```

> 每次迭代，范围缩小为原来的 $\frac{1}{2}$，迭代次数最多为：$\lfloor log_2(size) \rfloor + 1$
>
> 每次迭代，执行的操作均为 $O(1)$
>
> 总时间复杂度为 $O(log(size))$
>
> 固定数量的变量，没有使用递归，总空间复杂度为 $O(1)$

Example: Exponential Time $O(2^n)$

```
int fibonacci(unsigned int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

递归过程可表示为二叉树，例如，计算 `fibonacci(5)`：

```
fibonacci(5)
├── fibonacci(4)
|   ├── fibonacci(3)
|   |   ├── fibonacci(2)
|   |   |   ├── fibonacci(1)  (returns 1)
|   |   |   └── fibonacci(0)  (returns 0)
|   |   └── fibonacci(1)      (returns 1)
|   └── fibonacci(2)
|       ├── fibonacci(1)      (returns 1)
|       └── fibonacci(0)      (returns 0)
└── fibonacci(3)
    ├── fibonacci(2)
    |   ├── fibonacci(1)      (returns 1)
    |   └── fibonacci(0)      (returns 0)
    └── fibonacci(1)          (returns 1)
```

时间复杂度：每个节点表示一次函数调用，且每次调用的工作量为 $O(1)$. 因此，总时间复杂度等于递归树的节点总数 $O(2^n)$

空间复杂度：每次递归调用会占用栈空间，直到递归返回. 栈的最大深度为 n，因此空间复杂度为 $O(n)$.

Example: Better Fibonacci

```
long fibonacciIter(int n){

    long result = 1;
    long prev = 1;

    if(n == 0)
        return 0;
    if(n == 1)
        return 1;

    for(int i=2; i<n; i++){
        long current = result;
        result += prev;
        prev = current;
    }

    return result;
}
```

时间复杂度：每次迭代只进行 $O(1)$ 次操作. 总时间复杂度为 $O(n)$.

空间复杂度：固定数量的变量，没有使用递归，空间复杂度为 $O(1)$.

Example: Better Recursive Fibonacci

> 尾递归

```
long fibHelper(int n, long p0, long p1){
    if(n==1) return p1;
    return fibHelper(n-1, p1, p0+p1);
}

long fibnacci(int n){
    if (n==0) return 0;
    return fibHelper(n, 0, 1);
}
```

时间复杂度：$O(n)$

空间复杂度：若编译器应用尾调用优化，为 $O(1)$；若未应用尾调用优化，为 $O(n)$.

补充：尾调用优化

> Tail Call Optimization, TCO. 不考

尾调用：一个函数在最后一步调用另一个函数（或它自身），并直接返回结果.

> 调用后没有额外操作需要完成，因此不需要保持当前的调用栈帧.
>
> 最后一步调用自身即尾递归.

传统递归中，每次递归调用都会在调用栈上分配一个新的栈帧，用于保存函数状态，直到函数执行完成或遇到 `return` 语句后弹出栈. 如果递归调用深度很大，可能触发栈溢出错误（Stack Overflow）.

现代编译器（如 GCC）支持尾调用优化：

① 释放当前栈帧：如果当前函数是尾调用，编译器会识别这种模式，并在调用尾函数时直接复用（覆盖）当前栈帧，而不是分配新栈帧.

② 通过复用栈帧，尾调用优化将递归等效为迭代，显著降低内存占用.

Example: Maximum Subsequence Sum Problem

> 最大子序列和问题

给定一个整数数组（可以包含正数、负数或零），寻找一个连续子序列，使其元素之和最大，并返回这个最大和.

① 暴力枚举法

```
int maxSubsequenceSumBruteForce(int arr[], int n) {
    int maxSum = 0; // 允许空序列
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++) {
            int currentSum = 0;
            for (int k = i; k <= j; k++)
                currentSum += arr[k];
            if (currentSum > maxSum)
                maxSum = currentSum;
        }
    return maxSum;
}
```

时间复杂度：$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)=O(n^3)$

空间复杂度：$O(1)$

② 优化暴力法

> 在内层循环中，维护当前子序列的累加和，避免重复计算.

```
int maxSubsequenceSumOptimized(int arr[], int n) {
    int maxSum = 0;  // 允许空序列
    for (int i = 0; i < n; i++) {
        int currentSum = 0;
        for (int j = i; j < n; j++) {
            currentSum += arr[j];
            if (currentSum > maxSum)
                maxSum = currentSum;
        }
    }
    return maxSum;
}
```

时间复杂度：$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1} O(1) = O(n^2)$

空间复杂度：$O(1)$

③ 动态规划法

> Kadane's Algorithm，高效简单，最常用.

```
int maxSubsequenceSumKadane(int arr[], int n) {
    int maxSum = 0;  // 允许空序列
    int currentSum = 0;

    for (int i = 0; i < n; i++) {
        currentSum = currentSum + arr[i];
        if (currentSum < 0)
            currentSum = 0;
        if (currentSum > maxSum)
            maxSum = currentSum;
    }

    return maxSum;
}
```

时间复杂度：$O(n)$

空间复杂度：$O(1)$

④ 分治法

```
int max(int a, int b, int c) {
    int maxVal = a > b ? a : b;
    return maxVal > c ? maxVal : c;
}

int maxCrossingSum(int arr[], int left, int mid, int right) {
    int leftSum = 0, rightSum = 0;

    int tempSum = 0;
    for (int i = mid; i >= left; i--) {
        tempSum += arr[i];
        if (tempSum > leftSum)
            leftSum = tempSum;
    }

    tempSum = 0;
    for (int i = mid + 1; i <= right; i++) {
        tempSum += arr[i];
        if (tempSum > rightSum)
            rightSum = tempSum;
    }
```

```
    return leftSum + rightSum;
}

int maxSubsequenceSumDivideAndConquer(int arr[], int left, int right) {
    if (left == right)
        return arr[left] > 0 ? arr[left] : 0;

    int mid = (left + right) / 2;

    int leftMax = maxSubsequenceSumDivideAndConquer(arr, left, mid);
    int rightMax = maxSubsequenceSumDivideAndConquer(arr, mid + 1, right);
    int crossMax = maxCrossingSum(arr, left, mid, right);

    return max(leftMax, rightMax, crossMax);
}
```

时间复杂度：$T(n) = 2T(\frac{n}{2}) + O(n)$. 由主定理得，

$O(n\ log\ n)$

空间复杂度：$O(log\ n)$. 其中 $log\ n$ 为递归深度.

2.4 主定理

> Master Theorem，考试不考

主定理是分析**分治法**时间复杂度的强大工具，适用于以下递归关系：

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

通过比较 $p = \log_b a$ 和 d 的大小，可以快速得出时间复杂度的渐近结果：

- $d < p \Rightarrow O(n^p)$
- $d = p \Rightarrow O(n^p \log n)$
- $d > p \Rightarrow O(n^d)$

以下是详细介绍.

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

含义：

- $T(n)$: 表示问题规模为 n 时的运行时间.
- a: 子问题的数量.（每次划分生成的子问题数）
- n/b: 每个子问题的规模.（问题被划分为 b 等份）
- $O(n^d)$: 分解和合并子问题的代价.（除递归调用外，剩余的"额外工作"代价）

主定理通过比较 n^d 和 $n^{\log_b a}$（$a^{\log_b n}$）的增长速度来确定时间复杂度. 规则如下：

定义 $p = \log_b a$：表示递归树中子问题的增长速率.

比较 p 和 d：

① $d < p$

分解和合并的代价 $O(n^d)$ 比递归子问题的代价小，时间复杂度由子问题主导.

$$T(n) = O\left(n^p\right)$$

② $d = p$

两者共同决定时间复杂度，会乘上一个对数因子（源自递归树高度 $\log_b n$）.

$$T(n) = O\left(n^p \log n\right)$$

③ $d > p$

分解和合并的代价 $O(n^d)$ 比递归子问题的代价大，时间复杂度由分解和合并主导.

$$T(n) = O\left(n^d\right)$$

主定理的核心思想是**递归树**的分析：

- 每个递归调用会分解为 a 个子问题，每个子问题的大小为原问题的 $1/b$.
- 递归树的高度为 $\log_b n$（因为问题规模不断缩小，直到规模为 1）.

应用示例

① 二分查找

递归关系：

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

- $a = 1$（只生成一个子问题）
- $b = 2$（子问题规模减半）
- $d = 0$（分解合并的额外代价是常数 $O(1)$）

计算 $p = \log_b a = \log_2 1 = 0$.

比较 $d = 0$ 和 $p = 0$，满足 $d = p$.

因此，时间复杂度为：

$$T(n) = O(n^p \log n) = O(\log n)$$

② 合并排序

> Merge Sort

递归关系：

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

- $a = 2$（生成两个子问题）
- $b = 2$（子问题规模减半）
- $d = 1$（分解和合并的代价是 $O(n)$）

计算 $p = \log_b a = \log_2 2 = 1$.

比较 $d = 1$ 和 $p = 1$，满足 $d = p$.

因此，时间复杂度为：

$$T(n) = O(n^p \log n) = O(n \log n)$$

③ Strassen 矩阵乘法

递归关系：

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

- $a = 7$（生成 7 个子问题）
- $b = 2$（子问题规模减半）
- $d = 2$（分解和合并的代价是 $O(n^2)$）

计算 $p = \log_b a = \log_2 7 \approx 2.81$.

比较 $d = 2$ 和 $p \approx 2.81$，满足 $d < p$.

因此，时间复杂度为：

$$T(n) = O(n^p) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

局限

主定理并不适用所有递归关系，以下是一些限制：

非均匀划分：主定理假设子问题规模均匀，每次划分为 b 等份. 如果划分不均匀（如 $T(n) = T(n-1) + O(1)$），则主定理不适用.

额外代价非多项式：主定理假设额外工作是 $O(n^d)$ 的多项式阶. 如果额外工作是对数阶或更高阶（如 $O(n \log n)$），则主定理不适用.

递归子问题数量变化：如果子问题数量 a 不固定，也无法直接应用主定理.

2.5 常用近似表

function	name	typical value	approximation
$\lfloor x \rfloor$	floor	$\lfloor 3.14 \rfloor = 3$	x
$\lceil x \rceil$	ceiling	$\lceil 3.14 \rceil = 4$	x
$\lg N$	binary log	$\lg 1024 = 10$	$1.44 \ln N$
F_N	Fibonacci	$F_{10} = 55$	$\frac{\Phi^N}{\sqrt{5}}$
H_N	harmonic	$H_{10} \approx 2.9$	$\ln N + \gamma$
$N!$	factorial	$10! = 3628800$	$\sqrt{2\pi N}(\frac{N}{e})^N$
$\lg(N!)$		$\lg(100!) \approx 525$	$1.33 + (N + \frac{1}{2}) \lg N - 1.44N$

> binary log 就是以二为底的对数.
>
> $\Phi = \frac{\sqrt{5}+1}{2} \approx 1.618$
>
> $e = 2.718,\ \gamma = 0.577$
>
> Stirling's formula: $\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}(\frac{n}{e})^n} = 1$

2.6 常用复杂度表

常用数据结构

Data Structure	Time Complexity (Average)				Time Complexity (Worst)				Space Complexity (Worst)
	Access	Search	Insert	Delete	Access	Search	Insert	Delete	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))

Data Structure	Time Complexity (Average)				Time Complexity (Worst)				Space Complexity (Worst)
	Access	Search	Insert	Delete	Access	Search	Insert	Delete	
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

常用数组排序算法

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
Quicksort	Ω(n log(n))	Θ(n log(n))	$O(n^2)$	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	$\Theta(n^2)$	$O(n^2)$	O(1)
Insertion Sort	Ω(n)	$\Theta(n^2)$	$O(n^2)$	O(1)
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	$O(n^2)$	O(n)
Shell Sort	Ω(n log(n))	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	$O(n^2)$	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

3. 链表

> Array and Linked List

Linear Structure

predecessor 前驱

successor 后继

3.1 链表简介

List is a type of linear structure.

Elements in a list are of **the same data type**, and neighboring elements form ordered pairs.

Denote a list (size n) as $(a_0, a_1, \cdots, a_{n-1})$. For each $<a_{i-1}, a_i>$:

- a_{i-1} is the direct predecessor of a_i
- a_i is the direct successor of a_{i-1}

The number of elements n $(n \geq 0)$ in the list is called the **length** of list. List is called an empty when $n = 0$.

Each element in a non-empty list has a certain position, a_i is the i-th data element.

The length of list can be increased or decreased (Insert, delete, etc.)

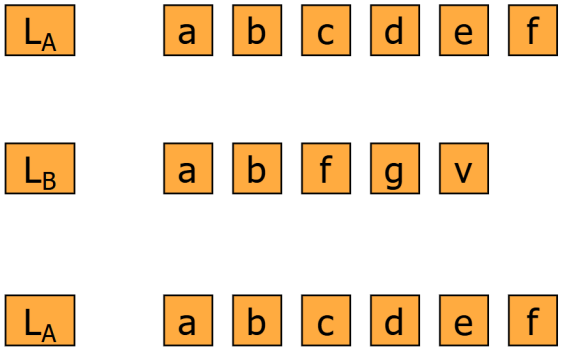
List 的操作

Operations	Condition	Result
InitList(&L)		create an empty List L
DestroyList(&L)	List L exists	destroy (free) List L
ClearList(&L)	List L exists	rest L to be an empty list
ListEmpty(L)	List L exists	return TRUE if L is empty, return FALSE otherwise
ListLength(L)	List L exists	return the number of elements in L
GetElem(L,i,&e)	List L exists, $1 \leq i \leq ListLength(L)$	return i-th element in L with value stored in e
LocateElem(L,e,compare())	List L exists, compare() is a predicate function	return the position of the first element which satisfies compare() with e. If none satisfies, then return -1
PriorElem(L,cur_e,&pre_e)	List L exists	if cur_e is an element in L, and not the first element, then return its predecessor in pre_e; otherwise, operation fails, pre_e is undefined.
NextItem(L,cur_e,&next_e)	List L exists	if cur_e is an element in L, and not the last element, then return its successor in next_e; otherwise, operation fails, next_e is undefined.
ListInsert(&L, i, e)	List L exists, $1 \leq i \leq ListLength(L)$	insert an element e at the i-th position in L; length of L increase by 1.
ListDelete(&L,i,&e)	List L exists, $1 \leq i \leq ListLength(L)$	delete the i-th element in L and return its value in e; length of L decrease by 1.
ListTraverse(L,visit())	List L exists	sequentially apply visit() to each element in L; the operation fails once visit() fails.

For the List Abstract Data Type (ADT) defined above, we can perform some more complex operations.

Example: Suppose we use 2 list L_A, L_B to represent set A and set B, respectively (i.e. data elements in the list are elements in the set), please compute a new set, such that $A = A \cup B$.

> 把 B 有 A 没有的元素插入 A，得到一个新集合（新链表）.



实现：For each element in L_B, check whether it is in L_A; if not, insert the element into L_A.

```
void union(List &La, List Lb) {
    La_len = ListLength(La);
    Lb_len = ListLength(Lb);
    for (int i=1; i <= Lb_len; i++) {
        GetElem(Lb, i, e);
        if (!LocateElem(La, e, equal()))
            ListInsert( La, ++La_len, e )
    }
}
```

3.2 数组实现

> Sequential representation of List: Array Implementation

store data elements in a List with a group of memory units of consecutive address.

Suppose each element takes M memory units, then

- $Location(a_{i+1}) = Location(a_i) + M$
- $Location(a_i) = Location(a_1) + (i-1)M$

Declaration

> 采用了动态内存分配 (Dynamic sequential storage allocation)

```
#define LIST_INIT_SIZE 100 // Initial storage size of List
#define LIST_INCREMENT 10 // Incremental storage size of List
Typedef struct {
    ElemType *elem; // Stores the Base Address
    int length; // Current length of List
    int listsize; // Current total memory allocation size of List
} SqList;
```

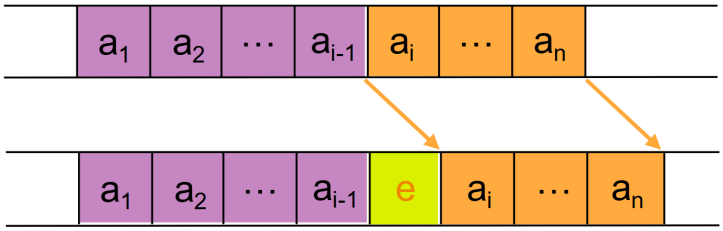
> `listsize` 以 `sizeof(Elemtype)` 为单位.
>
> Once not enough due to insertion, re-allocate storage, add additional storage of size `LIST_INCREMENT`.

Initialization

```
status InitList_Sq(SqList &L) {
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE*sizeof(ElemType));
    if(!L.elem) exit(OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
}
```

Insertion

> $(a_1,\cdots,a_{i-1},a_i,\cdots,a_n) \Rightarrow (a_1,\cdots,a_{i-1},e,a_i,\cdots,a_n)$
>
> $<a_{i-1},a_i> \Rightarrow <a_{i-1},e>,<e,a_i>$




```
status ListInsert_Sq(SqList &L, int i, ElemType e) {
    if ((i<1) || (i>L.length+1)) return ERROR;//注意i从1开始，数组下标从0开始
    if (L.length >= L.ListSize) {
        newbase = (ElemType *)realloc(L.elem, (L.ListSize+LISTINCREMENT)*sizeof(ElemType));
        if(!newbase) exit(OVERFLOW);
        L.elem = newbase;
        L.ListSize += LISTINCREMENT;
    }
    q = &(L.elem[i-1]);
    for(p=&(L.elem[L.length-1]); p>=q; --p)
        *(p+1) = *p;
    *q = e;
    ++L.length;
    return OK;
}
```

> 注意，`void *realloc(void *ptr, size_t new_size);` 重分配内存，当 `new_size` 小于原内存，只保证 `new_size` 以内的数据有效（尽量保留原数据）.

注意，逐地址循环

```
    q = &(L.elem[i-1]);
    for(p=&(L.elem[L.length-1]); p>=q; --p)
        *(p+1) = *p;
```

当中，`*(p+1)` 是正确写法，而不用写成 `*(p+sizeof(ElemType))`. 因为指针的算数运算会基于指针类型自动调整.

> 例如，如果 `p` 指向 `int` 类型，`p+1` 实际移动 `sizeof(int)` 个字节（一般为 4）.
>
> 如果 `p` 指向 `double` 类型，`p+1` 实际移动 `sizeof(double)` 个字节（一般为 8）.
>
> `p+1` 写成 `p+sizeof(ElemType)` 反而错误，它会移动 `sizeof(ElemType)*sizeof(ElemType)` 个字节.

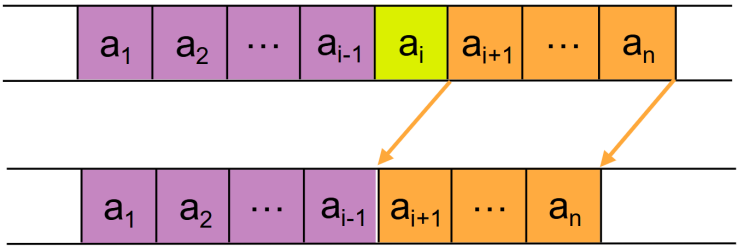
也可改为逐下标循环：

```
    q = i-1;
    for(p=L.length-1; p>=q; --p)
        L.elem[p+1] = L.elem[p];
```

Delete

> $(a_1,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n) \Rightarrow (a_1,\cdots,a_{i-1},a_{i+1},\cdots,a_n)$

> $<a_{i-1}, a_i>, <a_i, a_{i+1}> \Rightarrow <a_{i-1}, a_{i+1}>$




```
status ListDelete_Sq(SqList &L, int i, ElemType &e){
    if ((i<1) || (i>L.length)) return ERROR; // 注意和 Insert 的区别
    p = &(L.elem[i-1]);  // 取到要删除的元素地址
    e = *p;
    q = L.elem + L.length-1; // 地址运算，取到最后一个元素地址
    for(++p; p<=q; ++p)
        *(p-1) = *p;
    --L.length;
    return OK;
}
```

> 删除操作结束后末尾会有两个 a_n，但是不影响任何操作，因为 `--L.length`.

复杂度分析

> Time Complexity Analysis of Insert

Assume that P_i is the probability of inserting an element before the i-th element. Then when inserting an element to a list of size n, the average number of data element movement is:

$$E_{insert} = \sum_{i=1}^{n+1} P_i \cdot (n - i + 1)$$

Assume that inserting an element at any position is of equal probability, namely:

$$P_i = \frac{1}{n+1}$$

Hence,

$$E_{insert} = \sum_{i=1}^{n+1} \frac{1}{n+1} \cdot (n - i + 1) = \frac{n}{2}$$

Similarly, we can get:

$$E_{delete} = \sum_{i=1}^{n} \frac{1}{n} \cdot (n - i) = \frac{n-1}{2}$$

So, for a List of sequential storage structure, inserting or deleting an element requires moving about half of the data elements. If the list length is n, then `ListInsert_Sq` and `ListDelete_Sq` have time complexity of $O(n)$.

> 平均移动一半元素.

This is significant when the List length is large. This drawback is because sequential storage requires that all elements stored consecutively.

> 牵一发而动全身.

So sequential storage is usually used in cases where elements in the list is stable, requiring little insertion and delete.

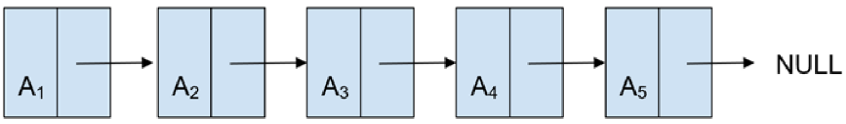
> 频繁增删不宜用数组实现. 数组的优势在于静态数据索引.

3.3 链表实现

Linked list: a basic data structure where each item contains the information that we need to get to the next item (the link)

> 在 C 语言中，可以使用指针实现.

Advantage: the capability to rearrange the items efficiently.



In theory, linked list can be infinite and cyclic

Ways to denote the end of linked list:

- NULL pointer (最常见)
- Dummy node (哑节点，不存储有意义的信息，仅用于避免特殊情况处理)

> 哑节点的值通常是一个占位值，如 0 或 NULL，链表中通常作为头节点或尾节点，用于简化插入、删除等操作. 优点是简化代码，无需单独处理边界，缺点是增大内存开销.

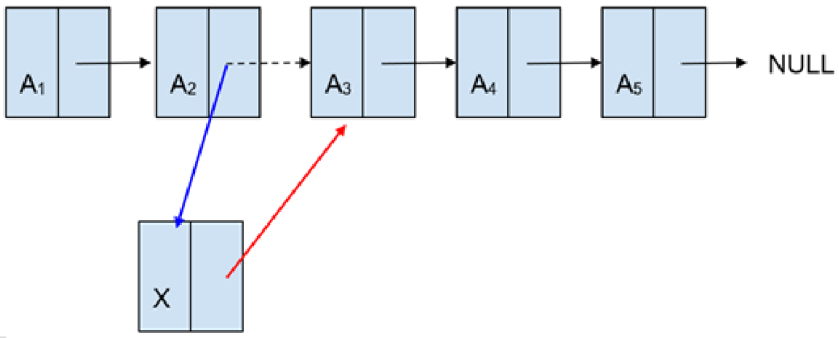
- First node, the list becomes circular.

Declaration

```c
typedef struct node {
    int data;
    struct node *next;
} Node;
```

Insertion

> Insertion is implemented by changing the links
>
> We need to search the position for the insertion



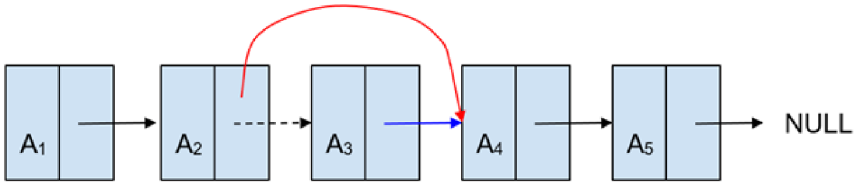

```c
// we need to return a new head of linkedlist because
// the head may change after insertion
Node* insert(Node* listHead, int dataToInsert) {
    if (listHead == NULL) {
        // list is empty, insert at first element
        Node *newNode = malloc(sizeof(Node));
        newNode->data = dataToInsert;
        newNode->next = NULL;
        return newNode;
    }
    // list is not empty, let's insert at the end
    Node *current;
    for (current = listHead; current != NULL; current = current->next) {
        if (current->next == NULL) { // if it's end of list...
            Node *newNode = malloc(sizeof(Node));
            newNode->data = dataToInsert;
```

```
            newNode->next = NULL;
            current->next = newNode;
            return listHead;
        }
    }
}
```

> 这里采用尾插法.

Deletion

> Deletion is also implemented by changing links
>
> Remember to release the memory (free) of the deleted nodes




```
// we need to return a new head of linkedlist because
// the head may be deleted
// Let's remove all nodes have the value dataToDelete
Node* delete(Node* listHead, int dataToDelete) {
    Node* current, * previous;
        previous = NULL;
        current = listHead;
    while (current != NULL) {
        if (current->data == dataToDelete) {
            if (listHead == current) { // if the listHead to be deleted
                listHead = current->next;
                free(current);
                current = listHead;
            } else { // if delete some middle nodes
                previous->next = current->next;
                free(current);
                current = previous->next;
            }
        } else { // if we do not delete this node
            previous = current;
            current = current->next;
        }
    }
    return listHead;
}
```

Find k-th element:

```
typedef struct node {
    int value;
    struct node * next;
} Node;

int find_kth(Node * listHead, int k) {
    for(int i=0; i<k; i++)
        listHead = listHead->next;
    return listHead->value;
}
```

> 这里 `listHead` 是形参，修改它不会影响真实的链表首地址（重指针才会影响）.
>
> 注意 k 从 0 开始.

3.4 数组实现 vs 链表实现

Operation	Array	Linked List
`create()`	$O(1)$	$O(1)$
`insert()`	$O(n)$	$O(1)$
`delete()`	$O(n)$	$O(n)$
`find()`	$O(n)$	$O(n)$
`find_kth()`	$O(1)$	$O(n)$
`previous()`	$O(1)$	$O(n)$
`free()`	$O(1)$	$O(n)$

> 注意链表的 `delete()`，如果已知待删除节点的指针，复杂度只有 $O(1)$；如果只知道要删的值，就要额外进行查找，复杂度 $O(n)$.

3.5 特殊链表

① 循环链表

> Circular Linked List

Connect the last node to the first node

② 双向链表

> Double Linked List

Add a link to the previous node

3.6 矩阵

> Matrix Abstract Data Type

A matrix is a table of elements.

(2D) Matrix Illustrations

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ \vdots \\ m \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \dots & n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \end{array}$$

$$\begin{bmatrix} 0 & 1 & 5 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 9 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 0 & 8 \end{bmatrix}$$

Sparse matrix

矩阵操作

Operations	Content
`print_matrix()`	print all elements of the matrix
`clear_matrix()`	clear all elements of the matrix, and return zero matrix
`addition()` `multiplication()` `dot_product()`	some basic numerical operations of matrix
`insert()` `delete()`	insert or delete an element from the matrix
`scale()` `rotation()`	some basic transformation of matrix
`find_pos()`	return the element at position

矩阵实现

> Matrix Implementation

① 二维数组实现

> 2D array implementation

Declaration

```
int matrix[col][row];
```

> 基础的二维数组，不重点讨论.

② 二维链表实现

> 2D linked list implementation

Declaration

```
typedef struct node {
    int data;
    struct node *next;
} Node;

typedef struct row {
    Node * row_start;
    struct row *next;
} Mat;
```

③ 邻接表实现

> An array of linked list implementation (Also known as adjacency list, will be discussed in graph)

Declaration

```
typedef struct node {
    int data;
    struct node *next;
} Node;

Node* mat[row];
```

④ 三元组数组实现

> An array of trio implementation
>
> trio 三元组，第三项存值.

(*x*, *y*, *value*)
(1, 2, 1)
(1, 5, 1)
(2, 1, 1)
(2, 3, 1)
(2, 4, 1)

```
typedef struct trio {
    int row;
    int column;
    int value;
} Trio;

Trio Mat[size];
```

> `size` 是矩阵总元素个数.

⑤ 三元组链表实现

> A linked list of trio implementation

```
typedef struct trio {
    int row;
    int column;
    int value;
} Trio;

typedef struct node {
    Trio data;
    struct node *next;
} Node;
```

4. 排序算法

> sorting
>
> 这里讨论升序排序，降序思路相同，只要改一下符号.

We are going to study methods of sorting data $items$ containing $keys$

Each $item$ (record) contains a key, which is a small part of the item, and is used to control the sort.

> The remainder of the item is called satellite data

Sorting: rearrange the items such that their keys are ordered

> e.g. numerical order, alphabetical order, lexicographical order

Internal Sort: sorting takes place entirely within the main memory of the computer.

External Sort: handle massive amounts of data that cannot fit into the main memory. It requires a slower kind of memory (like hard-disk) for the sorting.

We assume the following in our discussion of the sorting algorithms:

- Internal - Each algorithm will be passed an array containing the elements and an integer containing the number of elements stored entirely in the main memory.
- Validity - We will assume that n, the number of elements passed to our sorting routines, has already been checked and is legal.
- Ordering - We require the existence of the `<` and `>` operators, which can be used to place a consistent ordering on the input.

Internal sorting is the process of gradually expanding the length of ordered sequence of elements.

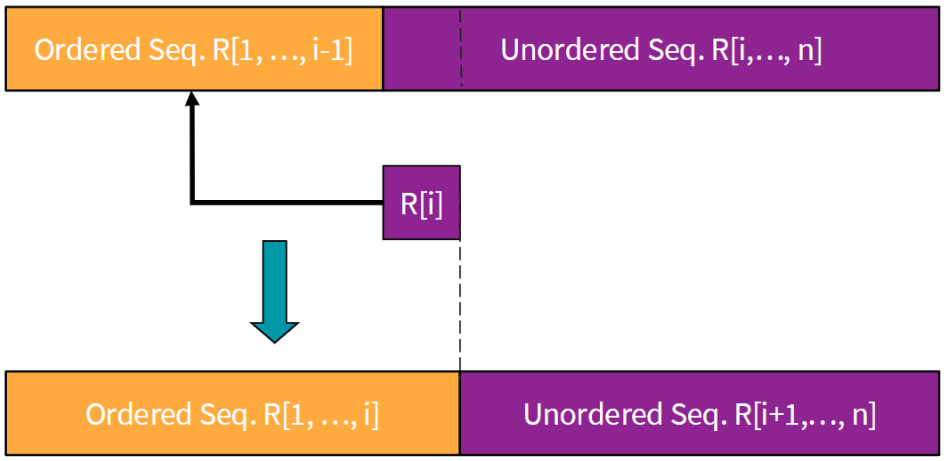
Different ways that the ordered sequence gets expanded:

> 排序算法的四大核心策略：插入、交换、选择、合并

- Insertion: Insert one or multiple items from the unordered elements into the ordered sequence.
- Swap: swapping records in the unordered sequence to find the one with the largest or smallest key, and add it into the ordered sequence.
- Selection: from the unordered sequence, find the record with the smallest or largest key, and add it to the ordered sequence.
- Merging: merging two or more ordered sequences.
- Others

4.1 插入

> 从左往右排



Three Steps for Insertion

- Find the insertion position for $R[i]$ in $R[1,\ \ldots,\ i-1]$

$$R[1,\ \ldots,\ j].key \leq R[i].key \leq R[j+1,\ \ldots,\ i-1].key$$

- Move all records in $R[j+1,\ \ldots,\ i-1]$ back by one position.
- Insert (copy) the value of $R[i]$ to $R[j+1]$

① 插入排序

> Insertion Sort

Consider the items one at a time.

Insert each into its proper place among already considered (and sorted)

In computer, we need to make space for moving larger items one position to the right.

Clearly, insertion sort consists of $n-1$ passes.

- For pass $p=2$ through n, insertion sort ensures that the elements in position 1 through p are in sorted order.
- Insertion sort makes use of the fact: Elements in positions 1 through $p-1$ are already in sorted order.

Illustration

iter	34	8	64	51	32	21	#swap	#comp.
1	8	34	64	51	32	21	1	1
2	8	34	64	51	32	21	0	1
3	8	34	51	64	32	21	1	2
4	8	32	34	51	64	21	3	4
5	8	21	32	34	51	64	4	5

Main Idea: At *i*-th iteration, insert the *i*-th element to its right position

```
void insertionsort(int n, int a[]){
    int i, j, tmp;
    for (i = 1; i < n; i++){
        tmp = a[i];
        for (j = i; j > 0 && a[j - 1] > tmp; j--)
            a[j] = a[j - 1]; // shift down
        a[j] = tmp;          // the insert
    }
}
```

> 从第二个元素开始循环，每次插一个到前面已排序好的序列.
>
> 升序排序.

Initially $a[0]$ may be thought of as a sorted array of one element.

After k iterations, the elements $a[0]$ through $a[k]$ are in order.

Time complexity in various cases

- If the initial array is sorted, only one comparison is made on each iteration, so that the sort is $O(n)$.

> 就是对一个已经排序好的数组再次排序.

- If the array is initially sorted in the reversed order, the sort is $O(n^2)$, since the total number of comparisons is

$$1+2+\cdots+(n-1)=\frac{n(n-1)}{2}=O(n^2)$$

> 复杂度波动很大. 可能很快，也可能很慢.

The space requirements consists of only one temporary variable tmp $O(1)$

② 二分插入排序

> Binary Insertion Sort
>
> 在普通插入排序基础上，每次插入时改进为二分查找插入.

```
void binaryInsertionSort(int array[], int size) {
    for (int i = 1; i < size; i++) {
        int key = array[i]; // 当前要插入的元素
        int left = 0, right = i - 1;

        // 使用二分查找法找到插入位置
        while (left <= right) {
            int mid = (left + right) / 2;
            if (key < array[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
```

```
        }

        // 将插入位置后的元素整体右移一位
        for (int j = i - 1; j >= left; j--) {
            array[j + 1] = array[j];
        }

        // 将元素插入到正确位置
        array[left] = key;
    }
}
```

> 找到插入位置时，left 在 right 右边一个单位，且 left 是要插入的位置.
>
> 思路：比中间，移两边，不断循环.

时间复杂度：$O(n^2)$

> 虽然查找时间减少，但是插入仍然需要移动元素（数组无法克服的问题）.
>
> 如果只考虑 $n-1$ 次二分查找，只有 $O(n\log n)$，但是考虑插入后就是 $O(n^2)$.
>
> 每次二分查找复杂度为 $O(\log n)$

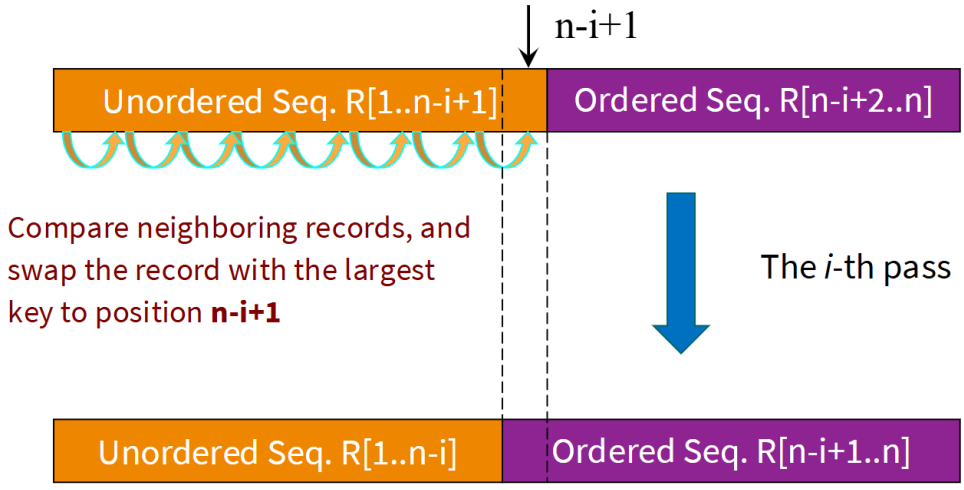
③ 希尔排序

> Shell Sort，自学.

4.2 交换

① 冒泡排序

> Bubble Sort



```c
void bubbleSort(int array[], int size) {
    for (int i = 0; i < size - 1; i++) { // 外层循环控制遍历次数
        for (int j = 0; j < size - 1 - i; j++) { // 内层循环控制比较和交换
            if (array[j] > array[j + 1]) {
                // 交换相邻的两个元素
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}
```

> 每次内层循环，至少有一个被放到正确位置（未排序序列中最大的那个）.
>
> 未排序序列中倒数第二大的不一定被放到正确位置，因为可能在中间某次与最大值比较时被扔到左边了.
>
> 从右往左排，大的先排好.

iter	34	8	64	51	32	21	#swap	#comp.
1	8	34	51	32	21	64	4	5
2	8	34	32	21	51	64	2	4
3	8	32	21	34	51	64	2	3
4	8	21	32	34	51	64	1	2
5	8	21	32	34	51	64	0	1

Main Idea: The largest number (bubble) floats to the end of the array in each iteration

时间复杂度：$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}=O(n^2)$

可以通过标志位来优化算法：

```c
void bubbleSortOptimized(int array[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int swapped = 0; // 标志位，记录是否发生交换
        for (int j = 0; j < size - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                swapped = 1; // 如果发生交换，设置标志位
            }
        }
        if (swapped == 0) {
            break; // 如果没有发生交换，说明数组已排序，提前退出
        }
    }
}
```

> The end condition is that: there is no swapping in the last pass.
>
> 该优化对于即将排序好的初始序列十分有效，最佳情况只需 $O(n)$.

时间复杂度：最佳 $O(n)$，最差 $O(n^2)$.

Bubble sort requires little additional space

> little 是很少的意思，因为它是一种原地排序算法（In-place Sorting Algorithm）

② 快速排序

> QuickSort
>
> 核心思想：分治法

```c
// 分区函数：将数组划分为两部分，并返回基准的最终位置
int partition(int array[], int low, int high) {
    int pivot = array[high]; // 选择最后一个元素作为基准
    int i = low - 1;         // i 指向小于基准的最后一个元素

    for (int j = low; j < high; j++) {
        if (array[j] < pivot) { // 如果当前元素小于基准
            i++;
            // 交换 array[i] 和 array[j]
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }

    // 把基准放到正确的位置（即 i + 1）
    int temp = array[i + 1];
    array[i + 1] = array[high];
    array[high] = temp;

    return i + 1; // 返回基准的最终位置
}

// 快速排序函数
void quickSort(int array[], int low, int high) {
    if (low < high) {
        // 分区操作，返回基准位置
        int pivotIndex = partition(array, low, high);

        // 对基准左侧和右侧递归排序
        quickSort(array, low, pivotIndex - 1);
        quickSort(array, pivotIndex + 1, high);
    }
}
```

> j 循环保证了循环结束时，i 及 i 前面的都是小于基准的，i+1及后面的都是大于等于基准的（因为只要有小于的，就会被调到 i++ 的位置）. 而基准又是未元素，因此循环结束后交换基准和 i+1 然后返回 i+1 即为基准位置.
>
> 注意所有操作都在同一个 array 上进行，也不需要额外空间.
>
> 注意分治法要设置结束条件，即 `low >= high` 是结束（实际上有两种情况，`low==high` 单元素排序和 `low==high+1` 没有元素可以排序，遇到这两种情况分治法的一个分支就结束了）.

Quicksort is one of the fastest known sorting algorithms in practice.

Its worse-case performance is $O(n^2)$ and its average runtime is $O(n \log n)$.

A divide-and-conquer recursive algorithm

Quicksort algorithm:

- If the number of elements in S is 0 or 1, then return
- If the number of elements in S is 2, then direct comparison and return
- Pick any element v in S as the pivot.

  > 基准选择：可以随意.

- Partition $S' = S - \{v\}$ into two disjoint groups

  > 一个都比基准小，一个都比基准大.
  >
  > $S_1 = \{x \in S' | x < v\} \quad S_2 = \{x \in S' | x \geq v\}$

- Return $\{quicksort(S_1),\ v,\ quicksort(S_2)\}$

> 分而治之，递归.

Ideally we want half the keys to go into S_1 and another half into S_2.

The choice of pivot can strongly affect the relative sizes of the partitions

We can implement the partitioning to be performed in place and very efficiently.

> 原地排序，不需要额外空间.

Ways to select pivot:

- The first/last element: bad choice. Keep making bad partitioning when the array is sorted or reversed.
- A random pivot: safe, but you take extra time to generate the random number.
- Median-of-Three Partitioning: pick the median of $a[0], a[n-1]$ and $a[\frac{n}{2}]$ to be the pivot.

> $\frac{n}{2}$ 和 $\frac{n-1}{2}$，只在偶数个元素时有区别，一个定位在中点右侧，一个在左侧.

三数取中法

> Median-of-Three，QuickSort 的基准选择策略.

```
// 三数取中法选择基准
int medianOfThree(int array[], int low, int high) {
    int mid = (low + high) / 2;

    // 比较并交换，确保 array[low] <= array[mid] <= array[high]
    if (array[low] > array[mid]) {
        int temp = array[low];
        array[low] = array[mid];
        array[mid] = temp;
    }
    if (array[low] > array[high]) {
        int temp = array[low];
        array[low] = array[high];
        array[high] = temp;
    }
    if (array[mid] > array[high]) {
        int temp = array[mid];
        array[mid] = array[high];
        array[high] = temp;
    }

    // 此时 array[mid] 是中位数，将其作为基准
    return mid;
}

// 分区函数
int partition(int array[], int low, int high) {
    // 使用三数取中法选择基准
    int pivotIndex = medianOfThree(array, low, high);
    int pivot = array[pivotIndex];

    // 将基准放到最后位置
    int temp = array[pivotIndex];
    array[pivotIndex] = array[high];
    array[high] = temp;

    int i = low - 1; // i 指向小于基准的最后一个元素
    for (int j = low; j < high; j++) {
        if (array[j] < pivot) {
            i++;
            // 交换 array[i] 和 array[j]
            temp = array[i];
            array[i] = array[j];
            array[j] = temp;
```

```
        }
    }

    // 将基准放到正确位置
    temp = array[i + 1];
    array[i + 1] = array[high];
    array[high] = temp;

    return i + 1; // 返回基准的最终位置
}

// 快速排序函数
void quickSort(int array[], int low, int high) {
    if (low < high) {
        // 分区操作，返回基准位置
        int pivotIndex = partition(array, low, high);

        // 对基准左侧和右侧递归排序
        quickSort(array, low, pivotIndex - 1);
        quickSort(array, pivotIndex + 1, high);
    }
}
```

> 三数取中后，把中间数和末尾对调，接下来的方法和原来一样.
>
> 尽量使基准左右两侧个数相近，能有效防止极端情况.

时间复杂度:

最优 $O(n \log n)$

平均 $O(n \log n)$

最坏 $O(n^2)$

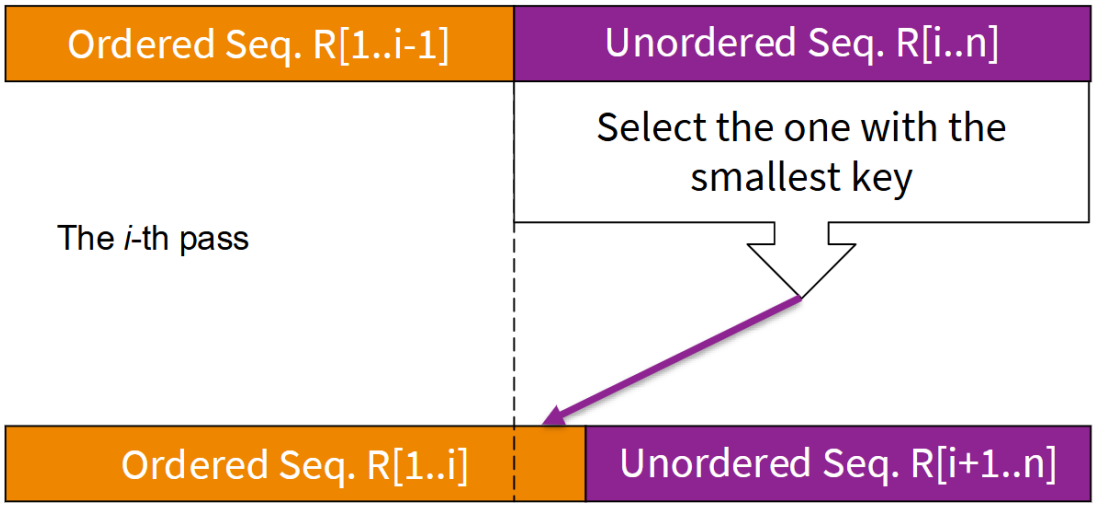
> 最坏情况出现的概率显著降低.

Quicksort Analysis - Average Case

> Self study

4.3 选择

① 选择排序



每次在未排序序列中选出最小的，接在已排序序列最右边.

与插入不同的是，把耗时的过程从寻找插入位置转到寻找待插值.

```c
#include <stdio.h>

// 选择排序函数
void selectionSort(int array[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i; // 假设当前元素是最小值
        for (int j = i + 1; j < size; j++) {
            if (array[j] < array[minIndex]) {
                minIndex = j; // 找到更小的值
            }
        }
        // 交换当前元素和找到的最小值
        int temp = array[minIndex];
        array[minIndex] = array[i];
        array[i] = temp;
    }
}
```

> 选择排序要从 index 0 开始，即初始已排序序列是空的.

The algorithm consists of a selection phase in which the smallest of the remaining elements, min, is repeatedly placed in its proper position, i

min is swapped with the element $a[i]$

The initial n-element unordered sequence is reduced by one element after each selection

The first pass makes $n-1$ comparisons, the second pass makes $n-2$, and so on

Therefore, total number of comparisons is

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}=O(n^2)$$

4.4 合并

① 归并排序

> Merging: Merge Sort

```c
// 合并两个子数组的函数
void merge(int array[], int left, int mid, int right) {
    int n1 = mid - left + 1;  // 左子数组的大小
    int n2 = right - mid;     // 右子数组的大小

    // 创建临时数组
    int leftArray[n1], rightArray[n2];

    // 将数据复制到临时数组
    for (int i = 0; i < n1; i++) {
        leftArray[i] = array[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArray[j] = array[mid + 1 + j];
    }

    // 合并临时数组到原数组
    int i = 0, j = 0, k = left; // i 是左子数组索引，j 是右子数组索引，k 是合并后数组的索引
    while (i < n1 && j < n2) {
        if (leftArray[i] <= rightArray[j]) {
            array[k] = leftArray[i];
            i++;
        } else {
            array[k] = rightArray[j];
            j++;
        }
        k++;
```

```
    }

    // 复制剩余的元素（如果有）
    while (i < n1) {
        array[k] = leftArray[i];
        i++;
        k++;
    }
    while (j < n2) {
        array[k] = rightArray[j];
        j++;
        k++;
    }
}

// 归并排序函数
void mergeSort(int array[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2; // 计算中点（避免溢出）

        // 递归对左右两部分排序
        mergeSort(array, left, mid);
        mergeSort(array, mid + 1, right);

        // 合并两个有序数组
        merge(array, left, mid, right);
    }
}
```

> 合并前已经保证了待合并的两个子序列是有序的.
>
> 复制剩余元素是因为左右两部分可能长度略有不同. 长的那个会多出来元素，直接复制过去即可，因为合并时两个子数组已经是有序的了.
>
> 注意分治法递归要设计结束条件. 这里当 `left>=right` 时结束，这里实际应该只有 `left==right` 的情况，对应双元素拆开成两个单元素，然后没法继续分，然后开始合并、返回.

The classic example of divide and conquer strategy:

- Divide: Split the array into two halves and solve the sorting problem independently
- Conquer: Merge the sorted sub-array to get back a whole sorted array.

The fundamental operation is merging two sorted lists

Since the lists A and B are sorted, the merging can be done by simply a linear can through the input and put the output to an auxiliary list C.

> 注意这里要创建额外的辅助数组. 上面的代码中，两个辅助数组用于存放子数组. 排序操作在原数组上进行.

时间复杂度：$O(n \log n)$

> $O(\log n)$ 层分解，每层合并要 $O(n)$ 时间.

数学推导：

$T(n) = 2T(\frac{n}{2}) + cn \Rightarrow \frac{T(n)}{n} = \frac{T(\frac{n}{2})}{\frac{n}{2}} + c$

解得 $T(n) = O(n \log n)$

> 也可以直接主定理. 见 `2.4 主定理`.

虽然 merge sort 时间复杂度小，但是 it uses up $O(n)$ extra space.

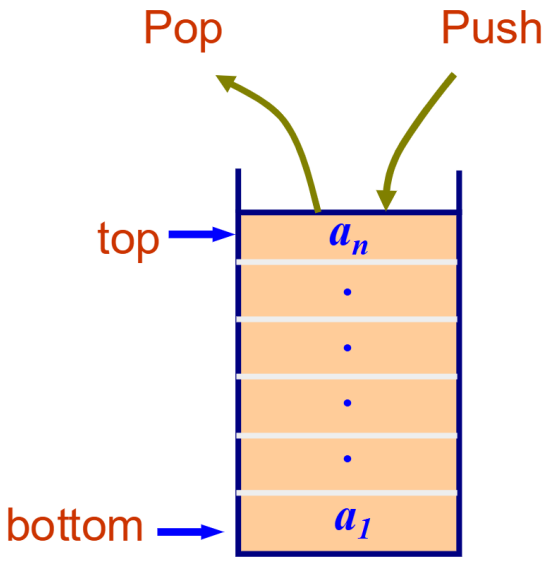
5. 栈和队列

> Stack and Queue，堆栈和队列

Stack and Queue are both linear structures, but with limited operations.

5.1 栈

> Stack
>
> 后进先出（Last In First Out, LIFO）



One side to insert and delete values: Stack Top

The other side: Stack Bottom

Push：将元素压入堆栈.

Pop：将元素从堆栈中弹出.

Peek/Top：查看堆栈顶部的元素，但不弹出.

IsEmpty：检查堆栈是否为空.

Operations:

```
InitStack(&S);
DestroyStack(&S);
ClearStack(&S);
StackEmpty(S);
StackLength(S);
Top(S,&e); // Get top value
Push(&S,e); // Insert to top
Pop(&S,&e); // Delete from top
StackTraverse(S,visit());
```

Operation	Stack	Return value
1. push('C')	C	-
2. push('S')	C S	-
3. push('C')	C S C	-
4. push('I')	C S C I	-
5. pop()	C S C	-
6. top()	C S C	C
7. push('E')	C S C E	-

Operation	Stack	Return value
8. pop()	C S C	-
9. pop()	C S	-

5.1.1 数组实现

> Stack: Array Implementation

Pros: Less pointer manipulations
Cons: Need to declare an array size ahead of time

Key idea:

- tos (variable) is defined to be -1 when the stack is empty
- push(x) - increase tos and assign x to stack[tos]
- pop() - decrease tos
- top() - return stack[tos]

详细操作:

1. Declaration

```
#define EMPTY_TOS -1
#define MAX_SIZE 100

typedef struct stack_s {
    int tos; // top of stack, 顶部元素对应下标，从0开始.空堆栈的tos是-1
    int data[MAX_SIZE];
} Stack;

int stack_is_empty(Stack* s) {
    if (s->tos == EMPTY_TOS)
        return 1;
    return 0;
}
```

2. Init, Clear

```
void stack_init(Stack **s){     // create a new stack
    *s = (Stack*) malloc(sizeof(Stack));
    stack_make_empty(*s);
}

void stack_make_empty(Stack *s){ // make the stack empty
    s->tos = EMPTY_TOS;
    memset(s->data, 0, sizeof(char)*MAX_SIZE);
}
```

3. Push

```
int stack_push(Stack* s, int x){
    if (s->tos+1 >= MAX_SIZE){
        perror("stack is full.\n");
        return -1; // return -1 if unsuccessful
    }
    s->tos++;
    s->data[s->tos] = x; // 新元素插入顶部，后进先出
    return s->tos; // return size of stack if successful
}
```

4. Top, Pop

```
int stack_top(Stack* s){
    if (!stack_is_empty(s))
        return (s->data[s->tos]);
    return INT_MIN; // upon error, return some undefined value (???直接报错)
}

void stack_pop(Stack* s){ // One loophole here, what is it? （空堆栈pop会出问题）
    s->tos--;
}
```

> pop 只要修改 tos 的值，不用真的抹去顶部元素.
>
> 下次 push 到这里的时候会自动覆盖原先内容.

5. topandpop, Free

```
int stack_topandpop(Stack* s){
    if (!stack_is_empty(s))
        return (s->data[s->tos--]);
    return INT_MIN;
}

void stack_free(Stack** s){
    free(*s);
    *s = 0;
}
```

> `*s=0` 和 `*s=NULL` 等价. 但推荐用 NULL.

5.1.2 链表实现

> Linked List Implementation

Pros: The size of stack is flexible
Cons: Pointer manipulations may be a problem to program beginner

Key Idea:

```
Keep a header node
• push(x) - insert the element x to the front
• pop() - delete the element at the front
• top() - return the element at the front
```

详细操作：

> 各种操作前，注意检查栈是否为空栈.

1. Declaration

```
typedef struct stack_s {
    int data;
    struct stack_s *next;
} Stack;

int stack_is_empty(Stack* s){
    return (s->next == NULL);
}
```

2. Init, Clear

```
void stack_init(Stack **s){
    *s = malloc(sizeof(Stack));
    (*s)->next = NULL;
    (*s)->data = INT_MIN; // INT_MIN 是在 <limits.h> 头文件中定义的，表示 int 类型的最小值.
}

void stack_make_empty(Stack* s){
    if (s == NULL) return;
    while (!stack_is_empty(s))
        stack_pop(s);
}
```

> 栈头不储存有意义的值，但是占有一个节点. 它指向的节点是第一个有意义的节点（且为栈的顶端元素 top）.

3. Push

```
void stack_push(Stack* s, int x){ //这里放的*s是堆栈头部，和链表的任意位置插入有区别
    Stack *t = malloc(sizeof(Stack));
    t->data = x;
    t->next = s->next;
    s->next = t;
}
```

> 为了满足后入先出的性质，push 和 pop 都对 `*s` 指向的节点（top）进行操作.

4. Pop, Top

```
int stack_top(Stack* s){
    if (!stack_is_empty(s)) return (s->next->data);
    return INT_MIN; // raise error value
}

void stack_pop(Stack* s){
    Stack *t = s->next;
    if (!stack_is_empty(s)){
        s->next = t->next;
        free(t);
    }
}
```

5.1.3 应用

Base Conversion

Balancing Symbols

> 检查小括号、中括号是否符合语法
>
> We can use a stack to help checking:
> • If a character is an opening symbol, push it onto the stack
> • If it is a closing, pop the stack and check if it matches to top

Line Editing

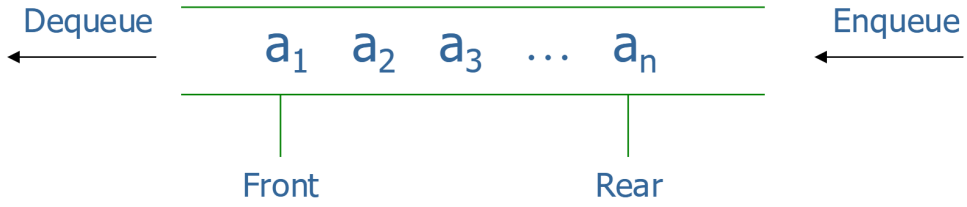
Maze

> 走迷宫

Postfix Evaluation

5.2 队列

> Queue
>
> 先进先出（first-in, first-out, FIFO）



Enqueue：将元素加入队列.

Dequeue：将元素从队列中移除，并返回被移除的值.

Front：查看队列前端的元素，但不移除.

IsEmpty：检查队列是否为空.

```
InitQueue(&Q);
DestroyQueue(&Q);
ClearQueue(&Q);
QueueEmpty(Q);
QueueLength(Q);
GetHead(Q,&e);
EnQueue(&Q,e);
DeQueue(&Q,&e);
```

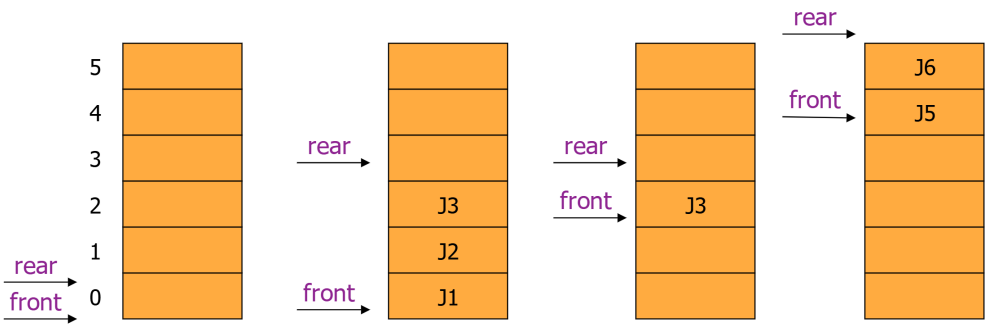
Operation	Queue	Return value
1. enqueue(‘C’)	C	-
2. enqueue(‘S’)	C S	-
3. enqueue(‘C’)	C S C	-
4. enqueue(‘I’)	C S C I	-
5. dequeue()	S C I	C
6. dequeue()	C I	S
7. enqueue(‘E’)	C I E	-
8. dequeue()	I E	C
9. dequeue()	E	I

5.2.1 循环数组实现

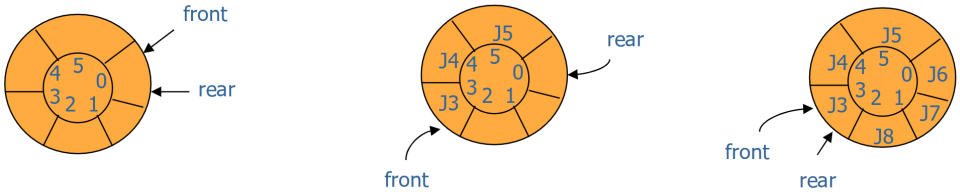
> Queue: Array Implementation with Circular Array

Queue: Naïve Array Implementation

Space can get unused...



普通数组会浪费很多空间. 考虑循环数组（Circular Array）：




```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<limits.h>

#define MAX_SIZE 100  // 定义队列的最大容量为100

// 定义队列结构体
struct queue {
    int data[MAX_SIZE];  // 存储队列数据的数组
    int front;           // 队列头部索引
    int rear;            // 队列尾部索引
    int size;            // 队列当前大小
};
typedef struct queue Queue;  // 为结构体定义别名 Queue

// 初始化队列
void queue_init(Queue** q) {
    Queue* tmpq = malloc(sizeof(Queue));  // 为队列分配内存
    queue_make_empty(tmpq);  // 将队列初始化为空
    *q = tmpq;  // 将初始化后的队列地址赋值给指针
}

// 将队列置为空
void queue_make_empty(Queue* q) {
    if (q != NULL) {
        q->front = 0;  // 队列头部索引置为0
        q->rear = 0;   // 队列尾部索引置为0
        q->size = 0;   // 队列当前大小置为0
    }
}

// 获取队列的当前大小
```

```c
int queue_size(Queue* q) {
    if (q == NULL)
        return 0;  // 如果队列为空，返回0
    return q->size;  // 返回队列的当前大小
}

// 判断队列是否为空
int queue_is_empty(Queue* q) {
    if (q->size == 0)  // 如果队列大小为0
        return 1;      // 返回1，表示队列为空
    else
        return 0;      // 返回0，表示队列不为空
}

// 判断队列是否已满
int queue_is_full(Queue* q) {
    if (q->size == MAX_SIZE)  // 如果队列大小等于最大容量
        return 1;             // 返回1，表示队列已满
    else
        return 0;             // 返回0，表示队列未满
}

// 向队列中添加元素
void queue_enqueue(Queue* q, int x) {
    if (queue_is_full(q)) {  // 如果队列已满
        return;              // 不执行任何操作
    }
    q->data[q->rear] = x;              // 在队尾位置插入元素
    q->rear = (q->rear + 1) % MAX_SIZE;  // 更新队尾索引，循环队列
    q->size++;                          // 队列大小加1
}

// 从队列中取出元素
int queue_dequeue(Queue* q) {
    int x;
    if (queue_is_empty(q)) {  // 如果队列为空
        return INT_MAX;       // 返回一个特殊值表示错误（INT_MAX表示无效值）
    }
    x = q->data[q->front];           // 获取队列头部元素
    q->front = (q->front + 1) % MAX_SIZE;  // 更新队列头部索引，循环队列
    q->size--;                       // 队列大小减1
    return x;                        // 返回取出的元素
}

// 释放队列内存
void queue_free(Queue** q) {
    if (*q != NULL) {
        free(*q);  // 释放队列的内存
        *q = NULL; // 将指针置为空，避免悬空指针
    }
}

// 打印队列内容
char* queue_print(Queue* q) {
    int len, index = q->front;  // len用于存储字符串的长度，index用于遍历队列
    char *output;  // 指向存储队列内容的字符串
    len = q->size * 12 + 15;  // 预估字符串所需的内存大小
    output = malloc(sizeof(char) * len);  // 为字符串分配内存
    memset(output, 0, sizeof(char) * len);  // 初始化字符串
    output[0] = '\0';  // 确保字符串以空字符开始

    strcat(output, "(front) ");  // 添加“(front)”标签表示队列起始

    for (int i = 0; i < q->size; i++) {  // 遍历队列
        char temp[12];  // 临时缓冲区用于存储每个元素
        snprintf(temp, sizeof(temp), "%d ", q->data[index]);  // 将队列元素转换为字符串
        strcat(output, temp);  // 将元素拼接到最终字符串中
        index = (index + 1) % MAX_SIZE;  // 更新索引（循环队列）
```

```
    }

    strcat(output, "(rear)");  // 添加“(rear)”标签表示队列结束
    return output;  // 返回生成的字符串
}
```

5.2.2 链表实现

Declaration

```
// Array implementation
#define MAX_SIZE 100

typedef struct queue_s {
    int data[MAX_SIZE];
    int front;
    int rear;
    int size;
} Queue;

// Linked list implementation
typedef struct node {
    int data;
    struct node*next;
} Node;

typedef struct queue_s{
    int size;
    Node* front, *rear;
} Queue;
```

Init

```
// Array implementation
void queue_init(Queue **q){
    *q = malloc(sizeof(Queue));
    queue_make_empty(*q);
}

// Linked list implementation
void queue_init(Queue **q){
    *q = malloc(sizeof(Queue));
    (*q)->front = NULL;
    (*q)->rear = NULL;
    (*q)->size = 0;
}
```

Is Empty

```
// Array implementation
int queue_is_empty(Queue* q){
    return q->size == 0;
}

// Linked list implementation
int queue_is_empty(Queue* q){
    return q->size == 0;
}
```

Enqueue

```
// Array implementation
void queue_enqueue(Queue* q, int x){
    if (queue_is_full(q)){
        printf("Error"\n");
        return;
    }
    q->size++;
    q->rear = (q->rear + 1) % MAX_SIZE; // front和rear可能同时平移，导致rear数值超出max但是实际上queue还没full，这个时候对max_size取模可以把值插回前面（成环）
    q->data[q->rear] = x;
}

// Linked list implementation
void queue_enqueue(Queue* q, int x){
    Node *tmp;
    tmp = malloc(sizeof(Node));
    tmp->data = x;
    tmp->next = NULL;
    q->size++;
    if (q->size == 1) {
        q->front = q->rear = tmp; // 链式赋值，从右到左
        return;
    }
    q->rear->next = tmp;
    q->rear = tmp;
}
```

> 插入第一个节点时，头尾都指向该节点. 之后有新节点加入，移动尾节点指针. 有节点移除，移动头节点指针.

Dequeue

```
// Array implementation
int queue_dequeue(Queue* q){
    int x;
    if (queue_is_empty(q))
        return INT_MIN;
    q->size--;
    x = q->data[q->front];
    q->front = (q->front + 1) % MAX_SIZE;
    return x;
}

// Linked list implementation
int queue_dequeue(Queue* q){
    int x; Node *tmp;
    if (queue_is_empty(q))
        return INT_MIN;
    if (q->size == 1)
        q->rear = NULL;
    q->size--;
    x = q->front->data;
    tmp = q->front;
```

```
    q->front = q->front->next;
    free(tmp);
    return x;
}
```

other operations

- Make the queue empty
- Free the queue
- Print the queue
- Check whether the queue is full or not (Circular array implementation)

5.2.3 应用

> Applications

▪ Printer queues

▪ Process queues

▪ Customer service ticket systems

▪ Anything that requires FIFO (First In, First Out)

6. 哈希

> Hashes

Many applications requires a dynamic set that supports only the dictionary operations insert, search and delete

- The set is often called a symbol table (ST)

A hash table is an effective data structure for implementing dictionaries.

Though searching an element in a hash table can take as long as $\Theta(N)$ in the worse case, the expected time to search for an element is only $O(1)$.

6.1 字典

> Dictionary Abstract Data Type

Direct-Address Tables

> 直接寻址表，key 值直接作为索引

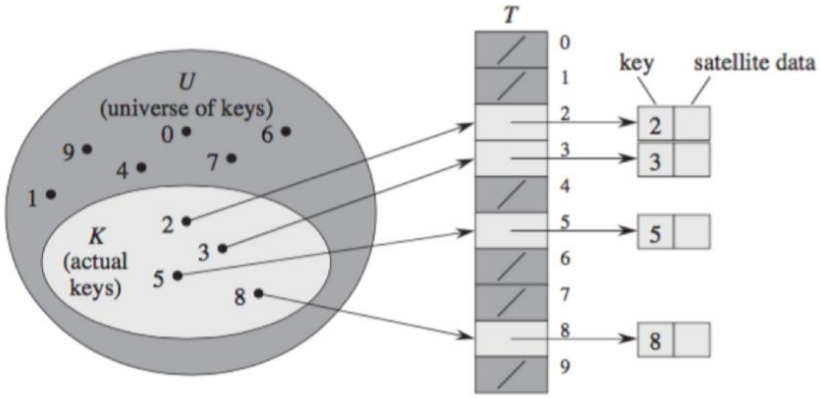
In direct address tables, records are placed using their key values directly as indexes

Direct addressing is a simple technique that works well when the universe U (the complete set) of keys is reasonably small

> key 的全集相对较小，可以直接索引，否则占内存

Suppose that an application needs a dynamic set in which each element has a key drawn from U = {0, 1, ..., M - 1} and M is not too large

If we assume no two elements have the same key, then we can use an array, or direct-address table, in which each position (slot) corresponds to a key in U



Direct-address Table

A Dictionary is used to store data values in `(key, value)` pairs.
Value can be an object (struct).
Operations:

- search(key): return the value based on the key
- insert(key, value): insert the (key, value) pair into the dictionary
- delete(key): delete the value based on the key
- contain(key): return true when dictionary contains the key
- size(): return the total number of (key, value) pairs in the dictionary
- keys(), values(): return all keys / values in the dictionary
- entries(): return all (key, value) pairs in the dictionary

Time complexity of the elementary operations is Θ(1)

- Insert, Search, Delete, Contain

Storage of the elements

- With the direct-address table itself
- Externally with a pointer from a slot in the table

6.2 哈希表与哈希函数

> Hash Table and Hash Function

Hash Tables

Hash table is one of the data structure for dictionary

The difficulty with direct addressing is obvious: if the universe U is large, storage a table of size |U| is impractical

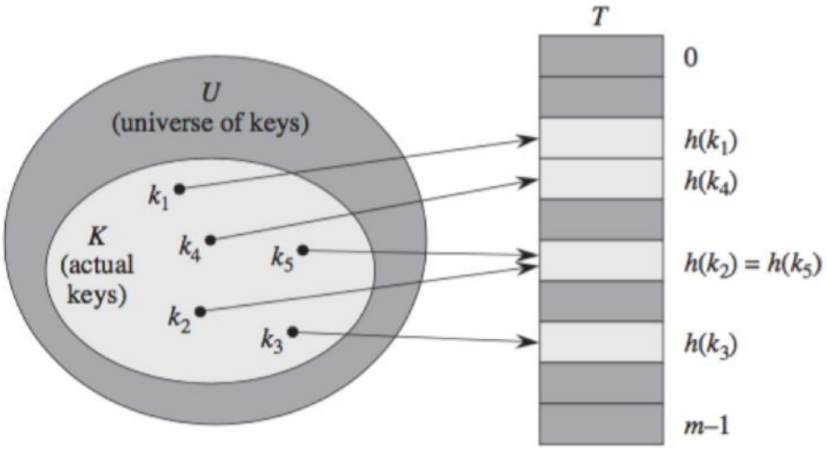
> key 的所有可能的集合太大，Direct Address Table 放不下

- e.g. ASCII string with 10 chars: 26^{10}

The set K of keys actually stored may be so small relative to U

- e.g. actually words with 10 chars

When the set K is small, a hash table requires much less storage than a direct address table Memory required: Θ(|K|)



Hash Table

Hash Function

With hashing, an element with key k is stored in slot $h(k)$
We use a hash function to compute the slot from a key k

Let M be the size of the hash table, then
$h : U \to \{0, 1, \ldots, M-1\}$

We say that an element with key k hashes to slot $h(k)$ while $h(k)$ is the hash value of key k

Hash function reduces the range of indices needed to be handled: M instead of $\mid U \mid$

> 哈希函数是一个映射，用于压缩键空间.
>
> 可能有不同的键被映射到相同的槽（哈希冲突），需要额外处理.
>
> 可能有很多键理论存在但是实际没有，使用哈希就不用为它们额外开辟空间.

The performance of the hashing depends on how well the hash function distribute the keys to the M slots

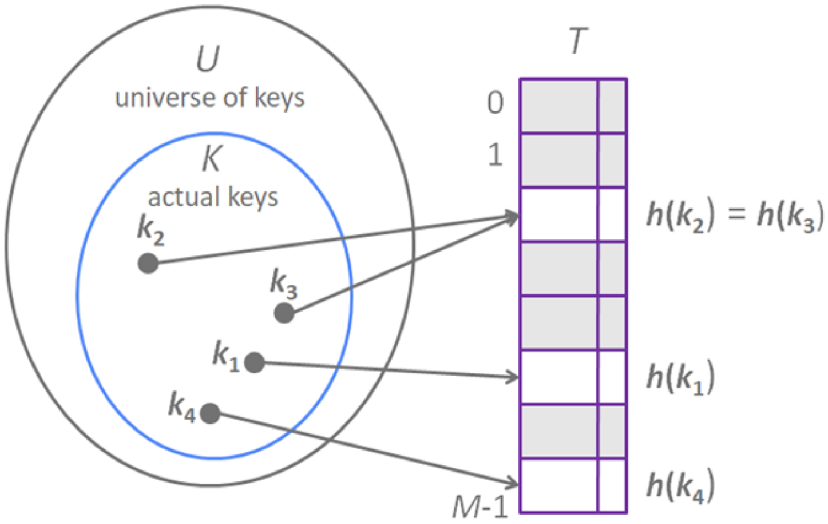
Simple uniform hashing: any given element is equally likely to be hash into any of the M slots

> 简单均匀哈希：对任意 key，哈希值 $h(k)$ 等可能分布

Collisions: two keys may hash to the same slot

- We need to find methods to handle collisions – collision solution (handling)

> 哈希冲突：两个 key 有相同的哈希值



Choosing a Good Hash Function

The hash function hhh should be:

- be easy and quick to be computed
- achieve an even distribution of the keys
- deterministic that a given input k always produce same output $h(k)$
  It is a good idea to ensure that the table size is prime. Why?

  > The modulo (%) operation used to map hash values to indices in the table will distribute the key more uniformly across all slots.
  >
  > Non-prime sizes can lead to clustering(聚集), where certain indices are more likely to be hit due to patterns in the keys and the hash function.
  >
  > 例如，如果表大小是 10，那么所有整十的数都会分配到 same slot.

  Hash Function for Integers

- If the input keys are integers, then simply returning K mod M is generally a reasonable strategy

  > 一个好用的针对整数键的哈希函数：直接对哈希表大小求模.

- When the keys are random integers, this function is simple to compute and can distribute keys evenly

Hash Function for Strings

一个不好的例子：

```
unsigned int hash(char *key, unsigned int m){
    unsigned int hash_val = 0;
    while (*key != '\0')
        hash_val += *key++;
    return (hash_val % m);
}
```

> This hash function sums up the value of each character
>
> Problem: This hash function does not hash evenly
>
> Example: ABC and CBA return the same hash key

优化版本：

```
unsigned int hash(char *key, unsigned int m){
    return (key[0] + 27 * key[1] + 729 * key[2]) % m;
}
```

> It examines the first three letters of the keys
>
> Problem: English words are not random
>
> Example: “Advantage”, “Adventure”, “Advertisement” return the same hash key

继续优化：使用霍纳法则（Horner’s rule）

> $a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = (((a_nx + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$
>
> 左边的多项式进行了大量无用计算（在算 x^n 的时候已经得到了 x^1 到 x^{n-1} 的答案，但是并没有保存下来）. 可转化为右边嵌套形式.

字符串记为 k. 对一个字符串键，计算 $\sum_{i=0}^{N-1} k[N-1-i] \times 32^i$. where N is the length of the key.

> 从右往左计算，32 是权重，越往左（靠近开头）越大.

```c
unsigned int hash(char *key, unsigned int m){
    unsigned int hash_val = 0;
    while (*key != '\0') hash_val = (hash_val << 5) + *key++;
    return (hash_val % m); // 这里注意取模
}
```

It is common not to use all characters - the length and properties of the keys would influence the choice

> 可以不用到键的所有信息

The hash function might include a couple characters from each field

截断（Truncation）：ignore part of the key and use the remaining part directly as the index

> e.g. 62,538,194 hash to 394 by taking 1st 2nd and 5th digits

折叠（Folding）：partition the key into several parts and combine parts in a convenient way

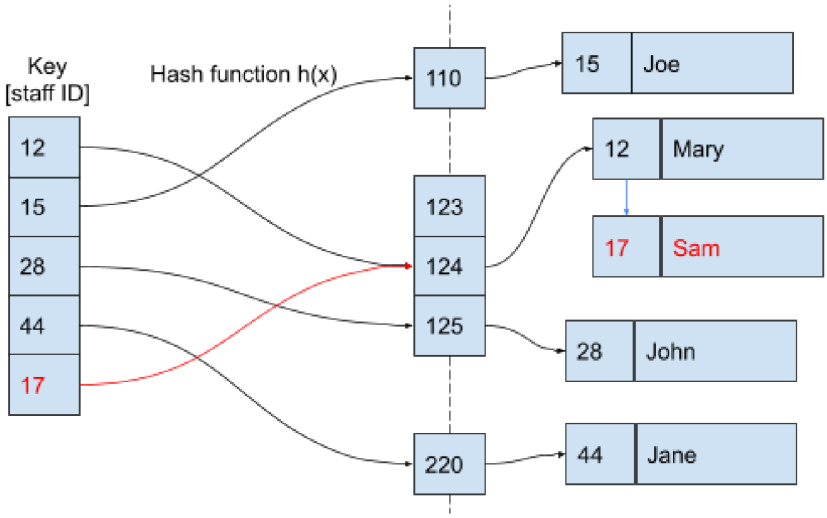
> e.g. 62,538,194 maps to 625 + 381 + 94 = 1100 and is truncated to 100

6.3 哈希冲突

> Collision Resolution

① 分离链接法

> Seperate Chaining
>
> **Separate Chaining（分离链接法）** 是一种解决哈希表中**哈希冲突（Hash Collision）**的常用方法. 当多个键被映射到相同的哈希表索引时，分离链接法通过在该索引处维护一个链表（或其他数据结构）来存储冲突的键值对.



In separate chaining, we put all elements that hash to the same slot in a linked list (or normal list)

Slot j contains a header node of the list that stores ALL elements that are hashed to j

> contains a NULL link if it is empty

代码实现

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<limits.h>

// 定义链表节点结构，用于存储键值对
typedef struct node {
    int key;                // 键
    int value;              // 值
    struct node *next;      // 指向下一个节点的指针
} List;

// 定义哈希表结构，使用分离链接法 (Separate Chaining)
typedef struct hash_sc {
    int size;               // 哈希表的大小（槽的数量）
    List ** slots;          // 槽数组，每个槽是一个链表的头指针
} Hash_SC;

// 插入或更新链表中的键值对
List* list_insert(List *L, int key, int value) {
    List *tmp;

    // 如果键已存在，则更新对应的值
    if (list_find(L, key) != INT_MAX) { // 查找键是否存在
        tmp = L;
        while (tmp->key != key)         // 遍历链表找到目标键
            tmp = tmp->next;
        tmp->value = value;             // 更新值
        return L;                       // 返回链表头
    }

    // 如果键不存在，创建新节点并插入链表头部
    tmp = malloc(sizeof(List));         // 分配内存
    tmp->key = key;                     // 设置键
    tmp->value = value;                 // 设置值
    tmp->next = L;                      // 新节点指向链表头
    L = tmp;                            // 更新链表头
    return tmp;
}

// 删除链表中的键值对
```

```c
List* list_delete(List *L, int key) {
    List *tmp, *prev;
    tmp = L;

    // 遍历链表找到目标键
    while (tmp != NULL && tmp->key != key) {
        prev = tmp;
        tmp = tmp->next;
    }

    // 如果未找到目标键，直接返回链表头
    if (tmp == NULL)
        return L;

    // 如果目标键是链表的头节点
    if (tmp == L) {
        prev = L;
        tmp = tmp->next;                // 更新链表头
        free(prev);                     // 释放原头节点
        return tmp;
    }

    // 如果目标键在链表中间或尾部
    prev->next = tmp->next;             // 跳过目标节点
    free(tmp);                          // 释放目标节点
    return L;
}

// 查找链表中的键值对，返回值；如果键不存在，返回 INT_MAX
int list_find(List *L, int key) {
    List *tmp = L;
    while (tmp != NULL && tmp->key != key) // 遍历链表找到目标键
        tmp = tmp->next;

    if (tmp == NULL)                        // 如果未找到，返回 INT_MAX
        return INT_MAX;

    return tmp->value;                      // 返回目标键对应的值
}

// 初始化哈希表
void hash_init(Hash_SC **T, int size) {
    Hash_SC *tmpT = malloc(sizeof(Hash_SC));   // 分配哈希表结构内存
    tmpT->size = size;                         // 设置哈希表大小
    tmpT->slots = malloc(sizeof(List*) * size);// 为槽数组分配内存
    for (int i = 0; i < tmpT->size; i++)       // 初始化每个槽为空
        tmpT->slots[i] = NULL;
    *T = tmpT;                                 // 更新哈希表指针
}

// 清空哈希表中的所有键值对
void hash_make_empty(Hash_SC *T) {
    for (int i = 0; i < T->size; i++)          // 遍历每个槽
        while (T->slots[i] != NULL)            // 清空槽中的链表
            T->slots[i] = list_delete(T->slots[i], T->slots[i]->key);
}

// 向哈希表插入或更新键值对
void hash_insert(Hash_SC *T, int key, int value) {
    int hash_value = hash_function(key, T->size);    // 计算哈希值
    T->slots[hash_value] = list_insert(T->slots[hash_value], key, value); // 插入链表
}

// 从哈希表中删除键值对
void hash_delete(Hash_SC *T, int key) {
    int hash_value = hash_function(key, T->size);    // 计算哈希值
    T->slots[hash_value] = list_delete(T->slots[hash_value], key); // 删除链表中的键
}
```

```c
// 检查哈希表中是否包含某个键
int hash_contain(Hash_SC *T, int key) {
    int hash_value = hash_function(key, T->size);    // 计算哈希值
    if (list_find(T->slots[hash_value], key) == INT_MAX) // 如果键不存在
        return 0;
    return 1;                                        // 键存在返回 1
}

// 在哈希表中查找键值对的值
int hash_find(Hash_SC *T, int key) {
    int hash_value = hash_function(key, T->size);    // 计算哈希值
    return list_find(T->slots[hash_value], key);     // 查找链表中的键
}

// 释放哈希表的内存
void hash_free(Hash_SC **T) {
    if (*T == NULL) return;                          // 如果表为空，直接返回
    hash_make_empty(*T);                             // 清空哈希表
    free((*T)->slots);                               // 释放槽数组内存
    free((*T));                                      // 释放哈希表结构内存
    *T = NULL;                                       // 将指针置空
}

// 哈希函数：根据键和表大小计算哈希值
int hash_function(int key, int size) {
    return ((key % size) + size) % size;             // 确保结果为非负值
}
```

Load Factor α

Given a hash table T with M slots that stores N elements, load factor $\alpha = \frac{N}{M}$

- Worst-case: $\alpha = N$
- Average-case: $\alpha = \frac{N}{M}$

② 开放寻址法

> Open Addressing Linear Probing, Quadratic Probing, Double Hashing
>
> 在哈希表中，**Open Addressing（开放寻址法）** 是一种用于解决哈希冲突的策略. 它的核心思想是：当冲突发生时，直接在哈希表中寻找下一个空闲位置存放数据，而不是将冲突的元素存储在链表或其他结构中（如**Separate Chaining**）.

In open addressing, all elements are stored in the hash table itself.

- Each slot contains either an element or NULL

The hash table can fill up so that no further insertions can be made α can never exceed 1

Advantage: pointers are avoided

We compute the sequence of slots to be examined

Insert: successively probe the hash table until we find an empty slot to save the key

Probing in fixed order $0, 1, \ldots, M-1$ is unwise since it requires $\Theta(N)$ search time

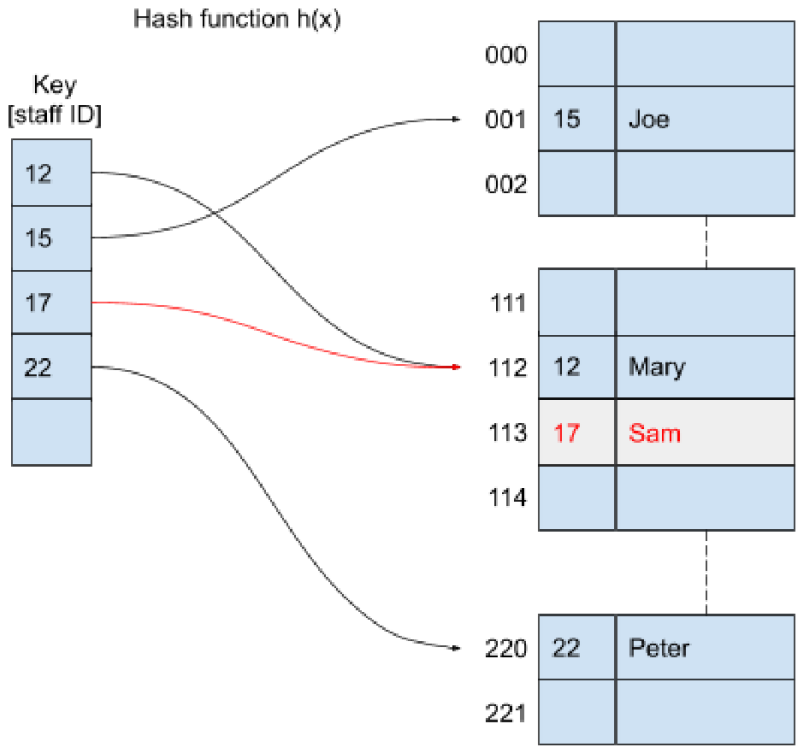
Linear Probing

> 线性探测

$$index = (h(k) + i) \ \% \ m$$

> i 是探测次数，从 0 开始循环增加，有空位就插入并结束探测.

> m 是哈希表大小，通常是质数.



优点：简单

缺点：容易出现聚集问题（Primary Clustering）：多个冲突的键会连续存放在表中，导致探测路径变长，从而影响查找效率.

Quadratic Probing

> 二次探测

$$index = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \ \% \ m$$

> 通常 $c_1 = 0,\ c_2 = 1$

优点：减少了线性探测中的主聚集问题（Primary Clustering），因为探测路径不再是连续的.

缺点：会引入次聚集问题（Secondary Clustering）：不同的键可能具有相同的探测路径.

如果表的大小 m 不是质数，可能导致无法找到空闲位置（循环不检查所有 slot）.

Double Hashing

> 二次哈希
>
> 当冲突发生时，使用另一个哈希函数 $h_2(k)$ 来计算探测步长，从而选择下一个位置.

$$index = (h_1(k) + i \cdot h_2(k)) \ \% \ m$$

> $h_1(k)$ 是初始哈希函数，用于决定起始索引.
>
> $h_2(k)$ 是第二个哈希函数，用于决定步长，通常选择一个与 m 无关的函数.
>
> 次聚集概率大大减小，因为不同 k 想要跑出相同路径，需要两个哈希值同时相等.

为了确保每个 slot 都能被访问，要求：

$h_2(k) \neq 0$ （步长不能为 0，否则原地踏步）

$h_2(k)$ 和表的大小 m 互质.

优点：避免了主聚集和次聚集问题. 探测路径更加分散，冲突处理更高效.

缺点：较复杂. 需要设计两个良好的哈希函数.

代码实现

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<limits.h>

#define HASH_NULL_KEY 2100  // 定义一个特殊值，用于表示哈希表中空槽（未使用）

// 定义哈希表的节点结构
typedef struct node {
    int key;    // 存储键
    int value;  // 存储值
} Node;

// 定义开放寻址哈希表结构
typedef struct hash_os {
    int size;               // 哈希表大小（槽数量）
    Node *slots;            // 存储节点数组
    int (*hash_function)(int, int);  // 主哈希函数指针
    int (*hash_function2)(int, int); // 二次哈希函数指针（用于双哈希）
} Hash_OA;

// 初始化哈希表
void hash_init(Hash_OA **T, int size, int (*h1)(int, int), int (*h2)(int, int)) {
    Hash_OA *tmpT = malloc(sizeof(Hash_OA));          // 为哈希表结构分配内存
    tmpT->size = size;                                // 设置哈希表大小
    tmpT->slots = malloc(sizeof(Node) * size);        // 为槽数组分配内存
    for (int i = 0; i < size; i++)                    // 初始化每个槽为空
        tmpT->slots[i].key = HASH_NULL_KEY;
    tmpT->hash_function = h1;                         // 设置主哈希函数
    tmpT->hash_function2 = h2;                        // 设置二次哈希函数
    *T = tmpT;                                        // 更新哈希表指针
}

// 清空哈希表
void hash_make_empty(Hash_OA *T) {
    for (int i = 0; i < T->size; i++)                 // 遍历所有槽
        T->slots[i].key = HASH_NULL_KEY;              // 将每个槽标记为空
}

// 插入或更新键值对
// 返回值：1 表示成功插入，2 表示更新了已有键，0 表示哈希表已满
int hash_insert(Hash_OA *T, int key, int value) {
    int hash_1 = T->hash_function(key, T->size);      // 使用主哈希函数计算初始索引
    int hash_2 = T->hash_function2(key, T->size);     // 使用二次哈希函数计算步长
    int i = 0;                                        // 探测次数

    // 使用双哈希法处理冲突
    while (T->slots[(hash_1 + i * hash_2) % T->size].key != HASH_NULL_KEY) {
        // 如果找到相同的键，更新其值
        if (T->slots[(hash_1 + i * hash_2) % T->size].key == key) {
            T->slots[(hash_1 + i * hash_2) % T->size].value = value;
            return 2; // 键已存在，更新值
        }
        i++; // 增加探测次数

        if (i == T->size) // 如果探测次数达到哈希表大小，表示哈希表已满
            return 0;
    }

    // 如果找到空槽，插入键值对
    T->slots[(hash_1 + i * hash_2) % T->size].key = key;
    T->slots[(hash_1 + i * hash_2) % T->size].value = value;
    return 1; // 成功插入
}

// 检查哈希表是否包含某个键
```

```c
// 返回值：1 表示键存在，0 表示键不存在
int hash_contain(Hash_OA *T, int key) {
    int hash_1 = T->hash_function(key, T->size);      // 使用主哈希函数计算初始索引
    int hash_2 = T->hash_function2(key, T->size);     // 使用二次哈希函数计算步长
    int i = 0;                                        // 探测次数

    // 使用双哈希法查找键
    while (T->slots[(hash_1 + i * hash_2) % T->size].key != key) {
        i++; // 增加探测次数
        if (i == T->size) // 如果探测次数达到哈希表大小，表示键不存在
            return 0;
    }

    return 1; // 键存在
}

// 查找键对应的值
// 返回值：键对应的值；如果键不存在，返回 INT_MAX
int hash_find(Hash_OA *T, int key) {
    int hash_1 = T->hash_function(key, T->size);      // 使用主哈希函数计算初始索引
    int hash_2 = T->hash_function2(key, T->size);     // 使用二次哈希函数计算步长
    int i = 0;                                        // 探测次数

    // 使用双哈希法查找键
    while (T->slots[(hash_1 + i * hash_2) % T->size].key != key) {
        // 如果遇到空槽，表示键不存在
        if (T->slots[(hash_1 + i * hash_2) % T->size].key == HASH_NULL_KEY)
            return INT_MAX;
        i++; // 增加探测次数
        if (i == T->size) // 如果探测次数达到哈希表大小，表示键不存在
            return INT_MAX;
    }

    // 返回找到的值
    return T->slots[(hash_1 + i * hash_2) % T->size].value;
}

// 释放哈希表内存
void hash_free(Hash_OA **T) {
    if (*T == NULL) return;                           // 如果哈希表为空，直接返回
    hash_make_empty(*T);                              // 清空哈希表内容
    free((*T)->slots);                                // 释放槽数组内存
    free((*T));                                       // 释放哈希表结构内存
    *T = NULL;                                        // 将指针置空
}

// 打印哈希表的内容
// 返回一个字符串，表示哈希表的所有槽及其内容
char *hash_print(Hash_OA *T) {
    int i, len;
    char *output, buffer[50];
    len = 36 * T->size + 2;                           // 估算输出字符串的长度
    output = malloc(sizeof(char) * len);              // 为输出字符串分配内存
    memset(output, 0, sizeof(char) * len);            // 初始化输出字符串
    for (i = 0; i < T->size; i++) {                   // 遍历所有槽
        if (T->slots[i].key == HASH_NULL_KEY)         // 如果槽为空
            sprintf(buffer, "%d:\n", i);              // 只打印槽的索引
        else                                          // 如果槽不为空
            sprintf(buffer, "%d:(%d)%d\n", i, T->slots[i].key, T->slots[i].value); // 打印槽的内容
        strcat(output, buffer);                       // 将结果追加到输出字符串
    }
    return output;                                    // 返回输出字符串
}
```

复杂度分析

> self study

6.4 再哈希

> Rehashing

Rehashing 是哈希表的一种动态扩展机制，主要用于解决哈希表中碰撞冲突增加或负载因子过高的问题. 当哈希表的负载因子 ($\alpha = \frac{N}{M}$) 超过某个预设值时，哈希表需要扩容并重新分配元素到新的哈希表中，这个过程称为 Rehashing.

步骤：

1. 扩容：创建一个比原来更大的哈希表（通常是原大小的 2 倍）.
2. 重新映射：将旧哈希表中所有的元素重新计算哈希值并插入到新的哈希表中.
3. 更新引用：将哈希表的引用指向新的哈希表.

代码实现

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INITIAL_CAPACITY 8 // 初始哈希表大小
#define LOAD_FACTOR 0.75   // 负载因子

// 哈希表节点结构
typedef struct Node {
    char *key;
    int value;
    struct Node *next;
} Node;

// 哈希表结构
typedef struct HashTable {
    Node **buckets;
    int capacity;
    int size;
} HashTable;

// 创建节点
Node *create_node(const char *key, int value) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->key = strdup(key);
    node->value = value;
    node->next = NULL;
    return node;
}

// 创建哈希表
HashTable *create_table(int capacity) {
    HashTable *table = (HashTable *)malloc(sizeof(HashTable));
    table->buckets = (Node **)calloc(capacity, sizeof(Node *));
    table->capacity = capacity;
    table->size = 0;
    return table;
}

// 哈希函数
unsigned int hash_function(const char *key, int capacity) {
    unsigned int hash = 0;
    while (*key) {
        hash = (hash * 31) + *key++;
    }
    return hash % capacity;
```

```
}

// 插入键值对
void insert(HashTable *table, const char *key, int value);

// Rehashing 函数
void rehash(HashTable *table) {
    int old_capacity = table->capacity;
    Node **old_buckets = table->buckets;

    // 扩容到原来的两倍
    int new_capacity = old_capacity * 2;
    table->buckets = (Node **)calloc(new_capacity, sizeof(Node *));
    table->capacity = new_capacity;
    table->size = 0;
    // 注意，这里 table->buckets 指向了另外的内存地址，不代表原来的内存就自动清空了. 正常来说需要马上free防止内存泄漏，
但是这部分信息还有用，所以先存到 old_buckets 里，等新的表创建完，把内容复制过去之后再free

    // 重新插入所有旧元素
    for (int i = 0; i < old_capacity; i++) {
        Node *node = old_buckets[i];
        while (node) {
            insert(table, node->key, node->value);
            Node *temp = node;
            node = node->next;
            free(temp->key);
            free(temp);
        }
    }

    free(old_buckets);
}

// 插入键值对
void insert(HashTable *table, const char *key, int value) {
    // 检查是否需要 Rehashing
    if ((float)(table->size + 1) / table->capacity > LOAD_FACTOR) {
        rehash(table);
    }

    // 计算哈希值
    unsigned int index = hash_function(key, table->capacity);
    Node *node = table->buckets[index];

    // 检查是否已存在键
    while (node) {
        if (strcmp(node->key, key) == 0) {
            node->value = value; // 更新值
            return;
        }
        node = node->next;
    }

    // 插入新节点
    Node *new_node = create_node(key, value);
    new_node->next = table->buckets[index];
    table->buckets[index] = new_node;
    table->size++;
}

// 查找键值对
int search(HashTable *table, const char *key, int *value) {
    unsigned int index = hash_function(key, table->capacity);
    Node *node = table->buckets[index];

    while (node) {
        if (strcmp(node->key, key) == 0) {
            *value = node->value;
```

```c
            return 1; // 找到
        }
        node = node->next;
    }

    return 0; // 未找到
}

// 打印哈希表
void print_table(HashTable *table) {
    for (int i = 0; i < table->capacity; i++) {
        printf("Bucket %d: ", i);
        Node *node = table->buckets[i];
        while (node) {
            printf("(%s, %d) -> ", node->key, node->value);
            node = node->next;
        }
        printf("NULL\n");
    }
}

// 释放哈希表
void free_table(HashTable *table) {
    for (int i = 0; i < table->capacity; i++) {
        Node *node = table->buckets[i];
        while (node) {
            Node *temp = node;
            node = node->next;
            free(temp->key);
            free(temp);
        }
    }
    free(table->buckets);
    free(table);
}

// 主函数测试
int main() {
    HashTable *table = create_table(INITIAL_CAPACITY);

    insert(table, "key1", 10);
    insert(table, "key2", 20);
    insert(table, "key3", 30);
    insert(table, "key4", 40);
    insert(table, "key5", 50);
    insert(table, "key6", 60);

    printf("Before Rehashing:\n");
    print_table(table);

    // 插入更多元素触发 Rehashing
    insert(table, "key7", 70);
    insert(table, "key8", 80);

    printf("\nAfter Rehashing:\n");
    print_table(table);

    // 查找元素
    int value;
    if (search(table, "key3", &value)) {
        printf("\nFound key3 with value: %d\n", value);
    } else {
        printf("\nKey3 not found\n");
    }

    // 释放哈希表
    free_table(table);
```

```
    return 0;
}
```

> `malloc` 只分配内存，不初始化；`calloc` 分配内存后会将每个字节初始化为零.

注意，创建新表后，哈希函数严格来说也变化了，因为对表的大小求余数，而表的大小变了. 所以要用 insert 方法一一重新计算哈希并插入，而不能直接 copy 索引对应内容.

以下是 self-study

If you are interested in this topic, you can continue to search for the terms

1. Random probing
2. Extendible hashing
3. Application of Hash Table: Cryptography (e.g. checksum, password encoding)

7. 树

> 树

The tree structure means a branching relations between nodes.

A finite set T of one or more nodes such that

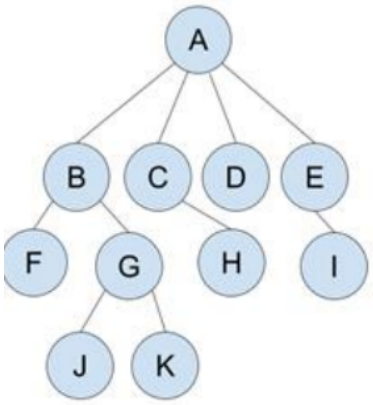
- There is one specially designed node called the root of the tree
- The remaining nodes are partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$ and each of these is a tree (T_i are called subtrees)

7.1 术语

> Tree Terminology

Terminology	Meaning	Example
Root 根节点	The distinguished node of the tree	A
Subtree 子树	The tree partitioned by any internal nodes	{C, H}
Degree 度	The number of subtrees of a node	deg(A) = 4
Leaf (Terminal node) 叶子节点（终端节点）	Nodes of degree 0	D, F, H, I, J, K
Internal node (Non-terminal node) 内部节点（非终端节点）	Nodes of non-zero degree	A, B, C, E, G
Height 高度	The height of leaves is zero. The height of a parent is greater than that of its children by 1	height(C) = 1
Depth (Level) 深度（层级）	The root has depth 0. The depth of a child is greater than that of its parent by 1	depth(C) = 1
Parent 父节点	直接在其上方的内部节点	parent(F) = B
Siblings 兄弟节点	具有相同父节点的节点	sibling(B) = {C, D, E}
Children 子节点	直接在其下方的节点	child(A) = {B, C, D, E}

Terminology	Meaning	Example
Grandparent 祖父节点	深度少两层的内部节点	grandparent(F) = A
Grandchild 孙子节点	深度多两层的节点	grandchild(A) = {F, G, H, I}
Path 路径	A sequence of node in which n_k is the ancestor of n_{k+1}	path(B, K) = {B, G, K}
Ancestor 祖先	从 (n_1) 到 (n_2) 存在路径，(n_1) 是 (n_2) 的祖先	ancestor(G) = {A, B}
Descendant 后代	从 (n_1) 到 (n_2) 存在路径，(n_2) 是 (n_1) 的后代	descendant(G) = {J, K}



7.2 分类

> Types of Tree

① 二叉树

> Binary tree: each node has at most 2 children，每个节点最多有 2 个子节点

Full Binary Tree

> 满二叉树：除叶子节点外，每个节点恰有 2 个节点

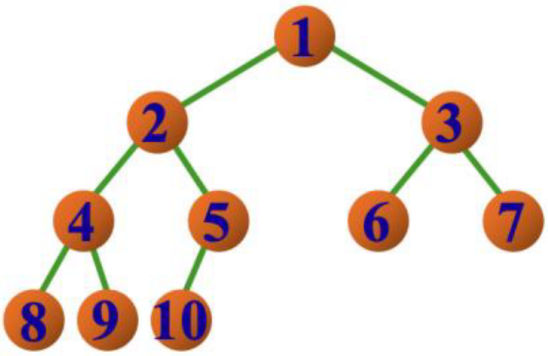
Complete Binary Tree

> 完全二叉树：除了最后一层，每一层都填满；最后一层节点从左到右排列.
>
> 可以看成最后一层不一定满的满二叉树.

Property of Complete Binary Tree

性质：n 个节点的完全二叉树，从上往下，从左往右给节点编号.

- If $i = 1$, 节点是 root, no parent, otherwise, $Parent(i) = \lfloor \frac{i}{2} \rfloor$
- If $2i > n$, 节点没有 left child, otherwise, $Lchild(i) = 2i$
- If $2i + 1 > n$, 节点没有 right child; otherwise, $Rchild(i) = 2i + 1$



Full Binary Tree

> 满二叉树

- Def 1: A binary tree of depth k, with $2^k - 1$ nodes (??).

> 总节点数：$\sum_{i=0}^{k} 2^i = \frac{1-2^{k+1}}{1-2} = 2^{k+1} - 1$
>
> 叶子节点数：2^k
>
> 内部节点数：$2^k - 1$

- Def 2: Any parent node in the binary tree has 0 or 2 children.

> 要么是叶子节点，要么有两个子节点.
>
> In other words, 满二叉树中每个内部节点都有两个子节点，且没有度为1的节点.

Complete Binary Tree

> 完全二叉树

- A binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible

> 所有层都被完全填满，只有最后一层的节点不必完全填满，但它们必须从左向右连续排列（从左往右填）

② 搜索树

> Search tree: each node has a (unique) key, descendants on the left has smaller key, while descendants on the right has larger key，每个节点有一个（唯一的）键，左边的后代有较小的键，右边的后代有较大的键

③ 平衡树

> Balanced tree: the height of the tree maintains automatically after each insertion or deletion. The height of all subtrees of a node can at most differ by 1，树的高度在每次插入或删除后自动维持。一个节点的所有子树的高度最多相差 1

④ 根树

> Rooted tree: a tree contains a root，树包含一个根节点

⑤ 有序树

> Ordered tree: the relative order of the subtrees is important (i.e. left subtree and right subtree has different meaning)，子树的相对顺序很重要（左子树和右子树有不同的意义）

⑥ 无序树

> Oriented tree: the opposite of ordered tree：与有序树相反

7.3 数据结构

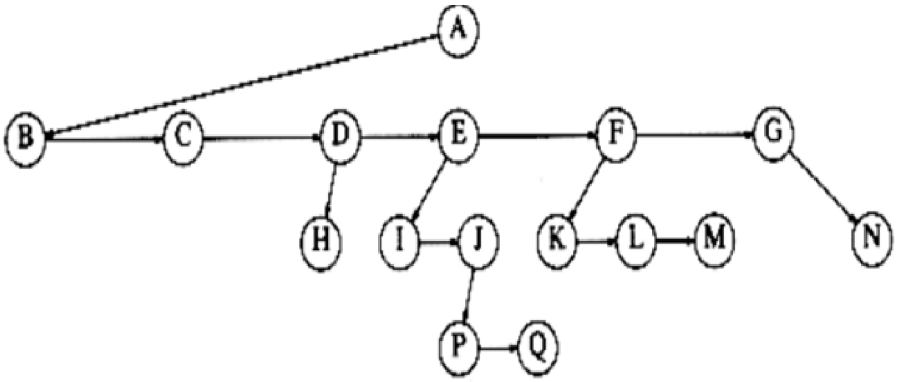
A tree is a collection of nodes with parent-child relations in a hierarchical manner without any cycles

Operations

- Search(nodeN): To search for nodeN in the tree
- Add(nodeP, nodeN): To add nodeN as children to the nodeP, if possible
- Remove(nodeN): To remove nodeN from the tree
- GetParent(nodeN), GetChildren(nodeN), GetSibling(nodeN): To get the parent/children/siblings of the nodeN
- Traverse(): To go through all elements of the whole tree

Left-child Right-sibling implementation

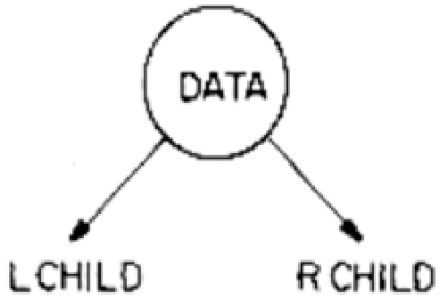
- Each node contains (key, value) pair, pointer for first child and pointer for next sibling




```
typedef struct node {
int data;
    struct node *next;
    struct node *child;
} Node;

// init a Node
Node* init_node(int data) {
    Node *tmp = malloc(sizeof(Node));
    tmp->next = NULL;
    tmp->child = NULL;
    tmp->data = data;
    return tmp;
}

Node *addSibling(Node *n, int data){
    if (n == NULL)
        return NULL;

    while (n->next != NULL)
        n = n->next;

    n->next = init_node(data)
    return n->next;
}

Node *addChild(Node * n, int data){
    Node *tmp;
    if (n == NULL) return NULL;
    // Check if child list is not empty
        if (n->child != NULL){
            tmp=addSibling(n->child, data);
            return tmp;
        }else{
            n->child = init_node(data)
            return n->child;
    }
}
```

Left-child, right-child implementation

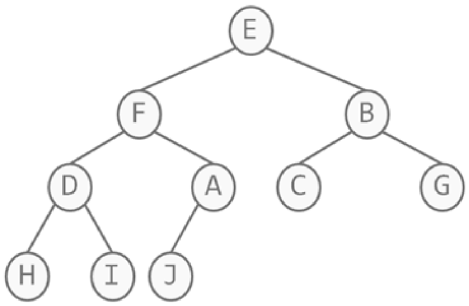
- Each node contains (key, value) pair, pointer for left child and right child
- For general tree, it is modified as list of pointers for all its children



> 二叉搜索树中讨论.

Array implementation (binary tree only)

- Root is stored at 0-th position
- Children of i-th node are $2i+1$-th node and $2i+2$-th node



	E	F	B	D	A	C	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

> Heap 中讨论.

7.4 树遍历

The algorithms for walking through a rooted and ordered tree

These methods to examine the nodes of the tree systemically so that each node is visited exactly once

① 前序遍历

Preorder Traversal

- Visit the root
- Traverse the left subtree
- Traverse the right subtree

② 中序遍历

Inorder Traversal

- Traverse the left subtree
- Visit the root
- Traverse the right subtree

(Only applicable to binary tree)

> 结果会按照升序排序.

③ 后序遍历

Postorder Traversal

- Traverse the left subtree
- Traverse the right subtree
- Visit the root

Preorder

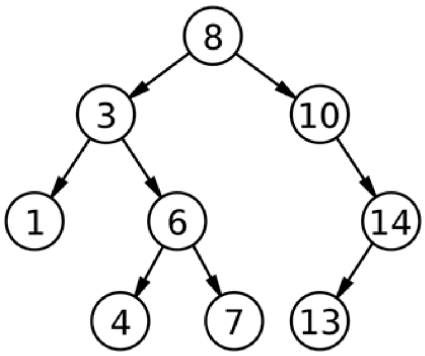
- 8, 3, 1, 6, 4, 7, 10, 14, 13

Postorder

- 1, 4, 7, 6, 3, 13, 14, 10, 8

Inorder

- 1, 3, 4, 6, 7, 8, 10, 13, 14



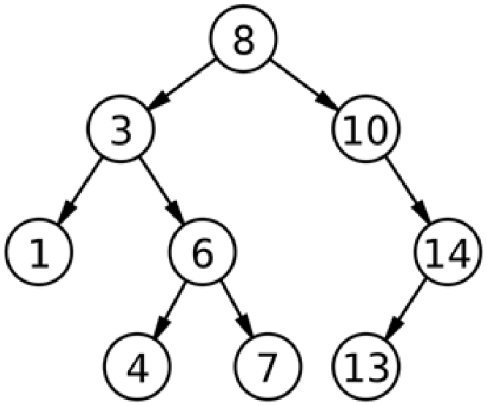
Inorder Traversal with Stack

> 待补充.

7.5 几种特殊树

① *二叉搜索树

> Binary Search Tree
>
> 重点考察



Each node in the binary tree is assigned a key

For every node with key k, the values of all the keys in its left subtree are smaller than k, and the values of all the keys in its right subtree are larger than k

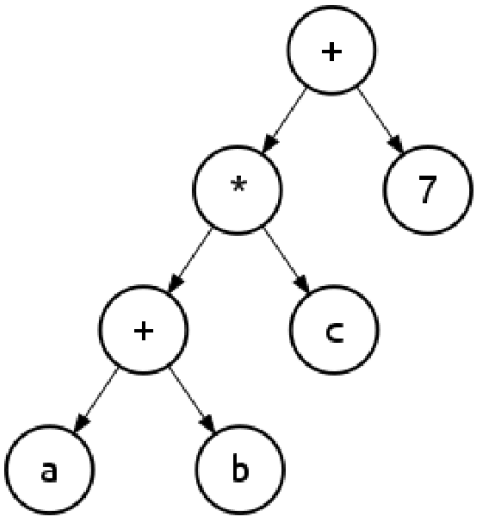
Binary search tree is also a rooted and ordered tree

Application: tree sort

```c
#include <stdio.h>
#include <stdlib.h>

// 定义节点结构
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// 创建一个新节点
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// 插入节点
Node* insertNode(Node* root, int data) {
    if (root == NULL) {
        return createNode(data); // 如果树为空，创建新节点
    }

    if (data < root->data) {
        root->left = insertNode(root->left, data); // 插入左子树
    } else if (data > root->data) {
        root->right = insertNode(root->right, data); // 插入右子树
    }

    return root; // 返回根节点
}

// 查找节点
Node* searchNode(Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root; // 找到节点或到达叶子节点
    }

    if (data < root->data) {
        return searchNode(root->left, data); // 在左子树中查找
    } else {
        return searchNode(root->right, data); // 在右子树中查找
    }
}

// 中序遍历
void inorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }

    inorderTraversal(root->left); // 遍历左子树
    printf("%d ", root->data);    // 输出当前节点
    inorderTraversal(root->right); // 遍历右子树
}
```

② 表达式树

> Expression Tree
>
> 掌握



It represents an arithmetic expression or a boolean expression

All internal nodes are operations, while all leaves are values

Expression tree for arithmetic expression is an ordered and rooted tree, while that for boolean is an oriented and rooted tree

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

// 定义树节点
typedef struct Node {
    char data;                // 节点内容：操作符或操作数
    struct Node *left;        // 左子树
    struct Node *right;       // 右子树
} Node;

// 创建新节点
Node* createNode(char data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// 检查是否为操作符
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

// 构建表达式树（从后缀表达式）
Node* constructTree(char postfix[]) {
    Node* stack[100];  // 用数组模拟栈
    int top = -1;

    for (int i = 0; postfix[i] != '\0'; i++) {
        char c = postfix[i];

        if (isOperator(c)) {
            // 如果是操作符，弹出两个节点作为左右子树
            Node* right = stack[top--];
            Node* left = stack[top--];
            Node* newNode = createNode(c);
```

```
            newNode->left = left;
            newNode->right = right;
            stack[++top] = newNode;  // 将新节点压入栈
        } else if (isalnum(c)) {
            // 如果是操作数，创建新节点并压入栈
            stack[++top] = createNode(c);
        }
    }

    // 栈顶元素即为表达式树的根节点
    return stack[top];
}

// 中序遍历（生成中缀表达式）
void inorderTraversal(Node* root) {
    if (root != NULL) {
        // 如果是操作符，添加括号
        if (isOperator(root->data)) printf("(");
        inorderTraversal(root->left);
        printf("%c", root->data);
        inorderTraversal(root->right);
        if (isOperator(root->data)) printf(")");
    }
}

// 后序遍历（生成后缀表达式）
void postorderTraversal(Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%c", root->data);
    }
}

// 求值（递归计算表达式树的值）
int evaluate(Node* root) {
    if (root == NULL) return 0;

    // 如果是叶节点（操作数），返回其值
    if (!isOperator(root->data)) {
        return root->data - '0';  // 假设操作数是单个数字
    }

    // 递归计算左右子树
    int leftVal = evaluate(root->left);
    int rightVal = evaluate(root->right);

    // 根据操作符计算结果
    switch (root->data) {
        case '+': return leftVal + rightVal;
        case '-': return leftVal - rightVal;
        case '*': return leftVal * rightVal;
        case '/': return leftVal / rightVal;
    }

    return 0;
}
```

③ *B 树

> B-Tree
>
> 重点考察
>
> 见 Assignment 3

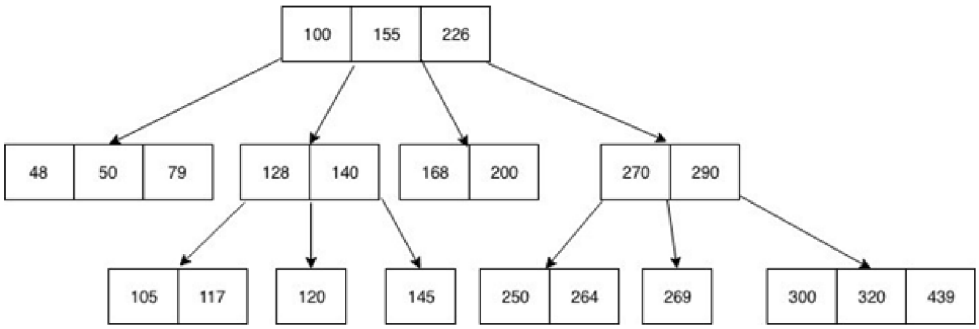
B-Tree is a generalization of binary search tree to multiple children

> 二叉搜索树扩展到多 children

B-Tree is a self-balancing rooted search tree

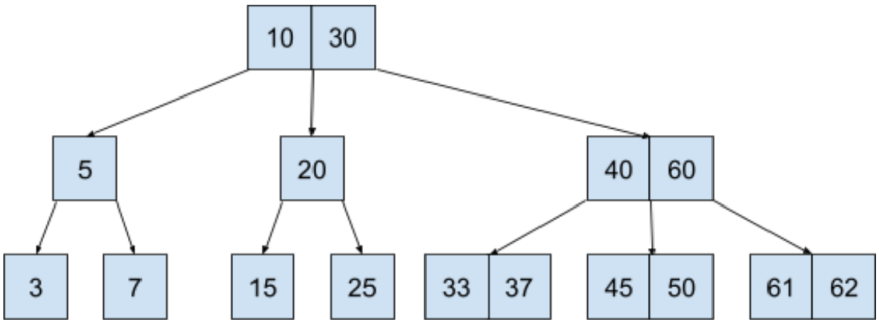
> 自平衡多叉搜索根树.

Application: Databases, file systems



例题:

2. This question is about the B-tree, with m = 3. (20%)



a. Show the results of inserting 9 into the original B-tree

b. Show the results of inserting 72 into the original B-tree.

c. Show the results of deleting 20 from the original B-tree.

d. Show the results of deleting 40 from the original B-tree.

Note: The four parts are independent. You should perform the action based on the original B-tree.

Because there are multiple possible implementations for insertion and deletion in B-trees (such as choosing the largest predecessor or the smallest successor, choosing the left sibling or the right sibling), the combination of these choices results in even more possible answers. Therefore, I will first limit the implementation in my answer to ensure uniqueness (see the parts marked in red).

According to the lecture and online information, the main logic of B-tree is as follows:

1. For an m-order B-tree, **every node has at most m children** (and m-1 keys). **A non-leaf node with k children contains k-1 keys**. **The root has at least 2 children and 1 key (if it is not a leaf node)**. **Non-root-and-non-leaf nodes must have at least ⌈m/2⌉ children and ⌈m/2⌉ - 1 keys**. **Leaf nodes must have at least ⌈m/2⌉ - 1 elements but without children**. All leaves appear in the same level.

2. For the **insert** operation, first, follow the rules to **reach a leaf node** and insert the key. **If there is no overflow in the node being inserted, the process ends normally**. **If overflow occurs** (more than m-1 keys after insertion), **the middle key (⌈m/2⌉) moves up to the parent node, and the node (being inserted) splits into two**. Note that the parent node might also overflow, in which case the process of "moving the middle key

up and splitting" repeats (of course, the subtrees also split evenly). If overflow continues up to the root node, the middle key (of the root) moves up to become the new root (increasing the tree height), and the original root node splits.

3. For the **delete** operation, **when deleting a key from a non-leaf node, convert it to deleting a key from a leaf node**: find its largest predecessor/smallest successor, **replace the key** (to be deleted) **with the largest predecessor/smallest successor**, **and then delete the largest predecessor/smallest successor from its original position**. Here in my solution, I just use the smallest successor for implementation (Otherwise, there would be too many possible answers, but not much significance).

For **deletion of a key from a leaf node**, **if there is no underflow after deletion, the process ends normally**. **If underflow occurs** (fewer than ⌈m/2⌉ - 1 keys after deletion), consider two scenarios:

The first scenario is to **check if there are any keys that can be borrowed from the left or right sibling nodes of the underflowing node**. Determine if borrowing is possible by **checking if the sibling node will not underflow after being borrowed a key from**. If a borrowable sibling exists (For my answer, prioritize the left sibling; if the left cannot lend, check the right sibling; if neither can lend, proceed to the second scenario), **move the key from the parent node, which is between the current (underflowing) node and the sibling node, into the underflowing node**. Then, move the borrowed key from the sibling (the largest key from the left sibling or the smallest from the right sibling) into the position of the moved key in the parent node.

The second scenario occurs **when neither sibling can lend** (borrowing would cause them to underflow), thus, **consider merging**. You can merge with either the left or right sibling (For my answer, prioritize the left sibling; if no left sibling, merge with the right sibling). If merging with the left sibling, find the key in the parent node between the current (underflowing) node and the merging sibling node (left sibling), move it into the left-sided node (left sibling), and then merge. If merging with the right sibling, find the key in the parent node between the current (underflowing) node and the merging sibling node (right sibling), move it into the left-sided node (the current underflowing node), and then merge.

If the parent node underflows due to a reduction in keys, then reconsider the two scenarios for this parent node. Note that if a key is borrowed from a sibling node, the subtree corresponding to that key must also be moved to the corresponding position of the current underflowing node to maintain the properties of the B-tree.

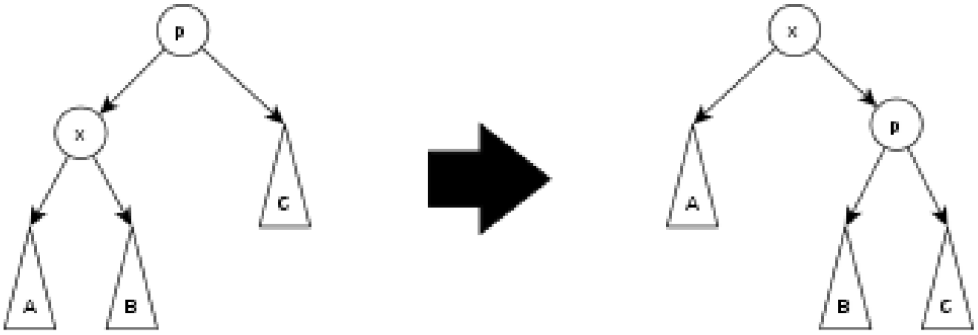
```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// B树的最小度数（每个节点最少有t-1个关键字，最多有2t-1个关键字）
#define T 3

// B树节点结构
typedef struct BTreeNode {
    int keys[2 * T - 1];               // 关键字数组
    struct BTreeNode* children[2 * T]; // 子节点指针数组
    int n;                             // 当前关键字数量
    bool leaf;                         // 是否是叶子节点
} BTreeNode;

// 创建一个新的B树节点
BTreeNode* createNode(bool leaf) {
    BTreeNode* node = (BTreeNode*)malloc(sizeof(BTreeNode));
    node->leaf = leaf;
    node->n = 0;
    for(int i = 0; i < 2 * T; i++) {
        node->children[i] = NULL;
    }
    return node;
}

// 在节点x的第i个子节点前插入新的子节点y
void insertNonFull(BTreeNode* x, int key);
void splitChild(BTreeNode* x, int i, BTreeNode* y);

// B树插入
void insertBTree(BTreeNode** root, int key) {
    BTreeNode* r = *root;
    if (r->n == 2 * T -1) {
        // 根节点满，创建新根
        BTreeNode* s = createNode(false);
        *root = s;
        s->children[0] = r;
        splitChild(s, 0, r);
        insertNonFull(s, key);
    }
    else {
        insertNonFull(r, key);
    }
}

// 分裂子节点y，i是x的子节点y的索引
void splitChild(BTreeNode* x, int i, BTreeNode* y) {
    BTreeNode* z = createNode(y->leaf);
    z->n = T -1;

    // 将y的后T-1个关键字复制到z
    for(int j = 0; j < T-1; j++) {
        z->keys[j] = y->keys[j + T];
    }

    // 如果y不是叶子节点，则复制y的后T个子节点到z
    if (!y->leaf) {
        for(int j = 0; j < T; j++) {
            z->children[j] = y->children[j + T];
        }
    }

    y->n = T -1;
```

```
    // 在x的子节点中插入z
    for(int j = x->n; j >= i+1; j--) {
        x->children[j+1] = x->children[j];
    }
    x->children[i+1] = z;

    // 将y的中间关键字上升到x
    for(int j = x->n -1; j >= i; j--) {
        x->keys[j+1] = x->keys[j];
    }
    x->keys[i] = y->keys[T-1];
    x->n += 1;
}

// 在非满节点x中插入key
void insertNonFull(BTreeNode* x, int key) {
    int i = x->n -1;

    if (x->leaf) {
        // 找到插入位置
        while (i >=0 && key < x->keys[i]) {
            x->keys[i+1] = x->keys[i];
            i--;
        }
        x->keys[i+1] = key;
        x->n +=1;
    }
    else {
        // 找到子节点
        while (i >=0 && key < x->keys[i]) {
            i--;
        }
        i +=1;
        if (x->children[i]->n == 2 * T -1) {
            splitChild(x, i, x->children[i]);
            if (key > x->keys[i]) {
                i++;
            }
        }
        insertNonFull(x->children[i], key);
    }
}

// B树搜索
BTreeNode* searchBTree(BTreeNode* x, int key) {
    int i =0;
    while (i < x->n && key > x->keys[i]) {
        i++;
    }
    if (i < x->n && key == x->keys[i]) {
        return x;
    }
    if (x->leaf) {
        return NULL;
    }
    return searchBTree(x->children[i], key);
}

// 中序遍历B树
void traverseBTree(BTreeNode* x) {
    int i;
    for(i =0; i < x->n; i++) {
        if (!x->leaf) {
            traverseBTree(x->children[i]);
        }
        printf("%d ", x->keys[i]);
    }
```

```
    if (!x->leaf) {
        traverseBTree(x->children[i]);
    }
}
```

④ 伸展树

> Splay Tree
>
> self study



A binary tree in which we move more used nodes towards the root, a process called "splaying"

> 每次访问一个节点后，会通过一系列旋转操作将该节点“伸展”到树的根部。这样可以保证最近访问的节点更快被访问，从而提高了访问具有局部性的数据的性能.

Splay tree is a self-balancing binary search tree

> 自平衡二叉搜索树

Application: Caches, Garbage Collection Algorithms

```
#include <stdio.h>
#include <stdlib.h>

// 定义伸展树的节点
typedef struct SplayNode {
    int key;
    struct SplayNode *left, *right;
} SplayNode;

// 创建一个新的节点
SplayNode* createNode(int key) {
    SplayNode* node = (SplayNode*)malloc(sizeof(SplayNode));
    node->key = key;
    node->left = node->right = NULL;
    return node;
}

// 右旋操作
SplayNode* rightRotate(SplayNode* x) {
    SplayNode* y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// 左旋操作
SplayNode* leftRotate(SplayNode* x) {
    SplayNode* y = x->right;
```

```
    x->right = y->left;
    y->left = x;
    return y;
}

// 伸展操作，将 key 对应的节点旋转到根
SplayNode* splay(SplayNode* root, int key) {
    if (root == NULL || root->key == key)
        return root;

    // key 在左子树中
    if (key < root->key) {
        if (root->left == NULL)
            return root;

        // Zig-Zig (左左)
        if (key < root->left->key) {
            root->left->left = splay(root->left->left, key);
            root = rightRotate(root);
        }
        // Zig-Zag (左右)
        else if (key > root->left->key) {
            root->left->right = splay(root->left->right, key);
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
        return (root->left == NULL) ? root : rightRotate(root);
    }
    // key 在右子树中
    else {
        if (root->right == NULL)
            return root;

        // Zag-Zig (右左)
        if (key < root->right->key) {
            root->right->left = splay(root->right->left, key);
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        // Zag-Zag (右右)
        else if (key > root->right->key) {
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }
        return (root->right == NULL) ? root : leftRotate(root);
    }
}

// 插入操作
SplayNode* insert(SplayNode* root, int key) {
    if (root == NULL)
        return createNode(key);

    root = splay(root, key);

    if (root->key == key)
        return root;

    SplayNode* newNode = createNode(key);

    if (key < root->key) {
        newNode->right = root;
        newNode->left = root->left;
        root->left = NULL;
    } else {
        newNode->left = root;
        newNode->right = root->right;
        root->right = NULL;
```

```
    }

    return newNode;
}

// 删除操作
SplayNode* deleteNode(SplayNode* root, int key) {
    if (root == NULL)
        return NULL;

    root = splay(root, key);

    if (root->key != key)
        return root;

    SplayNode* temp;
    if (root->left == NULL) {
        temp = root->right;
    } else {
        temp = splay(root->left, key);
        temp->right = root->right;
    }

    free(root);
    return temp;
}

// 查找操作
SplayNode* search(SplayNode* root, int key) {
    return splay(root, key);
}

// 中序遍历打印树
void inorder(SplayNode* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// 释放树的内存
void freeTree(SplayNode* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
```

⑤ 红黑树

> Red Black Tree
>
> self study

Every node is colored either red or black

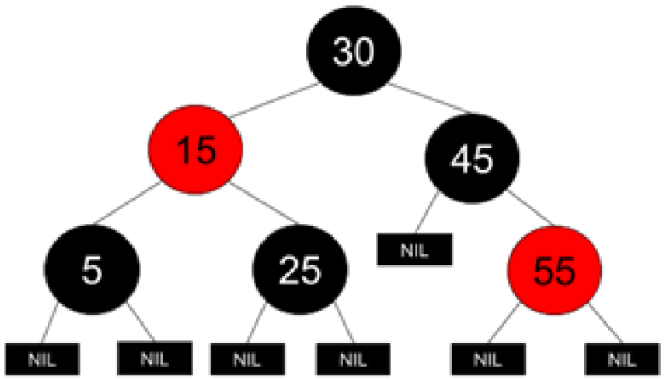
The root and all leaves are black

If the node is red, its children must be black

Every path from a node to a NULL pointer contains the same number of black nodes

RB-Tree is a self-balancing binary search tree

> 自平衡二叉搜索树
>
> 树高控制不如 AVL 树严格，因此查询效率略逊于 AVL 树. 但是插入删除更高效.



性质：

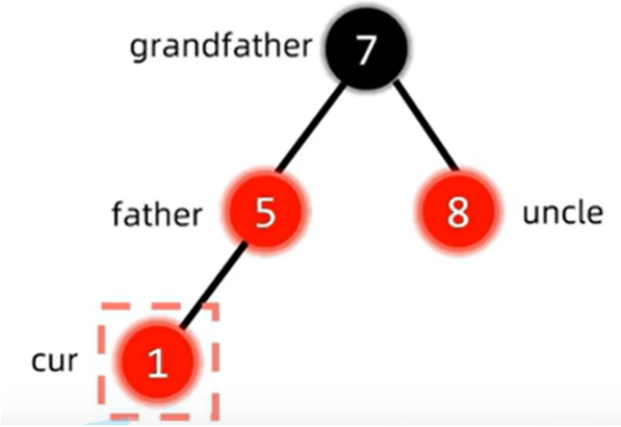
1. 每个节点是红色或黑色.
2. 根节点是黑色.
3. 每个叶子节点（NIL）是黑色.
4. 如果一个节点是红色，那么它的两个子节点都是黑色（从上到下，不能连续两红）.
5. 从任意节点到其每个叶子节点的所有路径上，黑色节点的数量相同.
6. 最长路径（黑红相间）不大于等于最短路径（连续的黑）的 2 倍.

左根右，根叶黑，不红红，黑路同

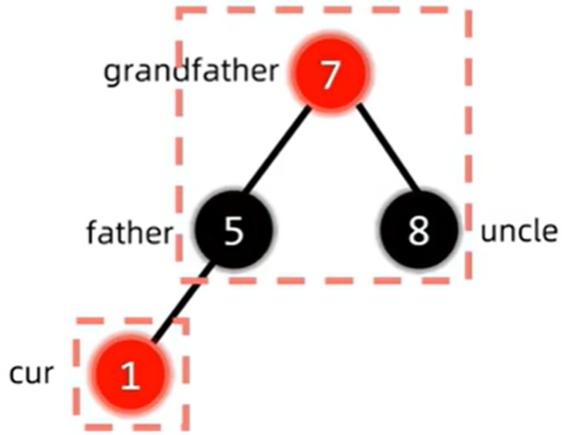
插入节点默认红色. 如果插入后，红黑树性质（不红红，根叶黑）被破坏，需要调整：

① 插入的是根节点 - 违反根叶黑 - 直接变黑

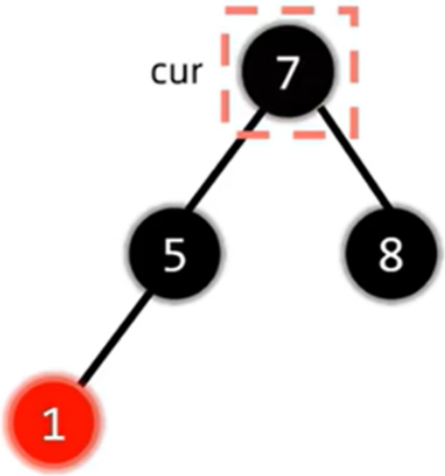
② 插入节点的叔叔是红色.



红（爸爸和叔叔）变黑，黑（爷爷）变红

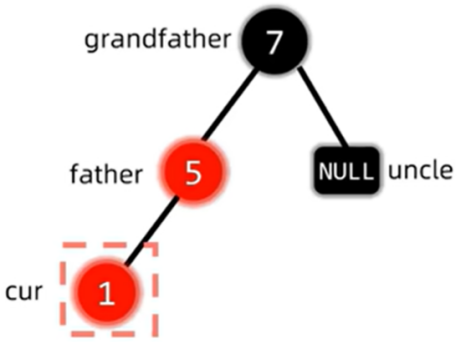


然后让cur指向爷爷，继续判定（发现违反根叶黑）.

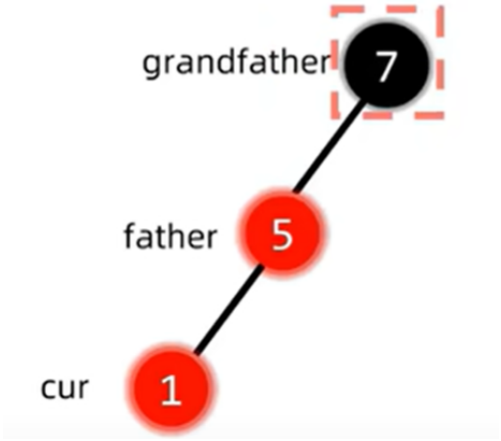


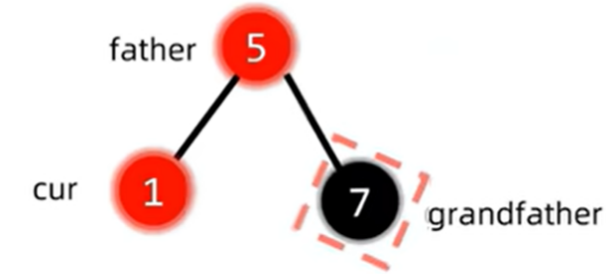
③ 插入节点的叔叔是黑色

> LL, LR, RL, RR

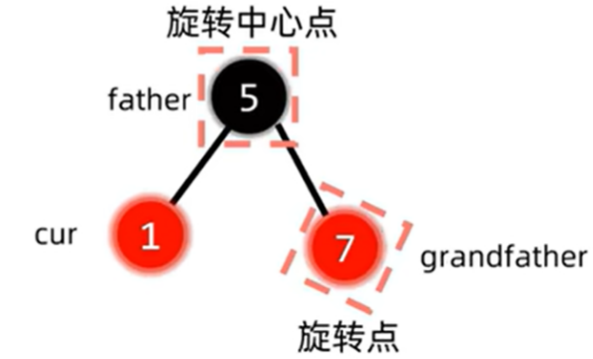


LL 型，以爷爷为旋转点进行右旋

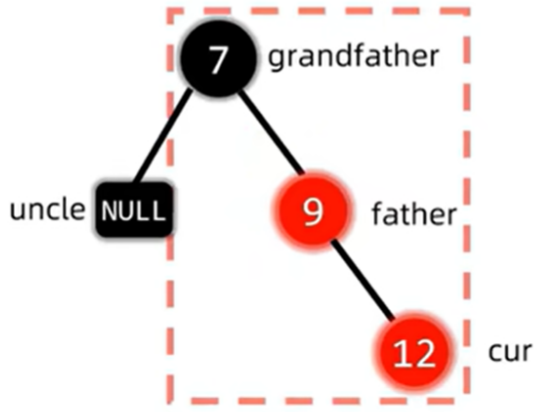


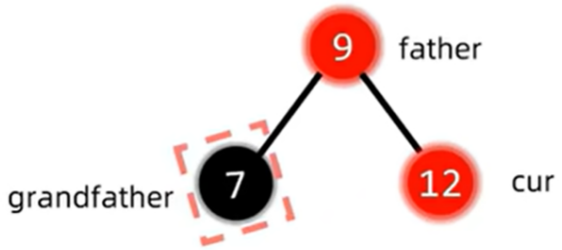


然后对旋转点和旋转中心点进行变色

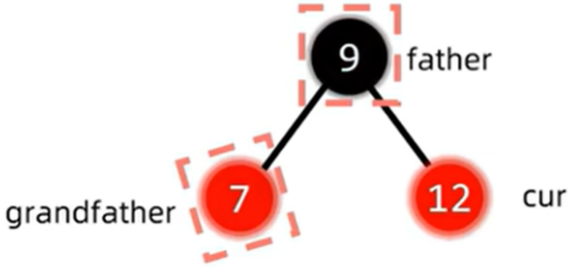


RR 型，以爷爷为旋转点进行左旋

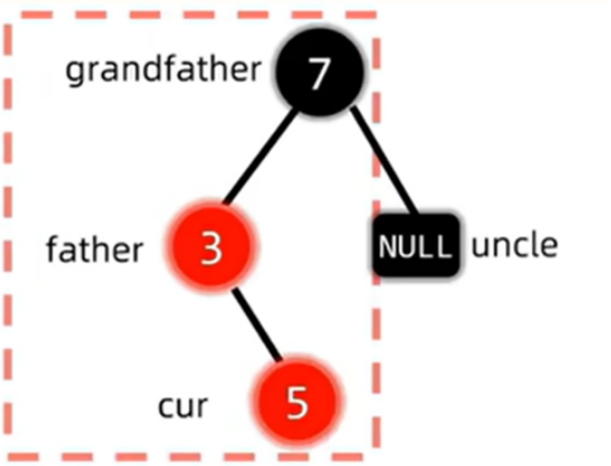


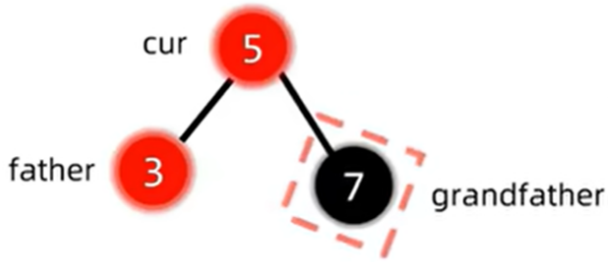


然后对旋转点和旋转中心点进行变色

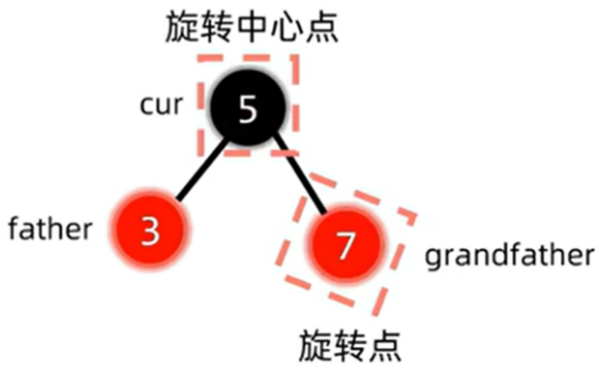


LR 型：先左旋左孩子，然后右旋爷爷

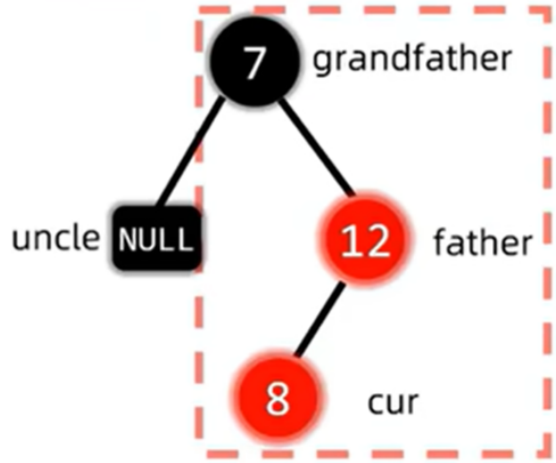




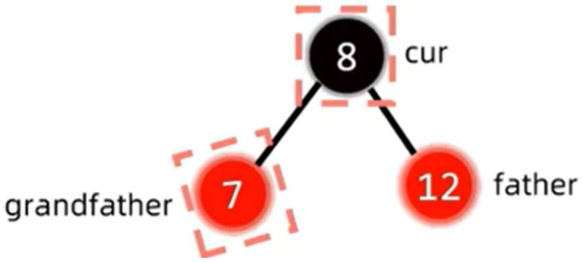
对旋转点和旋转中心点进行变色



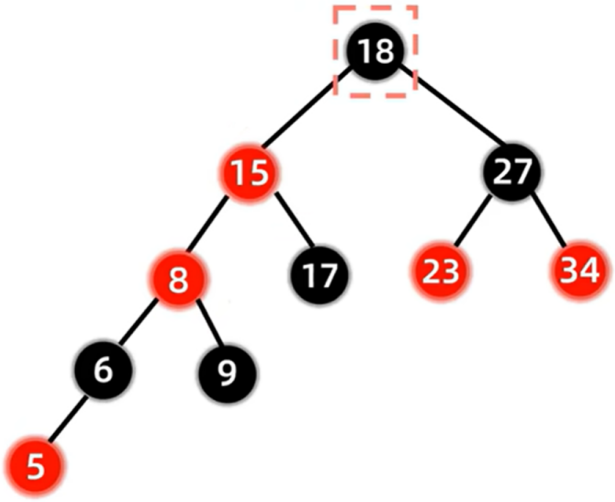
RL 型：先右旋右孩子，然后左旋爷爷



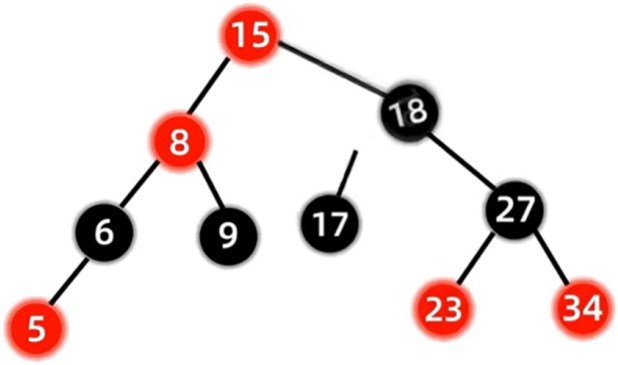
变色，最终结果：



注意，旋转操作可能涉及类似B树的子节点移动：



这里，LL，右旋爷爷 18 时，会把 15 原先的右节点变为 18 新的左节点. 18 本身变为 15 新的右节点：



红黑树的插入和删除操作比普通二叉搜索树多了颜色调整和旋转操作：

```c
#include <stdio.h>
#include <stdlib.h>

// 定义红黑树节点的颜色
typedef enum { RED, BLACK } Color;

// 红黑树节点结构
typedef struct RBTreeNode {
    int key;
    Color color;
    struct RBTreeNode *left, *right, *parent;
} RBTreeNode;

// 全局 NIL 节点（代表空节点）
RBTreeNode* NIL;

// 初始化 NIL 节点
void initializeNIL() {
    NIL = (RBTreeNode*)malloc(sizeof(RBTreeNode));
    NIL->color = BLACK;
    NIL->left = NIL->right = NIL->parent = NULL;
}

// 创建新节点
RBTreeNode* createNode(int key) {
    RBTreeNode* node = (RBTreeNode*)malloc(sizeof(RBTreeNode));
    node->key = key;
    node->color = RED;  // 新插入的节点默认为红色（维护黑路同的性质）
    node->left = node->right = node->parent = NIL;
    return node;
}

// 左旋
void leftRotate(RBTreeNode** root, RBTreeNode* x) {
    RBTreeNode* y = x->right;
    x->right = y->left;
    if (y->left != NIL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NIL)
        *root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
```

```
    x->parent = y;
}

// 右旋
void rightRotate(RBTreeNode** root, RBTreeNode* y) {
    RBTreeNode* x = y->left;
    y->left = x->right;
    if (x->right != NIL)
        x->right->parent = y;
    x->parent = y->parent;
    if (y->parent == NIL)
        *root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}

// 插入修复
void insertFixup(RBTreeNode** root, RBTreeNode* z) {
    while (z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            RBTreeNode* y = z->parent->parent->right;  // 叔叔节点
            if (y->color == RED) {  // Case 1: 叔叔是红色
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;  // 向上修复
            } else {
                if (z == z->parent->right) {  // Case 2: z 是父节点的右子节点
                    z = z->parent;
                    leftRotate(root, z);
                }
                // Case 3: z 是父节点的左子节点
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(root, z->parent->parent);
            }
        } else {
            RBTreeNode* y = z->parent->parent->left;  // 叔叔节点
            if (y->color == RED) {  // Case 1: 叔叔是红色
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;  // 向上修复
            } else {
                if (z == z->parent->left) {  // Case 2: z 是父节点的左子节点
                    z = z->parent;
                    rightRotate(root, z);
                }
                // Case 3: z 是父节点的右子节点
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                leftRotate(root, z->parent->parent);
            }
        }
    }
    (*root)->color = BLACK;  // 根节点始终是黑色
}

// 插入节点
void insert(RBTreeNode** root, int key) {
    RBTreeNode* z = createNode(key);
    RBTreeNode* y = NIL;
    RBTreeNode* x = *root;
```

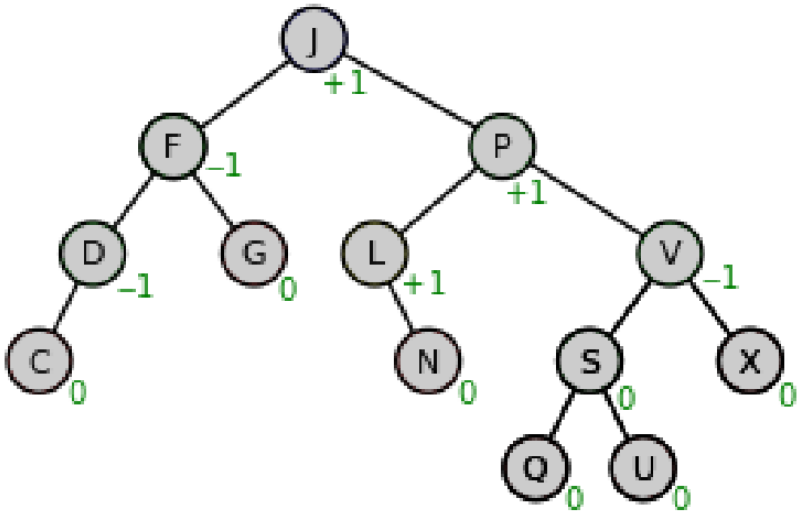
```
    while (x != NIL) {
        y = x;
        if (z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;
    if (y == NIL)
        *root = z;
    else if (z->key < y->key)
        y->left = z;
    else
        y->right = z;

    z->left = z->right = NIL;
    z->color = RED;

    insertFixup(root, z);
}

// 删除修复
void deleteFixup(RBTreeNode** root, RBTreeNode* x) {
    while (x != *root && x->color == BLACK) {
        if (x == x->parent->left) {
            RBTreeNode* w = x->parent->right;
            if (w->color == RED) {  // Case 1: 兄弟是红色
                w->color = BLACK;
                x->parent->color = RED;
                leftRotate(root, x->parent);
                w = x->parent->right;
            }
            if (w->left->color == BLACK && w->right->color == BLACK) {  // Case 2: 兄弟的子节点都是黑色
                w->color = RED;
                x = x->parent;
            } else {
                if (w->right->color == BLACK) {  // Case 3: 兄弟的左子节点是红色，右子节点是黑色
                    w->left->color = BLACK;
                    w->color = RED;
                    rightRotate(root, w);
                    w = x->parent->right;
                }
                // Case 4: 兄弟的右子节点是红色
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->right->color = BLACK;
                leftRotate(root, x->parent);
                x = *root;
            }
        } else {
            RBTreeNode* w = x->parent->left;
            if (w->color == RED) {  // Case 1: 兄弟是红色
                w->color = BLACK;
                x->parent->color = RED;
                rightRotate(root, x->parent);
                w = x->parent->left;
            }
            if (w->right->color == BLACK && w->left->color == BLACK) {  // Case 2: 兄弟的子节点都是黑色
                w->color = RED;
                x = x->parent;
            } else {
                if (w->left->color == BLACK) {  // Case 3: 兄弟的右子节点是红色，左子节点是黑色
                    w->right->color = BLACK;
                    w->color = RED;
                    leftRotate(root, w);
                    w = x->parent->left;
                }
                // Case 4: 兄弟的左子节点是红色
```

```
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->left->color = BLACK;
                rightRotate(root, x->parent);
                x = *root;
            }
        }
    }
    x->color = BLACK;
}

// 替换节点
void rbTransplant(RBTreeNode** root, RBTreeNode* u, RBTreeNode* v) {
    if (u->parent == NIL)
        *root = v;
    else if (u == u->parent->left)
        u->parent->left = v;
    else
        u->parent->right = v;
    v->parent = u->parent;
}

// 删除节点
void deleteNode(RBTreeNode** root, int key) {
    RBTreeNode* z = *root;
    RBTreeNode* y;
    RBTreeNode* x;

    // 查找要删除的节点
    while (z != NIL && z->key != key) {
        if (key < z->key)
            z = z->left;
        else
            z = z->right;
    }

    if (z == NIL) {
        printf("节点 %d 不存在! \n", key);
        return;
    }

    y = z;
    Color yOriginalColor = y->color;
    if (z->left == NIL) {
        x = z->right;
        rbTransplant(root, z, z->right);
    } else if (z->right == NIL) {
        x = z->left;
        rbTransplant(root, z, z->left);
    } else {
        y = z->right;
        while (y->left != NIL)
            y = y->left;
        yOriginalColor = y->color;
        x = y->right;
        if (y->parent == z)
            x->parent = y;
        else {
            rbTransplant(root, y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }
        rbTransplant(root, z, y);
        y->left = z->left;
        y->left->parent = y;
        y->color = z->color;
    }
```

```
    free(z);

    if (yOriginalColor == BLACK)
        deleteFixup(root, x);
}

// 中序遍历
void inorder(RBTreeNode* root) {
    if (root != NIL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// 释放红黑树节点
void freeTree(RBTreeNode* root) {
    if (root != NIL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
```

⑥ AVL 树

> AVL Tree
>
> self study



二叉搜索树的不足：如果数据本来就有序，效率会退化为 $O(n)$

The first self-balancing tree, invented by Russian scientists. We keep track of a tree's balance (difference in subtree heights) and perform rotations when balance is off

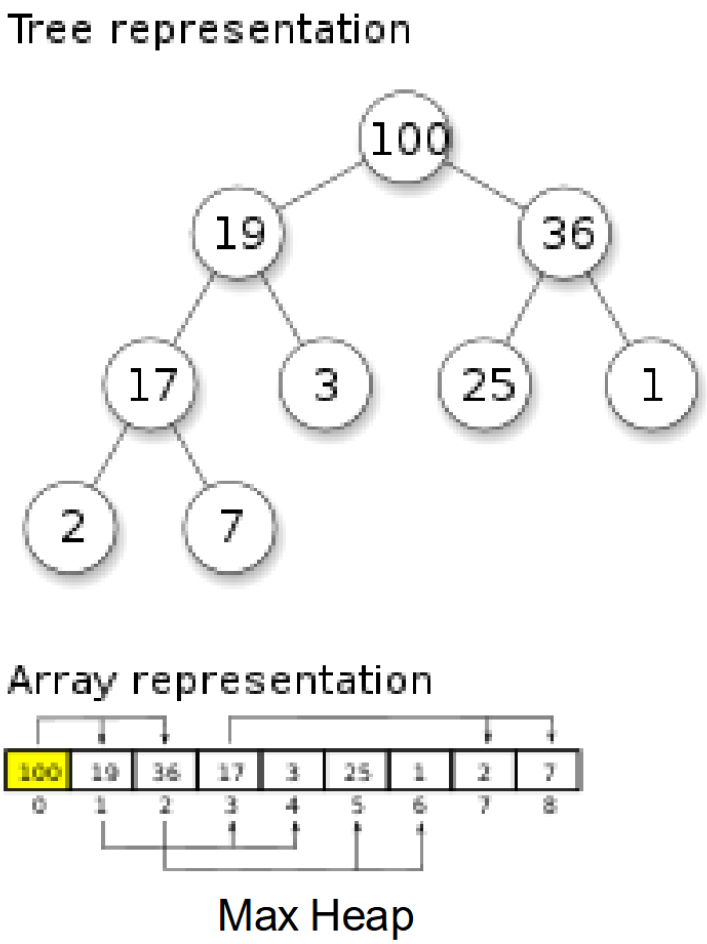
> 标的数字是平衡因子（右子树高度减左子树高度）

AVL tree is a self-balancing binary search tree

核心思想：在二叉搜索树的基础上，每次操作后检查是否失衡. 如果失衡，旋转调整.

⑦ *堆

> Heap
>
> 重点考察.



A max (min) heap, for any given node C, if P is the parent node of C, then $P.key > C.key$ ($P.key < C.key$)

Heap is a self-balancing binary tree

Application: Heapsort, Prim's Algorithm, Dijkstra's Algorithm

> 完全二叉树：堆的所有层（除了最后一层）都是满的，最后一层的节点从左到右连续填充
>
> 与二叉搜索树不同，堆是自上而下比大小.
>
> 最大堆：每个节点的值都大于等于其子节点的值.
>
> 最小堆：每个节点的值都小于等于其子节点的值.

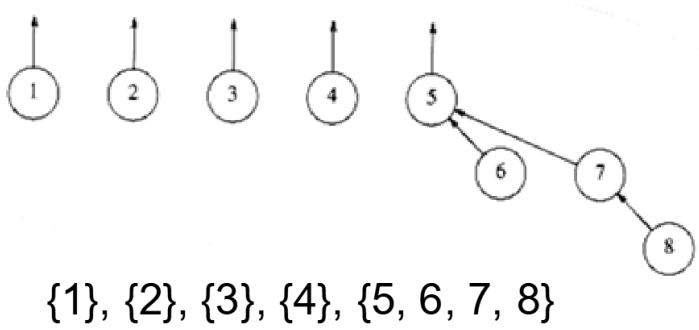
```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100  // 定义堆的最大容量

typedef struct {
    int data[MAX_SIZE]; // 存储堆元素的数组
    int size;           // 堆中元素的个数
} MaxHeap;

// 初始化堆
void initHeap(MaxHeap *heap) {
    heap->size = 0;
}

// 向上调整（插入时使用）
void siftUp(MaxHeap *heap, int index) {
    while (index > 0) {
```

```
        int parent = (index - 1) / 2; // 父节点索引
        if (heap->data[index] > heap->data[parent]) {
            // 交换当前节点和父节点
            int temp = heap->data[index];
            heap->data[index] = heap->data[parent];
            heap->data[parent] = temp;

            // 更新索引为父节点
            index = parent;
        } else {
            break;
        }
    }
}

// 插入元素到堆
void insert(MaxHeap *heap, int value) {
    if (heap->size >= MAX_SIZE) {
        printf("Heap is full!\n");
        return;
    }
    // 将新元素放到堆的最后
    heap->data[heap->size] = value;

    // 调整堆以满足最大堆性质
    siftUp(heap, heap->size);

    // 堆大小加一
    heap->size++;
}

// 向下调整（删除时使用）
void siftDown(MaxHeap *heap, int index) {
    while (2 * index + 1 < heap->size) { // 判断是否有左子节点
        int left = 2 * index + 1;  // 左子节点索引
        int right = 2 * index + 2; // 右子节点索引
        int largest = left;        // 假设左子节点是较大的

        // 如果右子节点存在且更大，则更新largest
        if (right < heap->size && heap->data[right] > heap->data[left]) {
            largest = right;
        }

        // 如果当前节点已经大于等于最大的子节点，调整结束
        if (heap->data[index] >= heap->data[largest]) {
            break;
        }

        // 否则交换当前节点和较大的子节点
        int temp = heap->data[index];
        heap->data[index] = heap->data[largest];
        heap->data[largest] = temp;

        // 更新索引为较大的子节点
        index = largest;
    }
}

// 删除堆顶元素（返回最大值）
int deleteMax(MaxHeap *heap) {
    if (heap->size <= 0) {
        printf("Heap is empty!\n");
        return -1;
    }

    int maxValue = heap->data[0]; // 堆顶元素

    // 将最后一个元素移到堆顶
```

```
    heap->data[0] = heap->data[heap->size - 1];
    heap->size--;

    // 调整堆以满足最大堆性质
    siftDown(heap, 0);

    return maxValue;
}

// 打印堆中的元素
void printHeap(MaxHeap *heap) {
    printf("Heap: ");
    for (int i = 0; i < heap->size; i++) {
        printf("%d ", heap->data[i]);
    }
    printf("\n");
}
```

⑧ *并查集

> Disjoint Set
>
> 重点考察.



{1}, {2}, {3}, {4}, {5, 6, 7, 8}

Disjoint set stores a collection of non-overlapping sets

Disjoint set is a forest of rooted oriented trees

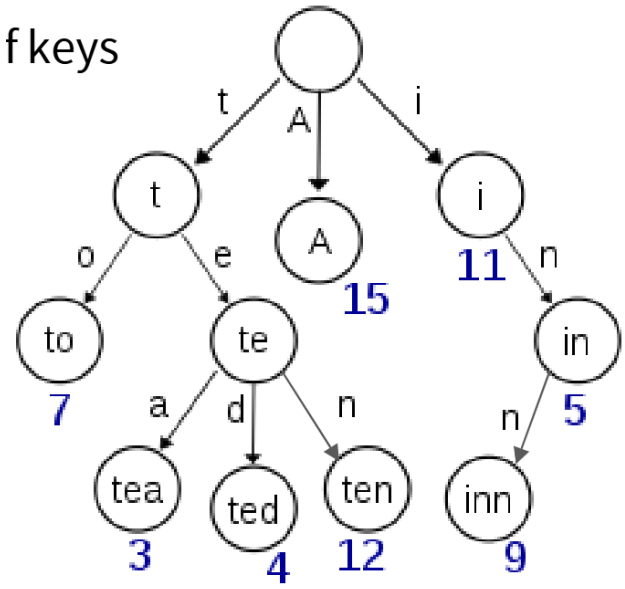
Application: Graph connectedness of undirected graph, Kruskal's minimum spanning tree Algorithm

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 1000  // 并查集最大节点数

typedef struct {
    int parent[MAX_SIZE];  // 存储每个元素的父节点
    int rank[MAX_SIZE];    // 存储每个集合的秩（树的高度）
} UnionFind;

// 初始化并查集（每个节点的父节点指向自己，秩为0）
void initUnionFind(UnionFind *uf, int n) {
    for (int i = 0; i < n; i++) {
        uf->parent[i] = i; // 每个节点的父节点初始化为自己
        uf->rank[i] = 0;   // 初始秩为0
    }
}

// 查找操作（带路径压缩）
int find(UnionFind *uf, int x) {
    if (uf->parent[x] != x) {
        // 递归查找父节点，并将当前节点的父节点直接指向根节点
        uf->parent[x] = find(uf, uf->parent[x]);
    }
```

```
    return uf->parent[x];
}

// 合并操作（按秩合并）
void unionSets(UnionFind *uf, int x, int y) {
    int rootX = find(uf, x);
    int rootY = find(uf, y);

    if (rootX != rootY) {
        // 根据秩进行合并，秩小的树挂到秩大的树下
        if (uf->rank[rootX] < uf->rank[rootY]) {
            uf->parent[rootX] = rootY;
        } else if (uf->rank[rootX] > uf->rank[rootY]) {
            uf->parent[rootY] = rootX;
        } else {
            // 如果秩相等，任选一棵树作为根，并将其秩加1
            uf->parent[rootY] = rootX;
            uf->rank[rootX]++;
        }
    }
}

// 判断两个元素是否在同一个集合中
int isConnected(UnionFind *uf, int x, int y) {
    return find(uf, x) == find(uf, y);
}
```

⑨ 前缀树

> Trie，又叫字典树
>
> self study



A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

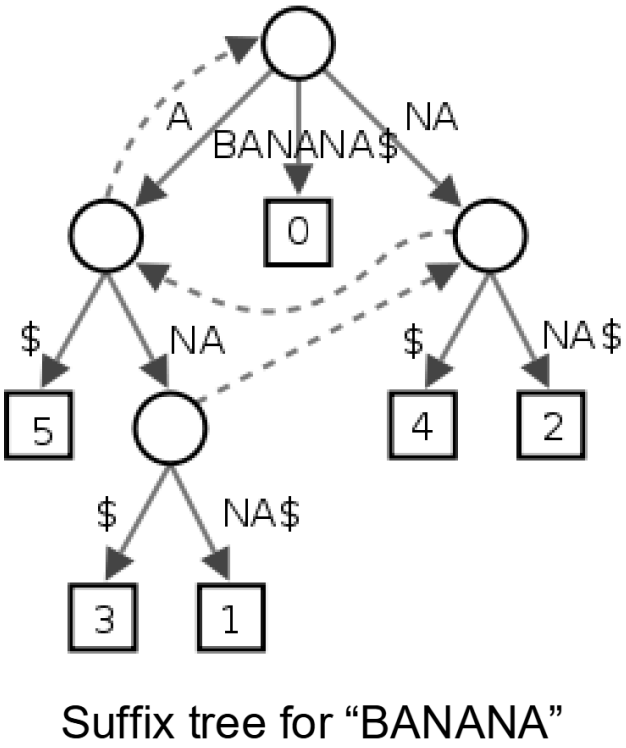
Trie is used for locating specific keys within a set of keys

Trie is an oriented search tree

Application: Dictionary representation

⑩ 后缀树

> Suffix Tree
>
> self study



Suffix tree for “BANANA”

Suffix tree is a compressed trie containing all the suffixes of a given string

Suffix tree is an oriented search tree

Application: Solving longest common substring problem in linear time

8. 图

> Graph

8.1 定义

> Graph Definition

A graph is a set of vertices plus a set of edges that connect pairs of distinct vertices (with at most one edge connecting any pairs).

Graph Mathematical Definition

① A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E

② Each edge is a pair (v, u), where $v, u \in V$

③ Vertex u is adjacent to vertex v if and only if $(v, u) \in E$

④ An edge may have weight, direction

⑤ A vertex may have weight

8.2 分类

> Types of Graph

① Directed Graph (unidirected graph): The pairs of node (edges) are ordered (or not)

② Weighted Graph: The edge contains a third component, called as weight or cost

③ Acyclic Graph: A graph contains no cycles

④ Simple Graph: No multiple edges between a pair of vertex, no edges connecting the same vertex

> 无自连，无重复边
>
> 有向简单图允许双向边，但不允许同向多条.

⑤ Connected Graph: There is a path from every vertex to every other vertex in undirected graph

> 连通图

⑥ Strongly Connected Graph: There is a path from every vertex to every other vertex in directed graph

> 强连通和弱连通是专门针对有向图的概念.
>
> 强连通：有向图中，任意两个顶点 u 和 v 都存在一条从 u 到 v 的路径和一条从 v 到 u 的路径，则该图是强连通图.
>
> 弱连通：如果将有向图中的所有有向边看作无向边后，整个图是连通的，则称该图是弱连通图.

⑦ Complete Graph (Clique): A simple undirected graph in which every pair of distinct vertices is connected by a unique edge

> 完全图：两两连线 无向

⑧ Planar Graph: A graph can be embedded in the plane, no cross among edges.

> 平面图：如果能通过重新排列边或顶点的方式，使得图中的边在平面上互不交叉，那么这个图就是一个平面图.

8.3 数据结构

> Graph Abstract Data Type and Data Structures

Graph Abstract Data Type

- To implement directed or undirected graph based on the definition of graph in graph theory
- Operations:
  - `adjacent(x, y)` : test whether there is an edge from vertex x to vertex y
  - `neighbors(x)` : list all vertices y such that there is an edge from vertex x to vertex y
  - `add_vertex(x)` : add vertex x
  - `remove_vertex(x)` : remove vertex x
  - `add_edge(x, y)` : add edge from vertex x to vertex y
  - `remove_edge(x, y)` : remove edge from vertex x to vertex y
  - `get_vertex_value(x)` : return the value of vertex x
  - `set_vertex_value(x, v)` : set the value of vertex x
  - `get_edge_value(x, y)` : return the value of edge (x, y)
  - `set_edge_value(x, y, v)` : set the value of edge (x, y)

Graph Data Structures

① 邻接矩阵

> Adjacency Matrix

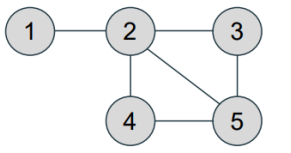
A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices.

The adjacency matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

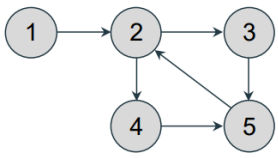
$$a_{ij} = \begin{cases} 1 & if\ (i,j) \in E \\ 0 & otherwise \end{cases}$$

- Memory requirement: $O(|V|^2)$, independent of $|E|$
- For undirected graph, the size of the matrix can be reduced to half (theoretically)

Adjacency Matrix Visualization



	1	2	3	4	5
1	0	1	0	0	0
2	1	0	1	1	1
3	0	1	0	0	1
4	0	1	0	0	1
5	0	1	1	1	0



	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	1	0
3	0	0	0	0	1
4	0	0	0	0	1
5	0	1	0	0	0

undirected：两个 vertices 只要有 edge 就为 1

directed：row 指向 column 才为 1

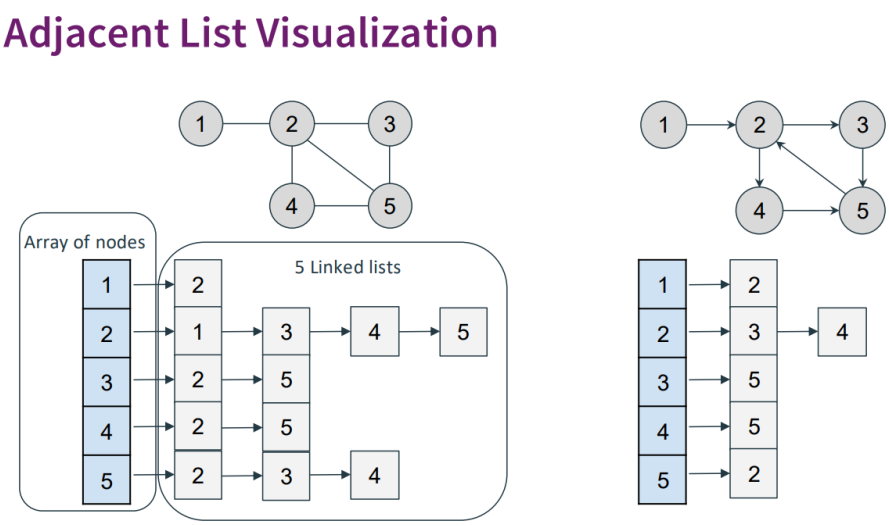
② 邻接表

> Adjacency List

Vertices are stored as records, and every vertex stores a list of adjacent vertices.

- The adjacency list representation of a graph G consists of an array `adj[]` of $|V|$ linked lists, one for each vertex
- The adjacency list `adj[u]` contains all the vertices v such that there is an edge $(u, v) \in E$.
- The vertices are stored in an arbitrary order
- Memory requirement: $O(|V| + |E|)$.

Adjacent List Visualization



undirected：一个 array 包含以每个 vertex 为序号的指针，指向 list，每个 list 包含与该 vertex 共边的所有 vertex

> 注意，Array 中存放的是指针变量，list 是结构体变量一个接一个，二者有本质区别.
>
> Array 中的每个指针要么是 NULL，要么指向一个结构体变量（某个 list 的头节点）.

directed：同上，但是 list 只包含从该 vertex 出发，指向另一个 vertex 的情况

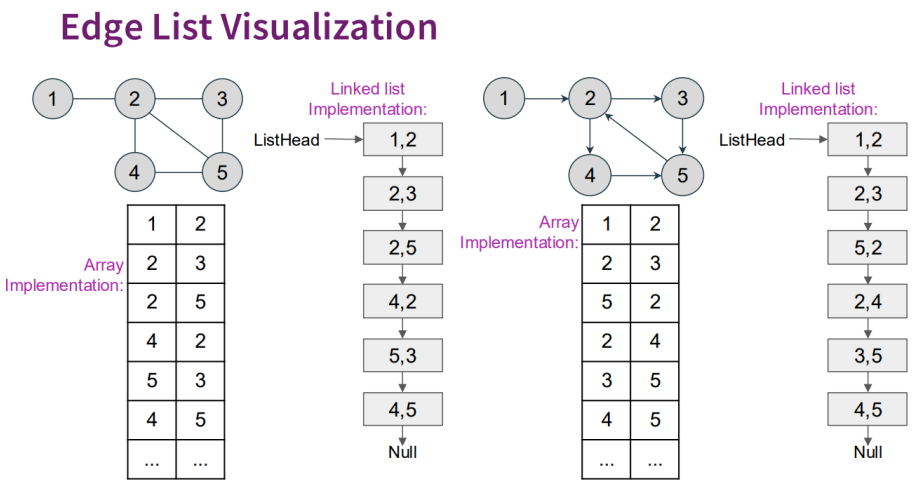
③ 边列表

> Edge List

A list of edges (linked list implementation, array implementation)

- The edge list representation of a graph G consists of a list of $|E|$ edges
- For a undirected graph
  - Linked list implementation is a list of a struct, i.e. pair of vertices
  - Array implementation is a two-column matrix
- The edges are stored in an arbitrary order
- Memory requirement：$O(|E|)$.

Edge List Visualization



List 的每个 Node 包含一条 edge 连的两个 vertices.

directed：每个节点内部的两个 vertices 是有序的.

④ 关联矩阵

> Incidence Matrix
>
> 注意和邻接矩阵区分.

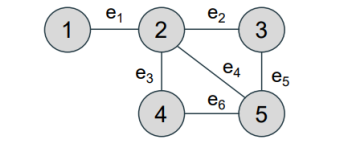
A two-dimensional boolean matrix, in which the rows represent the vertices and columns represent the edges.

- The incidence matrix representation of a graph G consists of a $|V| \times |E|$ boolean matrix that shows the relations between $|V|$ vertice and $|E|$ edges

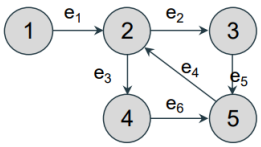
$$b_{ij} = \begin{cases} 1 & if\ v_i \in e_j \\ 0 & otherwise \end{cases}$$

- The vertices and edges are stored in an arbitrary order
- Memory requirement: $O(|V| \cdot |E|)$
- Incidence matrix can be extended to loop structures, clusters, multigraphs, etc

Incidence Matrix Visualization



	1	2	3	4	5	6
1	1	0	0	0	0	0
2	1	1	1	1	0	0
3	0	1	0	0	1	0
4	0	0	1	0	0	1
5	0	0	0	1	1	1



	1	2	3	4	5	6
1	-1	0	0	0	0	0
2	1	-1	-1	1	0	0
3	0	1	0	0	-1	0
4	0	0	1	0	0	-1
5	0	0	0	-1	1	1

row 是 vertex，colomn 是 edge

对于 directed，出发点为 -1，到达点为 1

8.4 图算法

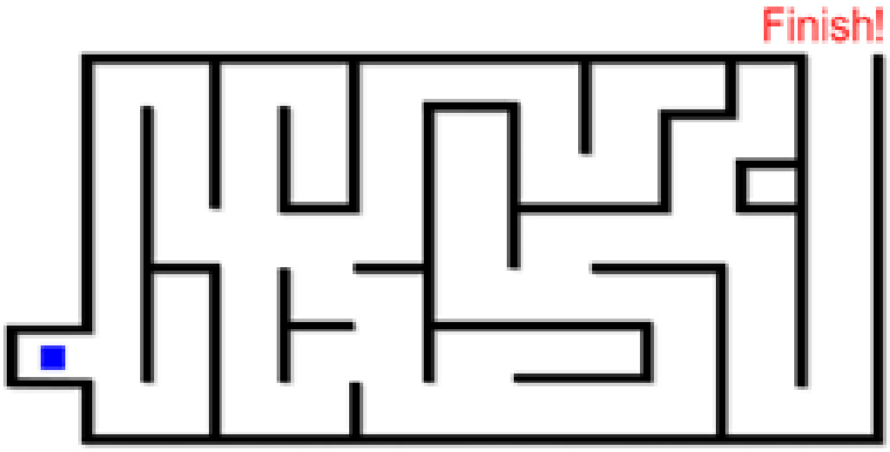
> Lecture 10 Graph Algorithms

Outline

- Search Algorithms
- Graph Connectedness
- Shortest Path Algorithms
- Minimum Spanning Tree
- Maximum Flow, Minimum Cut
- Maximum Bipartite Matching Problems
- Cycle Detection
- Graph Coloring

8.4.1 搜索算法

> Search Problems in Graph



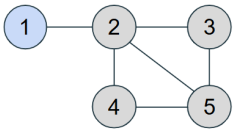
- There is a searcher (start position) and at least one target
- The problem is to see if the searcher can find the target in a path
- We need to ensure to explore every part of the map
- We do not retrace unnecessary steps
- Algorithms: Breadth-First Search, Depth-First Search
- Application: Maze exploration

① *深度优先搜索

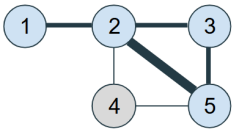
> Depth-First Search

- The strategy in depth-first search (DFS) is to search deeper in the graph whenever possible
- We explore out of the most recent discovered vertex v that still has unexplored edges leaving it
- When all v's' edges have been explored, the search backtracks to explore edges leaving the vertex from which v is discovered.
- If any discovered vertices remain, then one of them is selected as a new source and the search is repeated.
- The entire process is repeated until all vertices are discovered, or the termination condition is met.

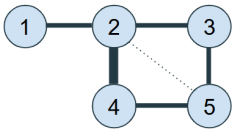
Start DFS at node 1



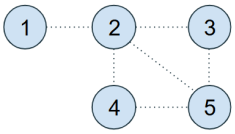
1 → 2 → 3 → 5
At node 5, it is found that node 2 is visited



1 → 2 → 3 → 5 → 4
At node 4, it is found that node 2 is visited



No more new nodes are found
DFS is completed



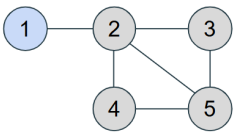
② *广度优先搜索

> Breath-First Search

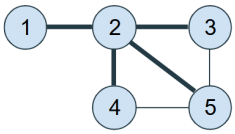
- Breath-first search (BFS) is another algorithm for searching a graph
- Given a graph $G = (V, E)$, we pick a distinguished source vertex s, BFS systematically explores the edges of G to discover every vertex that is reachable from s.

- It computes the shortest distance (smallest number of edges) from s to each reachable vertex
- Works on both directed or undirected graphs
- Breath: discovers all vertices at distance k from s before discovering any vertex at distance $k+1$.

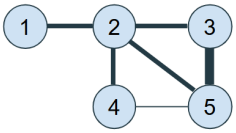
Start BFS at node 1



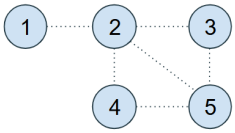
1 → 2 → 3 → 4 → 5
At node 2, it visit nodes 3, 4 and 5



1 → 2 → 3 → 4 → 5
At node 3, it cannot find any new nodes

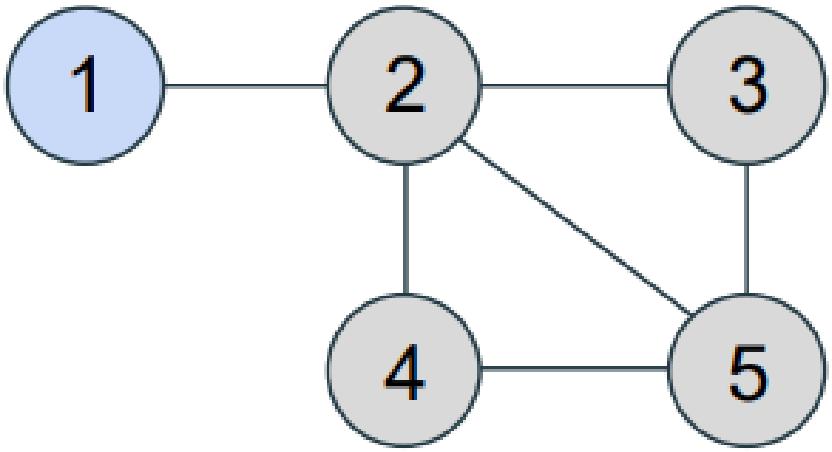


Similarly, at nodes 4 and 5, no more new nodes are found, and BFS is completed



③ 图遍历

> Graph Traversal



- Depth-first traversal is the generalization of pre-order traversal
  - Starting at searching, and recursively traversing all neighbors of it
  - e.g. 1,2,3,5,4
- Breath-first traversal is the generalization of level-order traversal
  - Processing vertices in layers, vertices closest to the searcher are evaluated first
  - e.g. 1,2,3,4,5
- Time complexity for both DFS and BFS is $O(|V|+|E|)$.

8.4.2 图的连通性

> Graph Connectedness

有向图的强连通分量：极大子图，两两互相到达.

① *并查集应用

② Kosaraju 算法

Kosaraju 算法是一种用于在**有向图**中找到**强连通分量**（SCC，Strongly Connected Components）的线性时间算法. 一个强连通分量是一个子图，其中任意两个顶点之间都可以互相到达. Kosaraju算法以其简单性和高效性（时间复杂度为 `O(V + E)`）而广为人知

核心思想：

1. 对原图进行DFS，在每个顶点结束时，将顶点压入栈中，生成的栈按照顶点完成时间从小到大排序.
2. 构建逆边图：将图中所有边方向反转，得到逆边图.
3. 在逆边图中按出栈顺序进行 DFS，所有被访问到的顶点组成一个强连通分量.

8.4.3 最短路径算法

> Shortest Path Algorithms

① *Dijkstra 算法

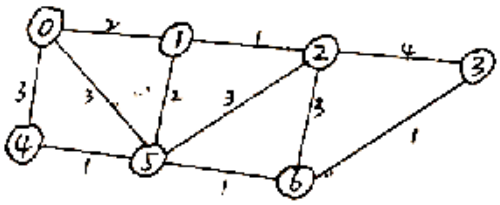
> Dijkstra's Algorithm

Dijkstra算法是一种经典的**单源最短路径算法**，用于计算从图中某一个源点到其他所有顶点的最短路径。它适用于**加权有向图或无向图**，但要求边的权值**非负**.

核心思想：**贪心策略**，每次选择当前距离源点最近的节点，将其加入到确定的最短路径集合中，并更新与该节点相邻的其他节点的最短距离.

维护一个距离向量和一个 previous 向量：

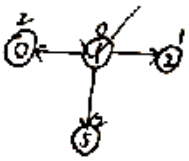
3. You are going to simulate dijkstra's algorithm in the question. Source is node 1, sink is node 3. (20%)



Beginning:

distance vector (INT_MAX means ∞)
INT_MAX, 0, INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX
previous vector (-1 means not found yet)
-1, -1, -1, -1, -1, -1, -1

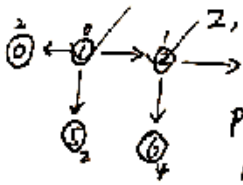
Step 1: Choose 1



distance vector
2, 0, 1, INT_MAX, INT_MAX, 2, INT_MAX
previous vector
1, -1, 1, -1, -1, 1, -1

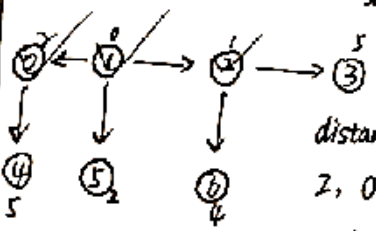
Step 2: Choose 2

distance vector



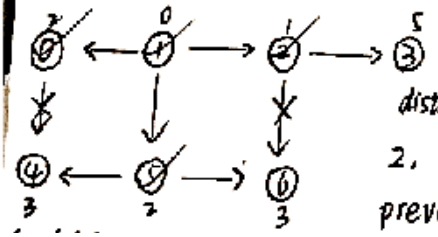
2, 0, 1, 5, INT_MAX, 2, 4
previous vector
1, -1, 1, 2, -1, 1, 2

Step 3: Choose 0 (If there're multiple options, select the node with the smallest value)

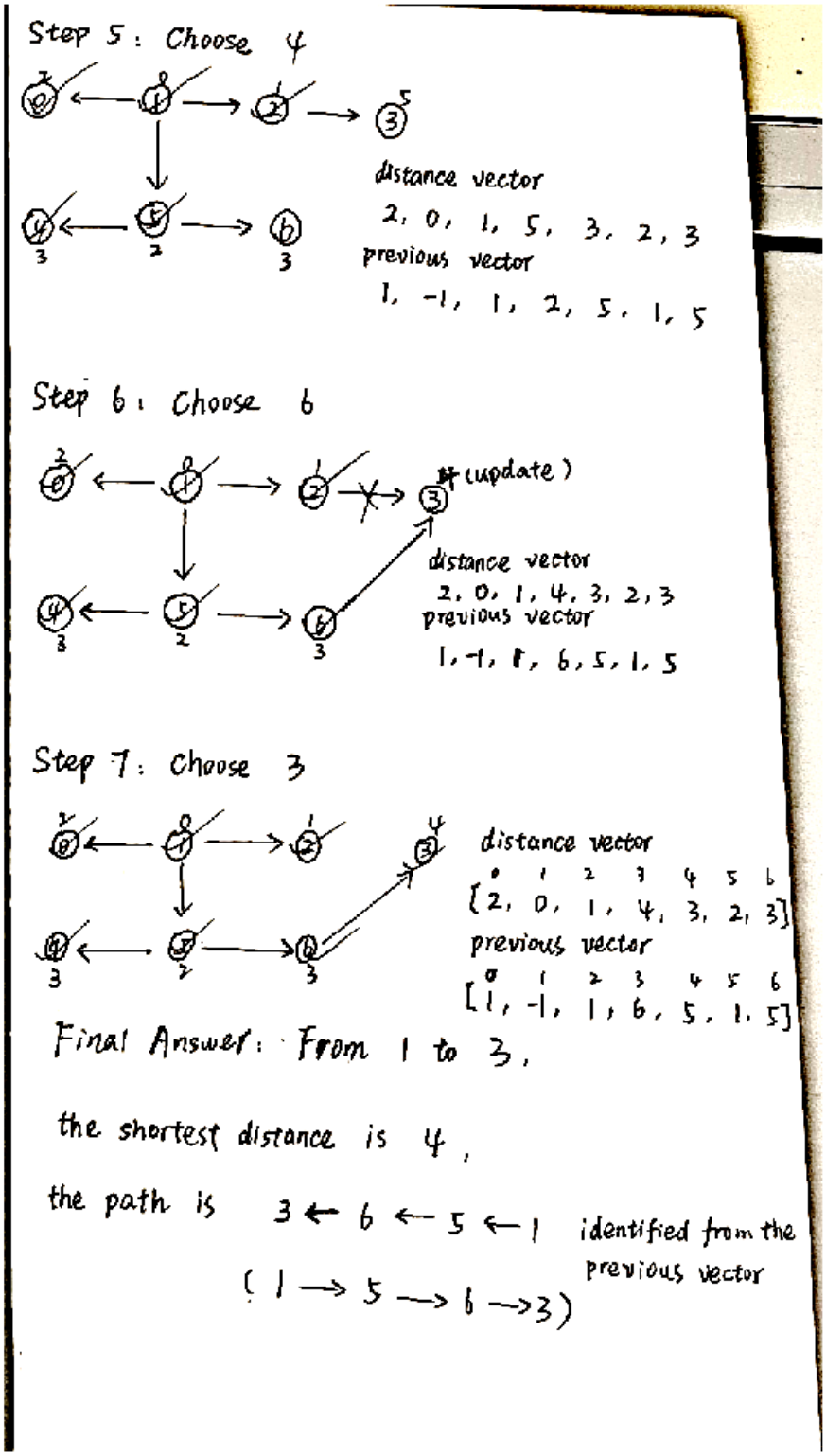


distance vector
2, 0, 1, 5, 5, 2, 4
previous vector
1, -1, 1, 2, 0, 1, 2

Step 4: Choose 5



distance vector
2, 0, 1, 5, 3, 2, 3
previous vector



② ***Bellman-Ford 算法**

> Bellman-Ford Algorithm

Bellman-Ford算法是一种经典的**单源最短路径算法**，用于计算从图中某个源点到所有其他顶点的最短路径. 它适用于**加权有向图或无向图**，并且**支持负权值边**，是 Dijkstra 算法的重要补充。Bellman-Ford算法还能检测图中是否存在**负权值环**.

核心思想：Bellman-Ford算法基于**动态规划**思想，逐步松弛（Relaxation）所有边的权值。它的核心操作是不断尝试通过某条边，更新某个顶点的当前最短路径，直到所有顶点的最短路径稳定为止.

1. 初始化：同 dijkstra
2. 松弛所有边：对于每条边 `(u, v, w)`，如果通过顶点 `u` 到达顶点 `v` 的距离比当前记录的 `dist[v]` 更短，则更新 `dist[v]`
3. 重复松弛：重复上述松弛操作 `V-1` 次（其中 `V` 是顶点数），因为一个最短路径最多包含 `V-1` 条边
4. 检测负权环：额外执行一次松弛操作，如果此时还能更新某个顶点的最短路径，则图中存在负权环

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<limits.h>

/*
You are going to implement the edge list using linked list
for an undirected, weighted and simple graph.
*/
typedef struct edge_t{
    int v;
    int w;
    int weight;
} Edge;

typedef struct graph_t{
    int V;
    int E;
    int Emax;  //max number of edge
    Edge *edgelist;
    int *vertexvalue;
}EdgeList;

/*
Return an empty list with at most V vertice and Emax edges
All entries in G->edgelist and G->vertexvalue are init as 0
*/

EdgeList *el_graph_init(int V, int Emax){
    EdgeList *G = malloc(sizeof(EdgeList));
    G->V = V;
    G->E = 0;
    G->Emax = Emax;
    G->edgelist = malloc(sizeof(Edge)*Emax);
    memset(G->edgelist, 0, sizeof(Edge)*Emax);
    G->vertexvalue = malloc(sizeof(int)*V);
    memset(G->vertexvalue, 0, sizeof(int)*V);
    return G;
}

/*
Insert edge e to G
If e already exists in G, update the weight
If e not in G, edge e is inserted to the last entry of the edgelist
When you insert new edge ex to edgelist, make sure ex.v < ex.w
*/

void el_insert_edge(EdgeList *G, Edge e){
    int v = e.v, w = e.w;
    //if(v>w){
    //    v = e.w;
    //    w = e.v;
    //}

    for(int i=0; i<G->E; i++){
        if(G->edgelist[i].v == v && G->edgelist[i].w == w){
            G->edgelist[i].weight = e.weight;
            return;
        }

    }
```

```
    if(G->E == G->Emax) return;

    G->edgelist[G->E].v = v;
    G->edgelist[G->E].w = w;
    G->edgelist[G->E].weight = e.weight;
    G->E++;

}


/*
Remove edge e from G
You need to check all three entries of e
If e does not in G, do nothing
If e exists in G, replace current slot by the last edge in the edgelist
*/

void el_remove_edge(EdgeList *G, Edge e){
    int v = e.v, w = e.w;
    if(e.weight == 0) return;

    if(v>w){
        v = e.w;
        w = e.v;
    }

    for(int i=0; i< G->E; i++){
        if(G->edgelist[i].v == v && G->edgelist[i].w == w && G->edgelist[i].weight == e.weight){
            G->E--;
            G->edgelist[i].v = G->edgelist[G->E].v;
            G->edgelist[i].w = G->edgelist[G->E].w;
            G->edgelist[i].weight = G->edgelist[G->E].weight;
            return;
        }
    }
}

/*
Search edge e in G
Return the weight of e in G
Note: e.weight is don't care in this function
*/

int el_find_edge(EdgeList *G, Edge e){
    int v = e.v, w = e.w;
    if(v > w){
        v = e.w;
        w = e.v;
    }

    for(int i=0; i<G->E; i++){
        if(G->edgelist[i].v == v && G->edgelist[i].w == w){
            return G->edgelist[i].weight;
        }
    }
    return 0;
}

/* Set the value of vertex V */
void el_set_vertex_value(EdgeList *G, int v, int value){
    G->vertexvalue[v] = value;
}

/* Return the value of vertex v */
int el_get_vertex_value(EdgeList *G, int v){
    return G->vertexvalue[v];
}
```

```
/*
Free the edge list
Return NULL pointer
*/

EdgeList *el_free(EdgeList *G){
    if(G==NULL) return NULL;
    free(G->edgelist);
    free(G->vertexvalue);
    free(G);
    return NULL;
}

/*
compare the edge weight;
if the edge weight is the same, compare the node
find the index with the smallest value
*/

void el_sort_edge(EdgeList *G){
    int minpos;
    int u, v, w;
    for(int i = 0; i< G->E; i++){
        minpos = i;
        for(int j = i+1; j<G->E; j++)
            if(G->edgelist[minpos].weight > G->edgelist[j].weight || (G->edgelist[minpos].weight == G->
edgelist[j].weight && (G->edgelist[minpos].v < G->edgelist[j].v ||(G->edgelist[minpos].v == G-
>edgelist[j].v && G->edgelist[minpos].w < G->edgelist[j].w))))
                minpos = j;

        u = G->edgelist[minpos].v;
        v = G->edgelist[minpos].w;
        w = G->edgelist[minpos].weight;

        G->edgelist[minpos].v = G->edgelist[i].v;
        G->edgelist[minpos].w = G->edgelist[i].w;
        G->edgelist[minpos].weight = G->edgelist[i].weight;

        G->edgelist[i].v = u;
        G->edgelist[i].w = v;
        G->edgelist[i].weight = w;

    }

}

int main(int argc, char *argv[]){

    FILE *fin, *fout;
    int i, j, u, v, w, n, e, cnt;
    int contain_cycle;
    int distance[100], previous[100];
    EdgeList *G;
    Edge tmpedge;

    fin = fopen(argv[1], "r");
    fout = fopen(argv[2], "w");

    /* Input format */
    fscanf(fin, "%d%d", &n, &e);

    /* You may need some initialization */
    G = el_graph_init(n, e);
    for(int i = 0; i<n; i++)
        distance[i] = INT_MAX;
    distance[0] = 0;
    for(int i=0; i<n; i++)
```

```
        previous[i] = -1;


    /* Input format */
    for (i = 0; i < e; i++) {
        fscanf(fin, "%d%d%d", &u, &v, &w);
        /* Your code here */
        tmpedge.v = u;
        tmpedge.w = v;
        tmpedge.weight = w;
        el_insert_edge(G, tmpedge);
    }

    el_sort_edge(G);

    /* Your code here */
    for(int i=0; i<n-1; i++){
        /* Output format */
        int update[10000];
        memset(update, 0, sizeof(int)*e);
        for (int j = 0; j < e; j++){
                //printf("j=%d ", j);
                //printf("distance j.v = %d \n", distance[G->edgelist[j].v]);
                //printf("distance j.w = %d \n", distance[G->edgelist[j].w]);
                //printf("j.weight = %d \n", G->edgelist[j].weight);

             if(distance[G->edgelist[j].v] != INT_MAX &&(distance[G->edgelist[j].v] + G-
>edgelist[j].weight < distance[G->edgelist[j].w])){
                update[j] = 1;
                //printf("debug1\n");

             }

        }

        for(int j=0; j<e; j++){
            if(update[j] == 1){
                distance[G->edgelist[j].w] = distance[G->edgelist[j].v] + G->edgelist[j].weight;
                previous[G->edgelist[j].w] = G->edgelist[j].v;
            }
        }

        for(int j=0; j< n; j++)
            fprintf(fout, "%d ", distance[j]);

        for (int j = 0; j < n - 1; j++)
            fprintf(fout, "%d ", previous[j]);
        fprintf(fout, "%d\n", previous[n-1]);

    }

    contain_cycle = 0;
    for(int i=0; i<e; i++){
        if(distance[G->edgelist[i].v] + G->edgelist[i].weight < distance[G->edgelist[i].w]){
            contain_cycle = 1;
            break;
        }

    }



    /* Output format */
    if (contain_cycle) {
        fprintf(fout, "The graph contains negative cycles.");
    } else {
        fprintf(fout, "The graph does not contain negative cycles.");
    }
```

```
    fclose(fin);
    fclose(fout);

    return 0;
}
```

③ *Floyd-Warshall 算法

> Floyd-Warshall Algorithm

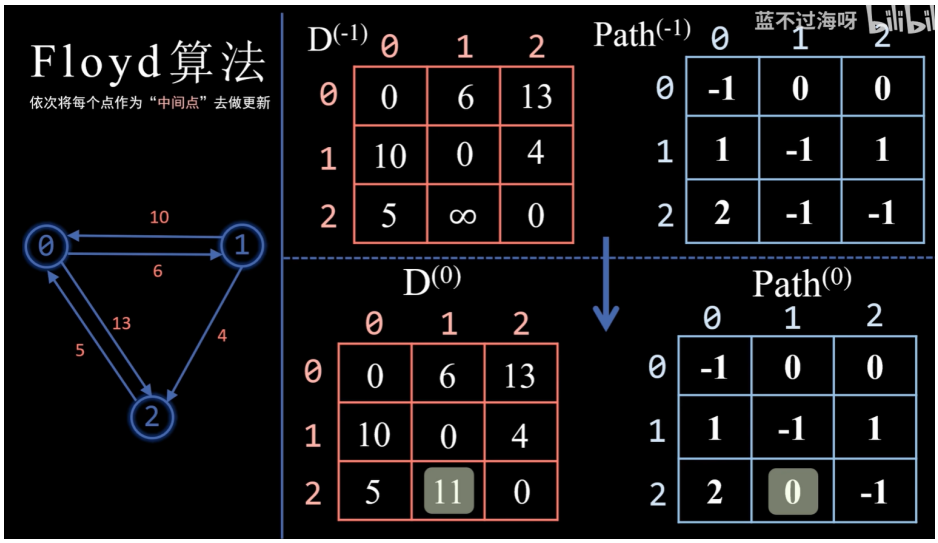
Floyd-Warshall算法是一种经典的**多源最短路径算法**，用于计算**加权有向图**中**任意两点之间的最短路径。**它适用于边权值**非负或带负权**的图，但不适用于含有**负权环**的图。该算法基于**动态规划**思想，通过逐步更新路径权值，最终得出所有顶点对之间的最短路径

Floyd-Warshall算法的核心在于逐步引入中间顶点来更新路径. 如果从顶点 `u` 到顶点 `v` 的最短路径经过某个中间顶点 `k`，那么最短路径的权值可以表示为：

```
dist[u][v] = min(dist[u][v], dist[u][k] + dist[k][v])
```

借助这个公式，算法逐步尝试使用所有顶点作为中间点来更新路径，最终得到所有顶点对之间的最短路径.

维护一个邻接矩阵和path矩阵：



④ SPFA

> Shortest Path Faster Algorithm

SPFA（Shortest Path Faster Algorithm）是**单源最短路径算法**的一种，是对**Bellman-Ford算法**的优化版本. 它通过引入**队列**来减少不必要的松弛操作，从而提高算法效率. SPFA算法可用于求解带有**负权边**的最短路径，并且可以检测**负权环**

核心思想：使用队列存储可能需要更新的顶点，并通过松弛操作更新邻接顶点的最短路径. 如果某个顶点的最短路径被更新，则将该顶点加入队列，从而避免对所有顶点重复松弛

8.4.4 最小生成树

> Minimum Spanning Tree

① *Prim 算法

> Prim's Algorithm
>
> 加点法

计算加权无向连通图的最小生成树.

核心思想：从一个起始顶点开始，逐步将权值最小的边加入树中，同时保证树的结构不形成环，直到所有顶点都包含在树中.

起始点任意选择.

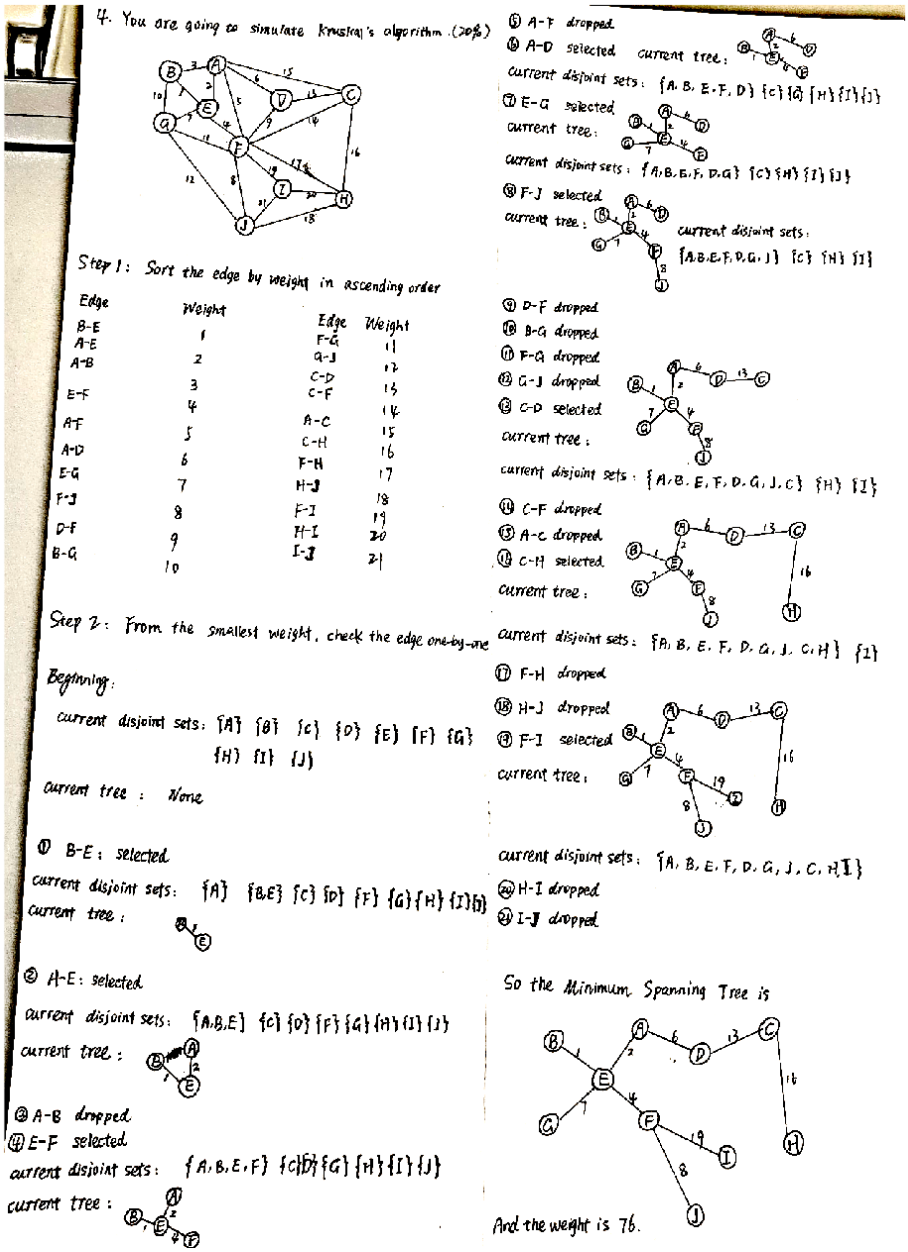
② *Kruskal 算法

> Kruskal's Algorithm
>
> 加边法

计算加权无向连通图的最小生成树.

核心思想：从小到大选择边，并逐步将边加入树中，同时避免形成环，直到最小生成树包含所有顶点

前置工作：升序排序边权重，维护并查集



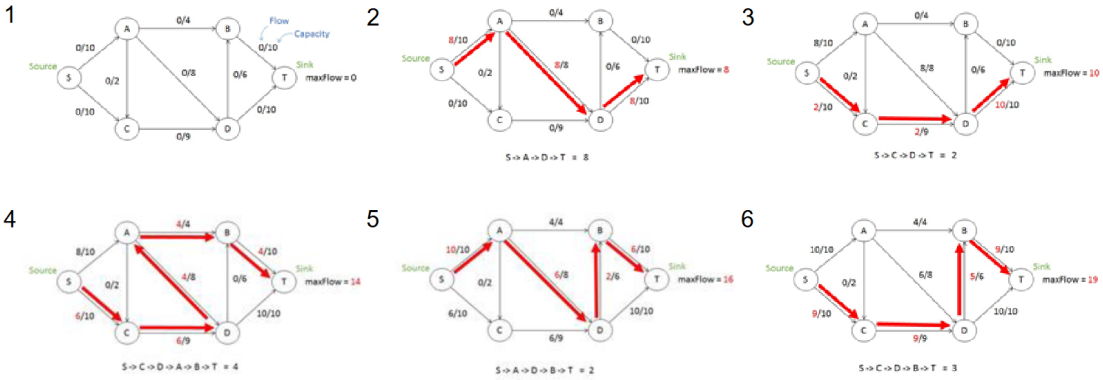
8.4.5 最大流最小割

> Maximum Flow, Minimum Cut

① Ford-Fulkerson 算法

> Ford-Fulkerson Algorithm

Ford-Fulkerson 算法是一种用于解决**最大流问题**的经典算法。它通过增广路径不断改进流量，直到不能找到增广路径为止。该算法广泛用于网络流问题中.



引用：ENGG2440 离散数学

10.2 Network Flow

> 网络流：directed $G = (V, E)$

定义

capacity $c(u, v)$: $c(u, v) > 0,\ \forall\ (u, v) \in E;\ c(u, v) = 0,\ \forall\ (u, v) \notin E.$

> 容量：边能承载的最大流量.

Exactly one node with no incoming (outgoing) edges, called the **source** s (**sink** t).

> 源点 s 和汇点 t，即流量的起点和终点.

Flow $f : V \times V \to R$ that satisfies

- Capacity constraint: $f(u, v) \leq c(u, v),\ \forall\ u, v \in V.$
- Skew symmetry: $f(u, v) = -f(v, u).$
- Flow conservation: $\sum_{v \in V} f(u, v) = 0,\ \forall\ u \in V - \{s, t\}.$

  > 除了源点和汇点，所有点流量守恒（流入等于流出，净流出为 0）

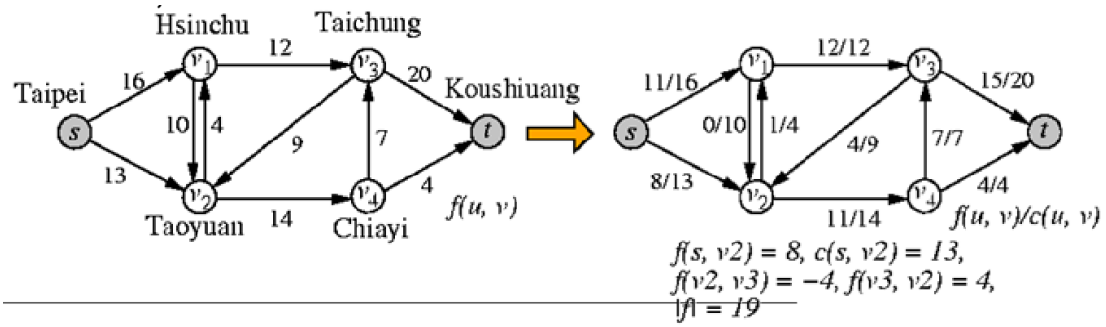
Value of a flow f. $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$, where $f(u, v)$ is the net flow from u to v.

> 即源点流出/汇点汇入的总流量.

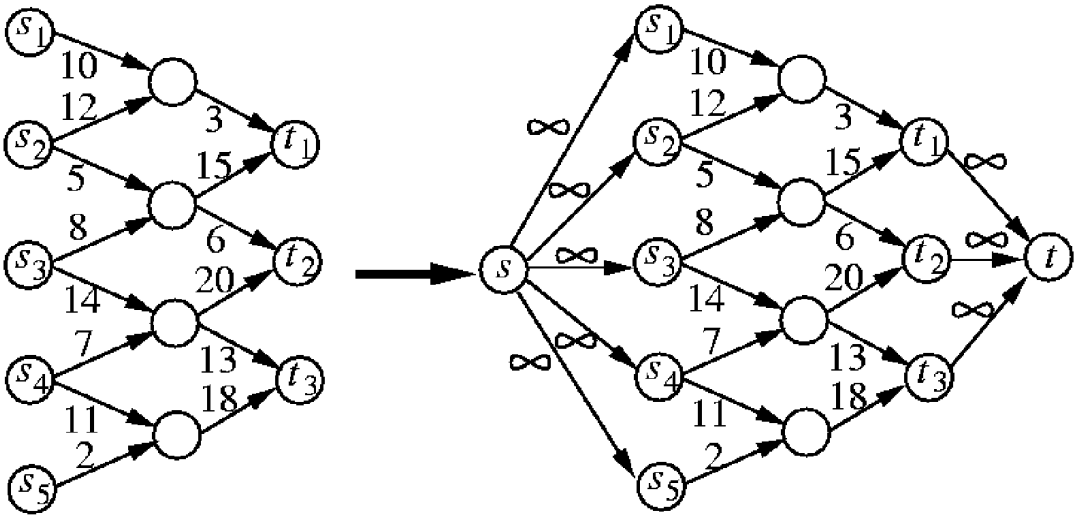
最大流问题

> The maximum flow problem

Given a flow network G with source s and sink t, find a flow of maximum value from s to t.



首先，多源多汇最大流问题可以简化为单源单汇：



Given a flow network with sources $S = \{s_1,\ s_2,\ \ldots,\ s_m\}$ and sinks $T = \{t_1,\ t_2,\ \ldots,\ t_n\}$, introduce a **supersource** s and edges $(s, s_i),\ i = 1,\ 2,\ \ldots,\ m$, with capacity $c(s, s_i) = \infty$ and a **supersink** t and edges $(t_i, t),\ i = 1,\ 2,\ \ldots,\ n$, with capacity $c(t_i, t) = \infty$.

然后，引入一些流的定义和性质

If $X, Y \subseteq V$, $f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$.

给定 $G = (V, E)$，f: flow in G

① $\forall\ X \subseteq V,\ f(X, X) = 0$.

② $\forall\ X, Y \subseteq V,\ f(X, Y) = -f(Y, X)$.

③ $\forall\ X, Y, Z \subseteq V$ with $X \cap Y = \emptyset$,

- $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ and
- $f(Z,\ X \cup Y) = f(Z, X) + f(Z, Y)$.

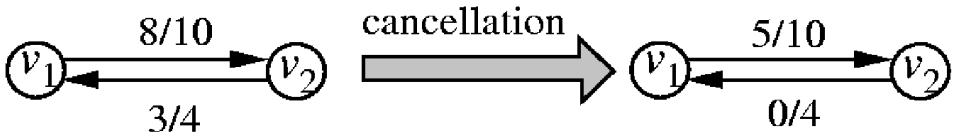
Value of flow = net flow into the sink:

$|f| = f(V, t)$.

$$
\begin{aligned}
|f| &= f(s, V) \\
&= f(V, V) - f(V - s, V) \\
&= f(V, V - s) \\
&= f(V, t) + f(V, V - s - t) \\
&= f(V, t)
\end{aligned}
$$

> 运用了流量守恒.

Cancellating flow going in opposite direction:



Basic Ford-Fulkerson Method

Ford-Fulkerson-Method(G, s, t)

1. Initialize flow f to 0
2. while there exists an augmenting path p do
3. Augment flow f along p
4. return f

Augmenting path: A path from s to t along which we can push more flow.

> 增广路径.

Need to construct a $residual\ network$ to find an augmenting path.

> 残余网络.

残余容量

Residual capacity of edge (u, v), $c_f(u, v)$: Amount of $additional$ net flow that can be pushed from u to v before exceeding $c(u, v)$:

$$c_f(u, v) = c(u, v) - f(u, v)$$

$G_f = (V, E_f)$: residual network of $G = (V, E)$ induced by f, where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$.

> 已经填满的就不用引入了.

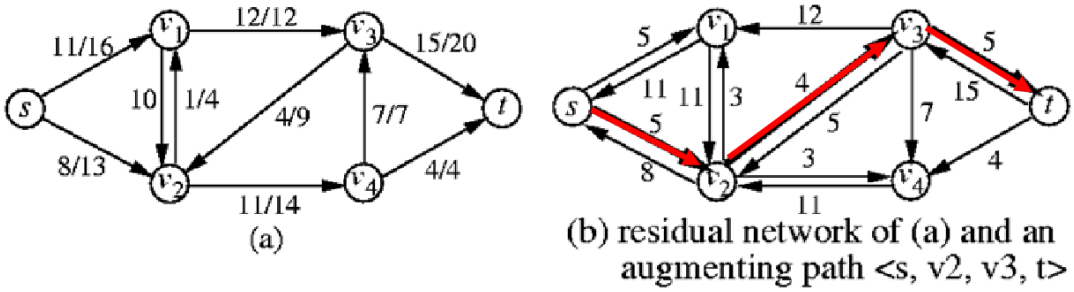
The residual network contains residual edges that can admit a positive net flow ($|E_f| \leq 2|E|$)

> 残余网络中，每条原始边 (u, v) 可能分解成正向边和反向边.
>
> 正向边：表示当前剩余的最大容量，$c_f(u, v) = c(u, v) - f(u, v)$
>
> 反向边：表示可以回退的最大流量，$c_f(v, u) = f(u, v)$

Let f and f' be flows in G and G_f, respectively. The **flow sum** $f + f' : V \times V \to R$:

$$(f + f')(u, v) = f(u, v) + f'(u, v)$$

is a flow in G with value $|f + f'| = |f| + |f'|$.



An **augmenting path** p is a simple path from s to t in the residual network G_f.

- $(u, v) \in E$ on p in the forward direction (a forward edge), $f(u, v) < c(u, v)$.
- $(u, v) \in E$ on p in the reverse direction (a backward edge), $f(u, v) > 0$

> 反向边：之前流过一些，现在可以回退.

Residual capacity of p, $c_f(p)$: Maximum amount of net flow that can be pushed along the augmenting path p, i.e.,

$c_f(p) = min\{c_f(u, v) : (u, v)\ is\ in\ p\}$.

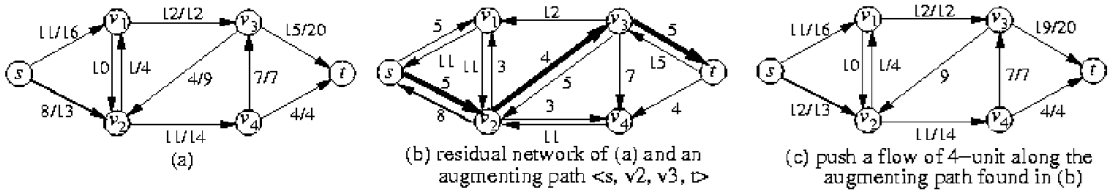
> 增广路径的容量取决于最小的一条，即最细的水管.

Let p be an augmenting path in G_f. Define $f_p : V \times V \to R$ by

$$f_p(u, v) = \begin{cases} c_f(p), & \text{if (u,v) is on p,} \\ -c_f(p), & \text{if (v,u) is on p,} \\ 0, & otherwise. \end{cases}$$

> 翻译：如果增广路径使用了反向边，就要回退流量. 如果使用正向边就增加流量.

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.



流网络的割

A cut (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

> 此外，割 (S, T) 将 V 分割成两个不相交子集，即 $S \cup T = V$ 且 $S \cap T = \emptyset$.

Capacity of a cut: $c(S, T) = \sum_{(u,v) \in E,\ u \in S,\ v \in T} c(u, v)$

> 只考虑 S 到 T（正向边）容量. 不考虑反向边.

$f(S, T) = |f| \le c(S, T)$, where $f(S, T)$ is net flow across the cut (S, T).

> $f(S, T)$ 是实际净流量.
>
> 割的容量表示割能阻断的流量上限（即从 S 流向 T 的最大可能流量）.
>
> 不同的割都对流量上限有约束作用. 实际流量要满足所有割的限制，因此有了最大流最小割定理.

最大流最小割定理

> Max-flow min-cut theorem:

最大流量（从源点 s 到汇点 t）等于网络中任意割的最小容量.

The following conditions are equivalent

① f is a max-flow.

② G_f contains no augmenting path.

③ $|f| = c(S, T)$ for some cut (S, T).

8.4.6 最大二分图匹配

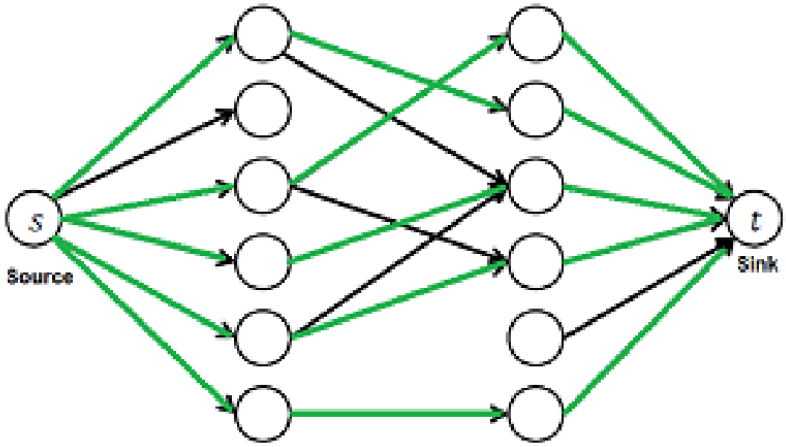
> Maximum Bipartite Matching Problems

① Ford-Fulkerson 算法

> Ford-Fulkerson Algorithm

最大二分图匹配问题可以转化为一个**最大流问题**，具体步骤如下：

1. 在二分图 G 中添加一个**源点 s** 和一个**汇点 t**.
2. 对于 U 中的每个顶点 u，添加一条从 s 到 u 的边，容量为 1.
3. 对于 V 中的每个顶点 v，添加一条从 v 到 t 的边，容量为 1.
4. 对于 U 和 V 中相连的每条边 (u,v)，添加容量为 1 的边.
5. 使用最大流算法（如 Ford-Fulkerson 算法或 Edmonds-Karp 算法）计算从 s 到 t 的最大流，最大流的值即为最大匹配的大小.



8.4.7 环检测

> Cycle Detection

① *深度优先搜索

> Depth-First Search

使用深度优先搜索（DFS）来检测环.

在遍历过程中，如果访问到一个已经访问过的节点，且该节点不是当前节点的父节点，则存在环.

8.4.8 图着色

> Graph Coloring

①②③④

Labs

Lab 1 链表

Q1 链表实现

Q2

Q3

Q4

Lab 2 排序算法

Q1 归并排序

Q2 插入排序

Q3 快速排序

Q4

Lab 3 栈和队列

Q1 队列：循环数组实现

Q2 栈：链表实现

Q3

Q4

Lab 4 哈希

Q1 分离链接法

Q2 开放寻址法

Q3

Q4

Lab 5 树 1

Q1 普通树

Q2 二叉树

Q3

Q4 平衡二叉树

Lab 6 树 2

Q1 二叉搜索树

Q2 二叉表达式树

Q3

Q4

Lab 7 树 3

Q1 最大堆

```c
#include"lab7-q1.h"
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<limits.h>


/* Init an empty heap with a given size */
MaxHeap* maxheap_init(int maxsize) {
    // Your code here
    MaxHeap *H = malloc(sizeof(MaxHeap));
    H->maxsize = maxsize;
    H->count = 0;
    H->data = malloc(sizeof(Node)*(maxsize+1));
    memset(H->data, 0, sizeof(Node)*(maxsize+1));
    return H;

}

/* Heaplify a max-heap based on the input array S with size S_size */
/* The root of heap is stored at H->node[1] */
/* Return the max-heap based on the input S */
MaxHeap *maxheap_heaplify(Node* S, int S_size) {
    // Your code here
    int i, index, swap, other;
    Node temp;
    MaxHeap *H = maxheap_init(S_size);
    for(i=0; i<S_size; i++){
        H->data[i+1].key = S[i].key;
        sprintf(H->data[i+1].value, "%s\0", S[i].value);
    }
    H->count = S_size;
    for(i=S_size/2; i>0; i--){
        temp.key = H->data[i].key;
        sprintf(temp.value, "%s\0", H->data[i].value);
        for(index = i; 1; index = swap){
            swap = (index << 1);
            if(swap > H->count) break;
            other = (swap | 1);
            if(other <= H->count && H->data[other].key > H->data[swap].key) swap = other;
            if(temp.key > H->data[swap].key) break;
            H->data[index].key = H->data[swap].key;
            sprintf(H->data[index].value, "%s\0", H->data[swap].value);
        }
        if(index != i){
            H->data[index].key = temp.key;
            sprintf(H->data[index].value, "%s\0", temp.value);
        }

    }
```

```
    return H;
}

/* Insert the (key,value) pair into the max-heap H */
/* If key exist in the heap, update the value */
/* If key does not exist in the heap and heap is not full, */
/* perform the insertion of a max-heap */
/* If key does not exist in the heap but heap is full, do nothing */
/* Return 1 if insertion is successful */
/* Return 2 if update of value is successful */
/* Return 0 if heap is full and insertion cannot be proceeded */
int maxheap_insert(MaxHeap *H, int key, char* value) {
    // Your code here
    int index, parent;
    for(index = 1; index <= H->count; index++){
        if(H->data[index].key == key){
            sprintf(H->data[index].value, "%s\0", value);
            return 2;
        }
    }
    if(H->count == H->maxsize) return 0;
    H->count++;
    for(index = H->count; index >1; index = parent){
        parent = (index >> 1);
        if(H->data[parent].key > key) break;
        H->data[index].key = H->data[parent].key;
        sprintf(H->data[index].value, "%s\0", H->data[parent].value);
    }
    H->data[index].key = key;
    sprintf(H->data[index].value, "%s\0", value);
    return 1;

}

/* Delete root of the max-heap */
/* Update the heap accordingly */
/* Return the (key,value) of the deleted root */
/* Return NULL if the heap is empty */
Node *maxheap_delete(MaxHeap *H) {
    // Your code here
    int index, swap, other;
    Node temp;
    Node *result;
    if(H->count==0) return NULL;

    temp.key = H->data[H->count].key;
    sprintf(temp.value, "%s\0", H->data[H->count--].value);

    result=malloc(sizeof(Node));
    result->key = H->data[1].key;
    sprintf(result->value, "%s\0", H->data[1].value);

    for(index = 1; 1; index = swap){
        swap = (index << 1);
        if(swap > H->count) break;
        other = (swap|1);
        if(other <= H->count && H->data[other].key > H->data[swap].key) swap = other;
        if(temp.key >H->data[swap].key) break;

        H->data[index].key = H->data[swap].key;
        sprintf(H->data[index].value, "%s\0", H->data[swap].value);

    }
    H->data[index].key = temp.key;
    sprintf(H->data[index].value, "%s\0", temp.value);
    return result;

}
```

```c
/* Free the max-heap */
/* Return NULL pointer */
MaxHeap *maxheap_free(MaxHeap *H) {
    if (H == NULL) return NULL;
    free(H->data);
    free(H);
    return NULL;
}
```

Q2 并查集

Q3 堆排序

Q4

Lab 8 图表示

Q1 邻接表

Q2 邻接矩阵

Q3 边列表

Q4 关联矩阵

Lab 9 图算法 1

Q1 Dijkstra 算法

Q2

Q3

Q4

Lab 10 图算法 2

Q1 最小生成树

> Minimum Spanning Tree

Q2 Bellman-Ford 算法

Q3

Q4

Assignment

Assignment 1

1. Write the big O of the following expressions in terms of n.

a) $\sum_{i=1}^{n} i$ (6%)
b) $\sum_{i=1}^{n} i^2$ (6%)
c) $\sum_{i=1}^{n} 4^i$ (6%)
d) $\frac{\log(n^2+1)}{\log(n)}$ (6%)
e) $\sum_{i=1}^{n} \frac{1}{i}$ (You don't need to have a closed-form solution for (e).) (6%)

solution:

a. O(n)

b. O(n)

c. O(n^2)

d. O(1)

e. O(n)

2. Let f(n), g(n) and h(n) be functions of integer n. Prove that if f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n)). (Hint: Use the definition of big-O.) (10%)

$f(n) \leq c_1 g(n), \forall n > n_{01}$

$g(n) \leq c_2 h(n), \forall n > n_{02}$

$f(n) \leq c_1 c_2 h(n), \forall n > max(n_{01}, n_{02})$

so $f(n) = O(h(n))$

3.

a) Let f(n) be a function of positive integer n. f(n + 1) ≥ f(n), f(1) = 1 and f(n) ≤ 1+f(⌈n/2⌉). ⌈x⌉ is the ceiling function. Prove f(n) = O(log n). (Hint: It is easy when n is a power of 2. What if n is not a power of 2?) (10%)

b) Let f(n) be a function of positive integer n. f(n + 1) ≥ f(n), f(1) = 1 and f(n) ≤ n + f(⌈n/99⌉). Prove f(n) = O(n log n). (Comment 1: The result can be further bounded by O(n). However, it is fine if you can prove it is O(n log n).) (10%)

(Comment 2: Some ideas in Question 2 can be found in this question.)

4. You are given a linked list A: 11→ 8→ 6→ 19→ 32→ NULL. Do the following operations on the linked list and show the result after each step.

a) What is the successor of the third term in A? (2%)
b) Delete the third term and show the new linked list A. *Explain briefly how you perform delete. (Hint: you have to perform on links between nodes.) (2%)*
c) What is the third term of A? (2%)
d) Change the head of the list to the third term of A*. Explain briefly how you perform head change and show the new linked list A**. (2%)**
e) Get the length of A. (2%)

5. Let S1 and S2 be two disjoint sets with discrete elements. The operation UNION takes S1 and S2 as input, and it returns a structure of set S = S1 ∪ S2 consisting of all the elements of S1 and S2. Show how to support UNION in O(1) time using a suitable list data structure. (Hint: You might want to use head/tail pointers in this question.) (10%)

6. This problem relates to the counting of inversions. You can write your codes or describe your algorithm in words. You need to explain that your algorithm can meet the time complexity requirement.

a) Consider an array A of n integers. Let e and e' be two integers in A such that e is positioned before e' in the array. We call the pair (e, e') an inversion in A if e > e'. Design an algorithm to report the number of inversions in A in O(n^2) time. (10%)

b) Can you design an algorithm to report the number of inversions in A in O(n log n) time? (Hint: try to follow the style and genre of merge sort, i.e., divide-and-conquer.) (10%)

Assignment 2

Assignment 3

Assignment 4

Final

2023

1. Explain the difference between arrays and linked lists in terms of memory allocation and element access.

特性	数组	链表
内存分配	连续的内存块 Contiguous block of memory	非连续，节点分散在内存中 Non-contiguous, scattered nodes
元素访问	随机访问，时间复杂度为 O(1) Random access in O(1) time	顺序访问，时间复杂度为 O(n) Sequential access, O(n) time

> 链表使用指针进行顺序访问.