算法总结

# 基础算法

## 优先队列（堆）

优先队列是一种抽象的数据结构，其与队列类似，只是其出对的顺序于优先级有关，这种抽象的数据结构有不同的事项方式；

### 数组实现

每次插入的时候，需要移动元素，将其插入到合适的位置；

### 单链表实现

每次插入的时候，将其插入到合适的位置（链表不需要移动元素）；

### 堆实现（最小堆、最大堆）

堆得实现，能在新增、删除元素时时间度log(n)的情形下实现出队操作（于堆排序的算法类似）；

新增元素的时候需要上移，删除元素的时候需要交换首元素，然后进行下移；

|  |
| --- |
| typedef struct {  int priority;  int value;  } priority\_element;  typedef struct \_priority\_queue {  int max;  int cnt;  priority\_element \*element;  } priority\_queue;  static void swap(priority\_element \*a, priority\_element \*b)  {  priority\_element tmp = \*a;  \*a = \*b;  \*b = tmp;  }  static void adjust\_heap\_down(priority\_element element[], int cnt, int idx)  {  priority\_element tmp = element[idx];  while (idx < cnt) {  int left = 2 \* idx + 1;  int right = 2 \* idx + 2;  if (left >= cnt) {  break;  }  if (right < cnt && element[right].priority < element[left].priority) {  left = right;  }  if (element[left].priority < tmp.priority) {  element[idx] = element[left];  idx = left;  continue;  }  break;  }  element[idx] = tmp;  }  static void adjust\_heap\_up(priority\_element element[], int cnt, int idx)  {  priority\_element tmp = element[idx];  while (idx > 0) {  int parent = (idx - 1) / 2;  if (parent < 0) {  break;  }  if (element[parent].priority > tmp.priority) {  element[idx] = element[parent];  idx = parent;  continue;  }  break;  }  element[idx] = tmp;  }  static void heapify(priority\_element element[], int cnt)  {  for (int i = cnt - 1; i >= 0; i--) {  adjust\_heap\_down(element, cnt, i);  }  }  void\* init\_priority\_queue(int max)  {  priority\_queue \*queue = (priority\_queue\*)*malloc*(sizeof(priority\_queue));  queue->*max* = max;  queue->cnt = 0;  queue->element = (priority\_element\*)*malloc*(sizeof(priority\_element) \* max);  return queue;  }  void free\_priority\_queue(void \*obj)  {  priority\_queue \*queue = (priority\_queue\*)obj;  *free*(queue->element);  *free*(queue);  }  void put\_priority\_queue(void \*obj, int val, int priority)  {  priority\_queue \*queue = (priority\_queue\*)obj;  queue->element[queue->cnt].priority = priority;  queue->element[queue->cnt++].value = val;  adjust\_heap\_up(queue->element, queue->cnt, queue->cnt - 1);  }  int get\_priority\_queue(void \*obj)  {  int ret = -1;  priority\_queue \*queue = (priority\_queue\*)obj;  ret = queue->element[0].value;  swap(&queue->element[0], &queue->element[--queue->cnt]);  adjust\_heap\_down(queue->element, queue->cnt, 0);  return ret;  } |

## 递归（剪枝）

## 全排列（回溯）

### 全排列

### 全排列去重

### 全排列字典顺序

## 位运算模拟加法、减法

## 素数的计算

## 大整数四则运算（加、减、乘、除）

## 逆波兰式

算法关键点描述：

* 使用两个栈来计算，一个存储数据栈，一个存储操作符；
* 遇到数字直接压栈数据栈；
* 遇到‘(’压栈操作符栈，遇到‘)’弹出操作符直到遇到‘（’；
* 操作符时，弹出比自己优先级高或等的操作符，直到‘（’，即，加减时弹出所有，乘除时弹出乘除；
* 弹出的过程可以将数据进行计算；
* 特别注意处理特殊情况，+、-可做运算符和符号位，如：3+2；-2 + 3；-2 +（+2 + （-2 + 4））

|  |
| --- |
| void caltwonum(int nstack[], int ntop, char op)  {  switch (op) {  case '+':  nstack[ntop - 2] += nstack[ntop - 1];  break;  case '-':  nstack[ntop - 2] -= nstack[ntop - 1];  break;  case '\*':  nstack[ntop - 2] \*= nstack[ntop - 1];  break;  case '/':  nstack[ntop - 2] /= nstack[ntop - 1];  break;  }  }  int calculate(char \* s)  {  int len = *strlen*(s);  int \*nstak = (int\*)*malloc*(sizeof(int) \* len);  char \*pstack = (char\*)*malloc*(sizeof(char) \* len);  int ntop = 0;  int ptop = 0;  bool signOK = true;  bool numOK = false;  int num = 0;  for (int i = 0; i < len; i++) {  if (s[i] == ' ') {  continue;  }  if (*isdigit*(s[i])) { // 计算数字  num = num \* 10 + (s[i] - '0');  numOK = true;  signOK = false;  continue;  }  if ((s[i] == '-' || s[i] == '+') && signOK == true) {  signOK = false; // 处理特殊场景添加0 -3，-(-2),(+2)  numOK = true;  num = 0;  }  if (numOK == true) {  nstak[ntop++] = num;  num = 0;  numOK = false;  }  switch (s[i]) {  case '+':  case '-':  while (ptop && pstack[ptop - 1] != '(') {  caltwonum(nstak, ntop, pstack[ptop - 1]);  ptop--;  ntop--;  }  pstack[ptop++] = s[i];  break;  case '\*':  case '/':  while (ptop && pstack[ptop - 1] != '(' &&  pstack[ptop - 1] != '+' && pstack[ptop - 1] != '-') {  caltwonum(nstak, ntop, pstack[ptop - 1]);  ptop--;  ntop--;  }  pstack[ptop++] = s[i];  break;  case '(':  pstack[ptop++] = s[i];  signOK = true;  break;  case ')':  while (ptop && pstack[ptop - 1] != '(') {  caltwonum(nstak, ntop, pstack[ptop - 1]);  ptop--;  ntop--;  }  ptop--;  break;  }  }  if (numOK) {  nstak[ntop++] = num;  }  while (ptop) {  caltwonum(nstak, ntop, pstack[ptop - 1]);  ptop--;  ntop--;  }  int ret = nstak[0];  *free*(nstak);  *free*(pstack);  return ret;  } |

## 优先队列

## 二分搜索

## 前缀和、差分数组

## 二叉树

## 滑动窗口

## 单调栈

## 图（有向图、无向图）

### 图的表示

#### 邻接矩阵

使用二维数组进行存储，元素的值为权值，若不可达，则置为无穷大；

如果顶点多，而边比较少是那么久比较浪费空间；

#### 邻接表

将每个顶点能到的边使用链表进行表示，节约空间；

### 图的遍历

#### DFS

#### BFS

### 图的最短路径问题

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | [743. 网络延迟时间](https://leetcode-cn.com/problems/network-delay-time/) |  |  |
|  |  |  |  |
|  |  |  |  |

#### Dijkstra算法（迪杰斯特拉算法）——单源最短路径

约束：

1. 用于有向图（或无向图），其中权值为正值；
2. 单源最短路径，求出的是一个点到其他所有点的最短距离；
3. 查找当前的最短路径可以使用优先队列进行优化；

算法的思想，类似于BFS，没走一步，找到离起点最近的一个点，然后从这个点去更新其他点的距离；相当于每一个点都是从最近的位置到某一个点，然后从这些点中在找出最小的；

|  |
| --- |
| #define MIN(*a*, *b*) ((*a*) > (*b*) ? (*b*) : (*a*))  #define MAX(*a*, *b*) ((*a*) > (*b*) ? (*a*) : (*b*))  #define MAXNET 100  typedef struct tagNetDelay {  int index;  int delay;  struct tagNetDelay \*next;  } NetDelay;  void AddOneDelay(NetDelay netDelay[], int start, int end, int delay)  {  NetDelay \*pre = &netDelay[start - 1];  NetDelay \*head = pre->next;  NetDelay \*tmp = (NetDelay\*)*malloc*(sizeof(NetDelay));  tmp->delay = delay;  tmp->index = end - 1;  pre->next = tmp;  tmp->next = head;  }  int networkDelayTime(int\*\* times, int timesSize, int\* timesColSize, int n, int k)  {  NetDelay netDelay[MAXNET] = { 0 };  for (int i = 0; i < timesSize; i++) {  AddOneDelay( netDelay, times[i][0], times[i][1], times[i][2]);  }  int visit[MAXNET] = { 0 };  int dist[MAXNET];  for (int i = 0; i < n; i++) {  dist[i] = *INT\_MAX*;  }  dist[k - 1] = 0;  for (int i = 0; i < n; i++) {  int minIdx = -1;  int minDelay = *INT\_MAX*;  for (int i = 0; i < n; i++) {  if (visit[i] == 0 && dist[i] < minDelay) {  minDelay = dist[i];  minIdx = i;  }  }  visit[minIdx] = 1;  for (NetDelay \*head = netDelay[minIdx].*next*; head; head = head->*next*) {  if (visit[head->index] == 0) {  dist[head->index] = MIN(dist[head->index], head->delay + minDelay);  }  }  }  int ret = 0;  for (int i = 0; i < n; i++) {  ret = MAX(dist[i], ret);  }  return ret == *INT\_MAX* ? -1 : ret;  } |

其中查找最小的节点的方法可以使用优先队列的方法加速（最小堆）；

#### Floyd算法（佛洛依德算法算法）——单源最短路径

多源最短路径算法

利用了动态规划的思想；

|  |
| --- |
| #define MAX(*a*, *b*) ((*a*) > (*b*) ? (*a*) : (*b*))  #define MIN(*a*,*b*) ((*a*) > (*b*) ? (*b*) : (*a*))  int networkDelayTime(int\*\* times, int timesSize, int\* timesColSize, int n, int k)  {  int \*\*dist = (int\*\*)*malloc*(n \* sizeof(int\*));  for (int i = 0; i < n; i++) {  dist[i] = (int\*)*malloc*(n \* sizeof(int));  for (int j = 0; j < n; j++) {  dist[i][j] = *INT\_MAX*;  }  dist[i][i] = 0;  }  for (int i = 0; i < timesSize; i++) {  dist[times[i][0] - 1][times[i][1] - 1] = times[i][2];  }  for (int k = 0; k < n; k++) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (dist[i][k] != *INT\_MAX* && dist[k][j] != *INT\_MAX*) {  dist[i][j] = MIN(dist[i][k] + dist[k][j], dist[i][j]);  }  }  }  }  int ret = *INT\_MIN*;  for (int i = 0; i < n; i++) {  ret = MAX(ret, dist[k - 1][i]);  }  return ret == *INT\_MAX* ? -1 : ret;  } |

#### Bellman\_ford算法（单源路径）

## 排序算法



### 选择排序

### 冒泡排序

### 插入排序

### 堆排序

### 归并排序

### 快速排序

### Shell排序

### 桶排序