COMP9024 考点复习

UNSW COMP9024

Outline 如果哪部分内容有遗漏，望告知 admin@shangxue.com.au

- 6月12号_新增了
  8.dj会崩掉的情况
  1.树中找最大值
  3.KMP find LCS
  5.BFS 2-colorable
- 6月13号_修改了 KMPfindLCS的几个错误
- **Course_name** ：COMP9024 Data Structure and Algorithm ；
- **Lecturer** ：HUI WU
- **Author** ：ZESHI WU

---

1. 看代码写输出

- 看代码写输出
- 最长连续【上升 / 下降】序列

```
class Solution {
public:
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    int max(int a, int b){
        if (a > b) return a;
        else return b;
    }
    int longestIncreasingContinuousSubsequence(vector<int>& A) {
        // Write your code here
        int res = 0, up = 1, down = 1;
        int n = A.size();
        if (n <= 2) return n;
        for (int i = 1; i < n; i++){
            if (A[i] > A[i-1]){
                up += 1;
                down = 1;
            }
            else if (A[i] == A[i-1]){
                up += 1;
                down += 1;
            }
            else if (A[i] < A[i-1]){
                up = 1;
                down += 1;
```

```
                }
                if (up > res || down > res)
                    res = max(up, down);
            }
            return res;
        }
    };
```

2. 写算法最差的情况

- 4个小问，写算法最差的情况
- 其中两个是欧拉，二叉树搜索
- 欧拉回路？？

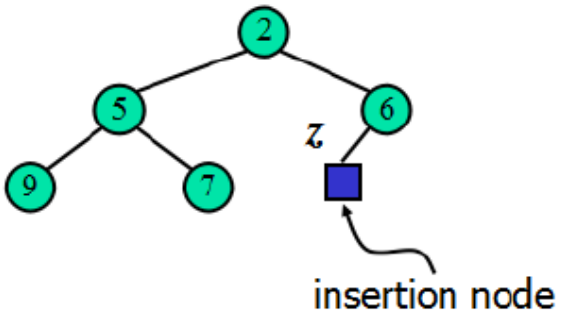
3. heap wk7

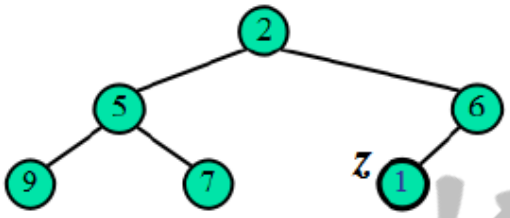
- heap 画出插入 / 删除一个数的过程 都是O(logn)

7.2.1 Insertion

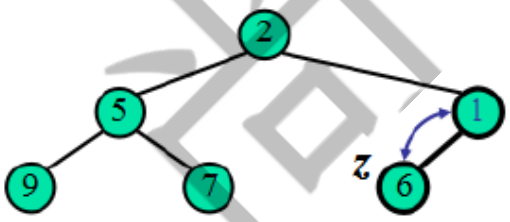
- 每次将新entry插入当前结尾结点的下一个位置, 并使用upheap()来更新du

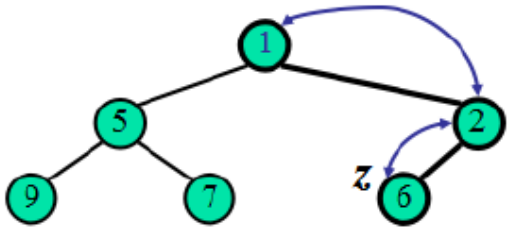
```
Algorithm Insert(
    find insert position Node z; //当前尾结点的下一个位置
    Store k at z;
    upheap();
}
```

- 

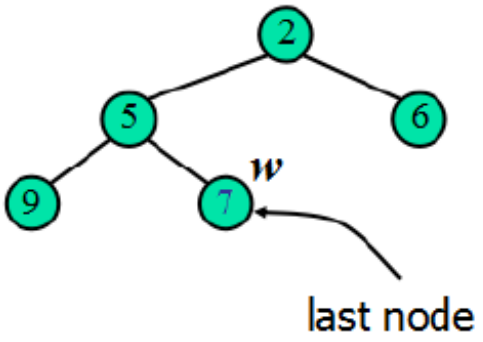


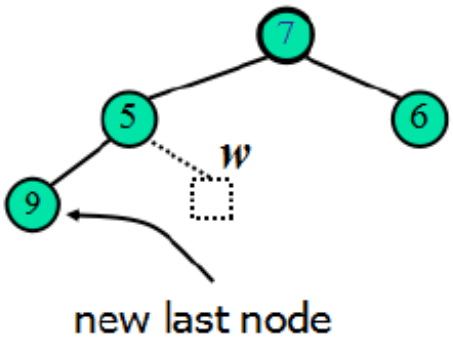
- upheap : 插入新entry后heap-order可能被破坏
  - 新插入结点与其父结点进行比较，直到满足heaporder或者新结点到达root;
  - upheap过程 $O(logn)$



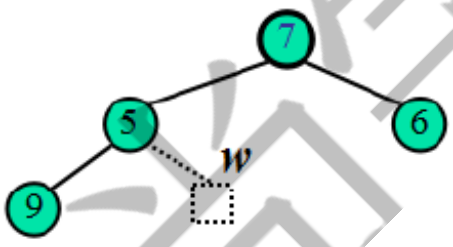


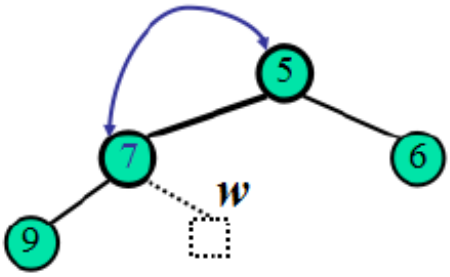
- Deletion
  - 每次删除的都是根结点: remove Min()
  - 返回root位置元素, 并将结尾元素替换到root ,之后使用downheap恢复堆排序
  - downheap比较耗时间, 总体复杂度: O(logn)





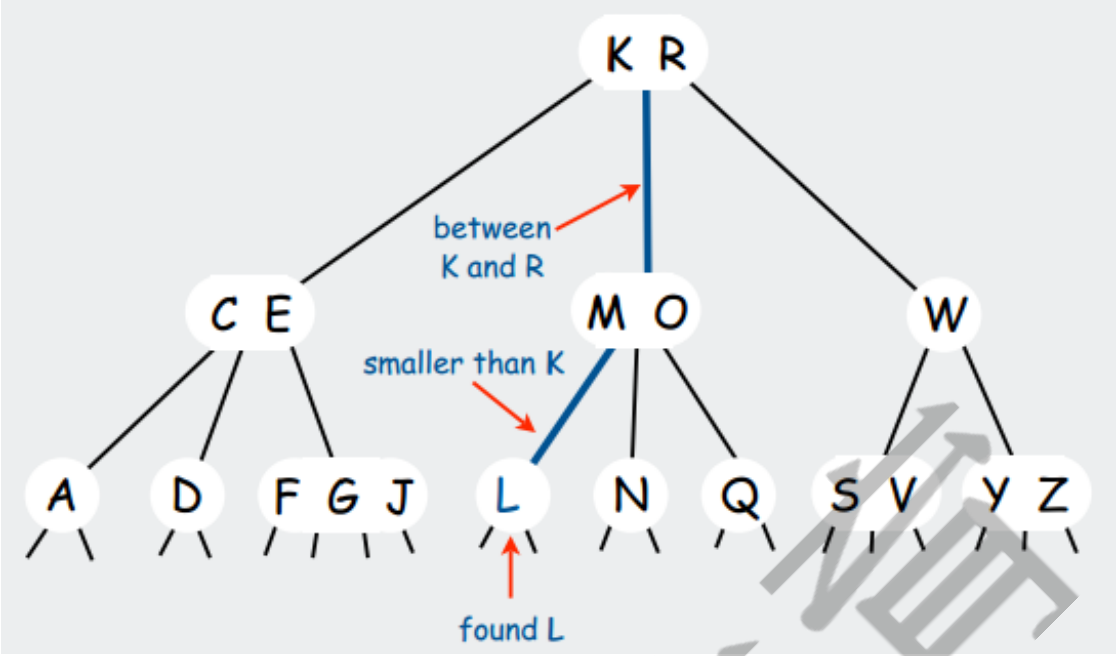
- Downheap : 结尾entry替换到root后，heaporder会被破坏
  - 从root开始，不断与子结点比较交换直到到达底层或是满足heap-order;
  - 优先交换子结点比较小的那个;
  - downheap过程 $O(logn)$;

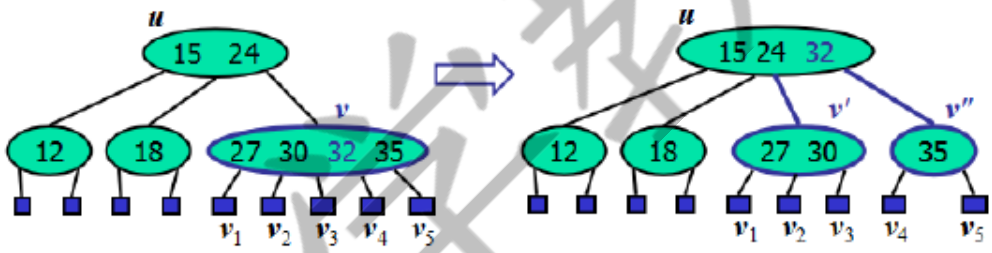





```
Algorithm remove{
    min = root.get Entry();
    root.set Entry(last Node);//将尾结点元素替换到root
    delete original position;
    downheap();
    return min;
}
```

4. (2，4) tree 画法

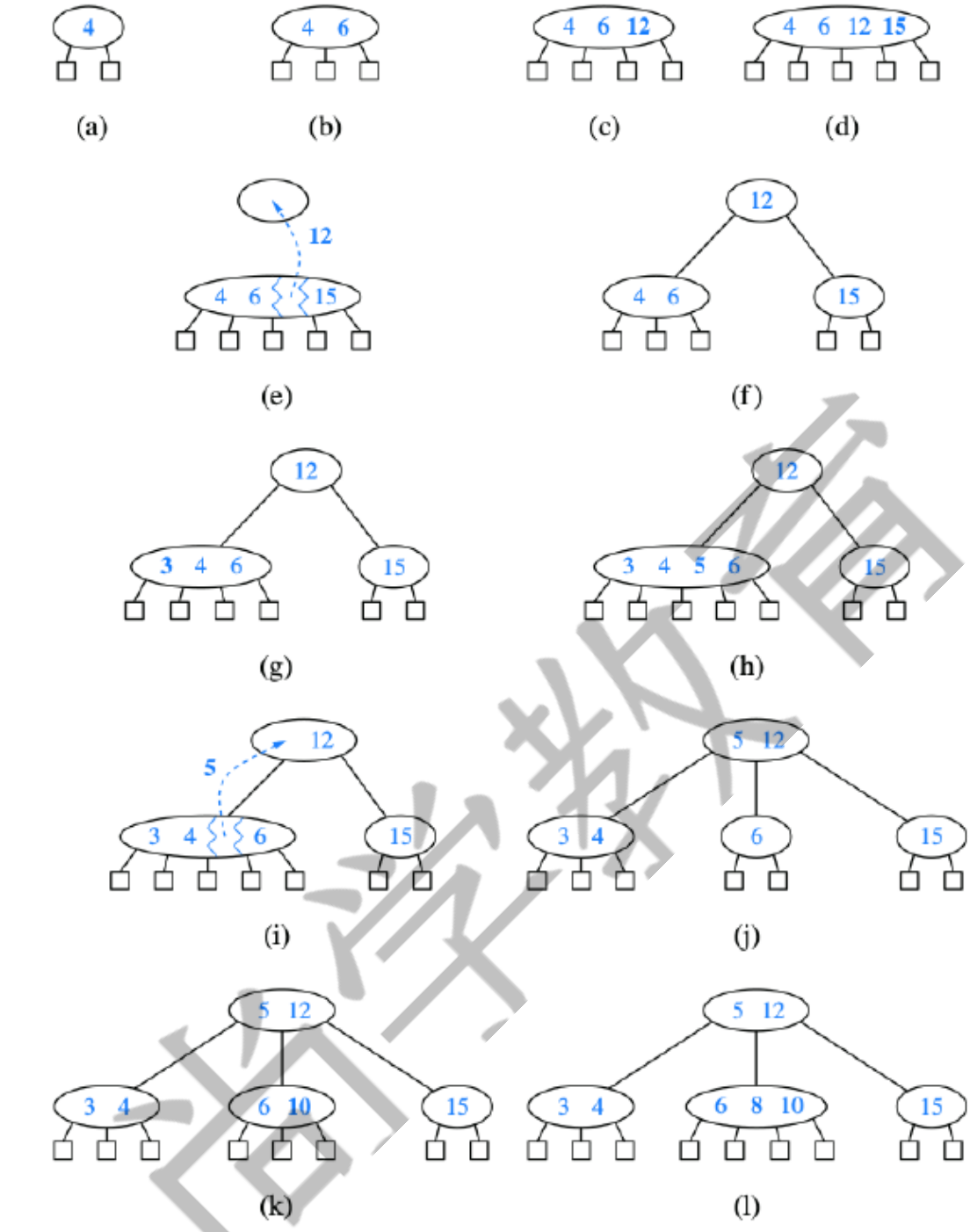
- 查找
  - 

- 插入节点
  - 6.4.2 Insertion
    - 插入后可能破坏规则(overflow), 需要进行split
    - 

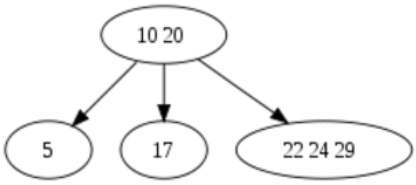
    - Insertion + split ***O(logn)***

```
Algorithm insert(k, o){
    search for key k to locate the insertion node v;
    add the new entry (k, o) at node v;
    while ( overflow(v) ){
        if (isRoot(v))
            create a new empty root above v;
        v = split(v);
    }
}
```

    - 更复杂的栗子:

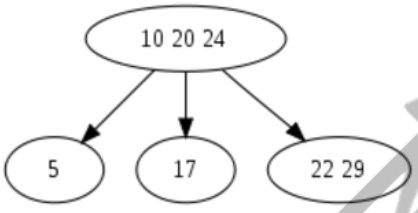
- 
4
4 6
4 6 12
4 6 12 15
(a)
(b)
(c)
(d)
12
4 6 15
(e)
12
4 6
15
(f)
12
3 4 6
15
(g)
12
3 4 5 6
15
(h)
12
5
3 4 6
15
(i)
5 12
3 4
6
15
(j)
5 12
3 4
6 10
15
(k)
5 12
3 4
6 8 10
15
(l)
- 不一样的解法

例如，现在要将值 "25" 插入到如下的 2-3-4 树中：



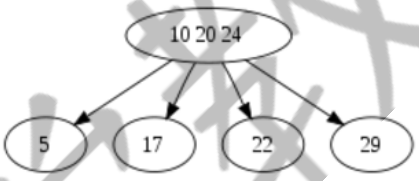
从根节点（10,20）开始查找，向子树查找，直到找到子节点（22,24,29）。因为区间（20,∞）包含 25。

节点（22,24,29）是一个 4 度节点，所以将其中间值 24 推到父节点中。



剩下的 3 度节点（22,29）将被分裂成一对 2 度节点，也就是（22）和（29）。新的父节点为（10,20,24）。

在下降到右侧子节点（29）。因为区间（24-29）包含 25。

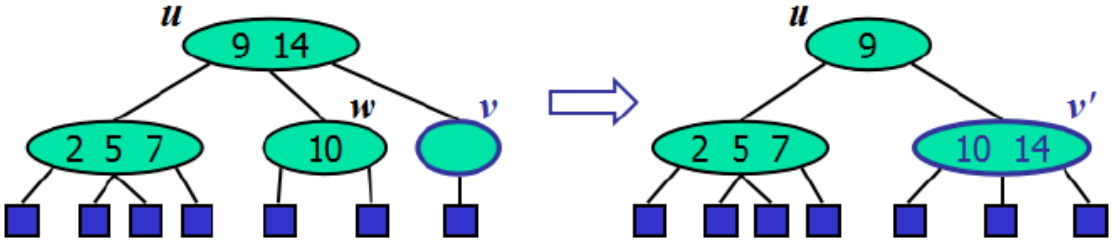


节点（29）没有左孩子。可将 25 直接插入到该节点中，插入完毕。

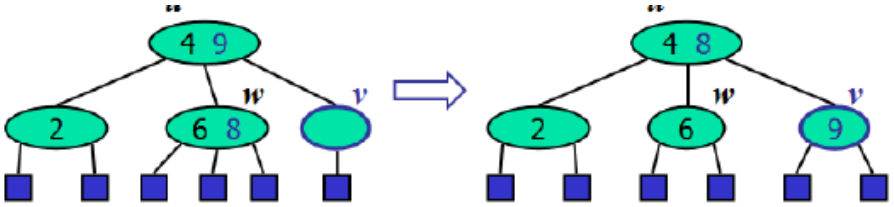
- 删除

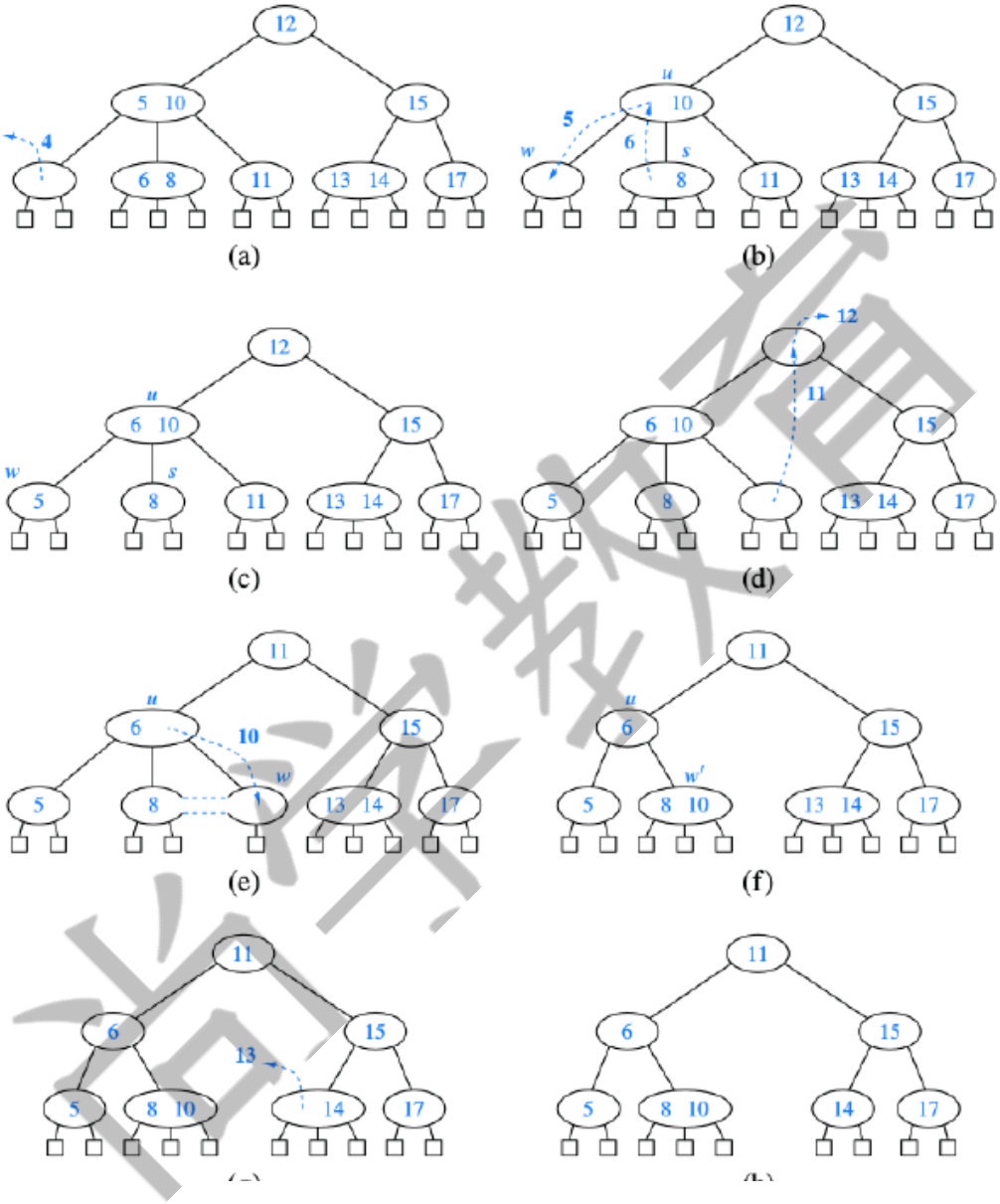
6.4.3 Deletion

- 删除操作: 用被删除点的successor(predecessor)替换其位置
- 删除后可能破坏规则(underflow)，需要进行fusion/transfer
- Fusion:



- Transfer:



- 更复杂的栗子:
  - 

5. 二叉树路径

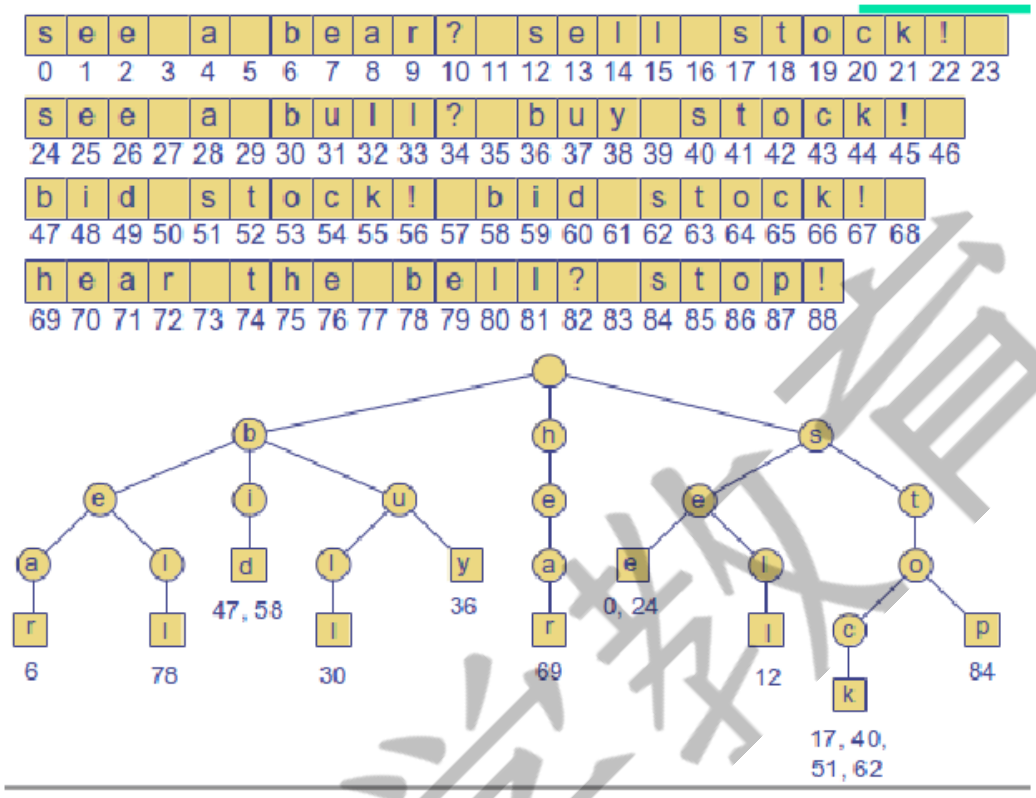
- preorder root，左，右
- postorder 左，右，root
- indorder 左，root，右
- BFS

```
10.  public class Breadth First Tree Traversal {
11.     Initialize queue Q storing root();
12.     while (!Q.is Empty()){
13.      p=Q.dequeue(); //p is the oldest in the queue
14.      visit(p);
15.      for each child in children(p){
16.         Q.enqueue(c);
17.         }
18.     }
19.  }
```

6. 画trie wk9

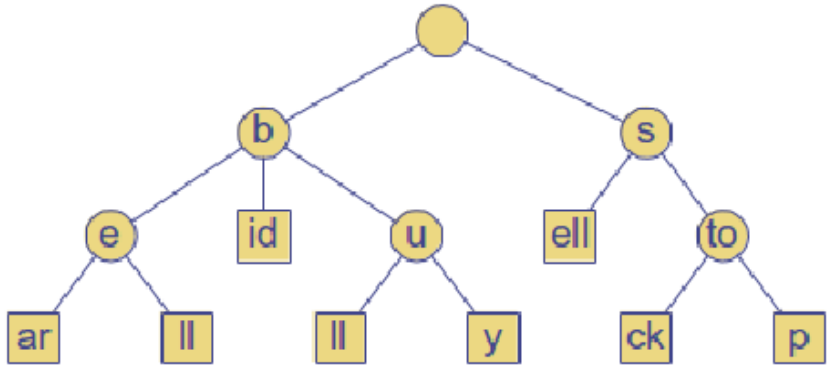
6.1 standard trie

- 对于总长度为n,存储了s个字符串的标准字典树T:
  - n : 总长度;
  - m : 每次操作动用字符串参数的数量;
  - d : 字母表的size;
  - 空间复杂度 O(n)
  - 搜索/插入/删除操作 O(m)
  - T的高度为s个字符串中最长的那个的长度;
  - 每个internal最多d个子结点;
  - T有s个external;
  - T的结点数量最多为n+1(所有单词不共享路径,再加上根结点);

- 使用字典树进行文本匹配
  - 将文本插入trie;
  - 每次操作始于搜索:先查询首字母,然后一步步查询单词;
  - 到达叶结点还能给出这个单词首字母的位置;
- 

6.2 compress trie

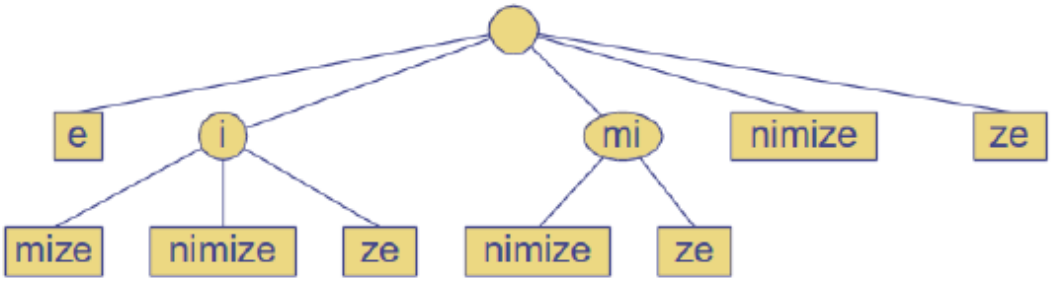
- 压缩多余结点:
  - 保证每个internal都有至少两个子结点
  - 若某个internal只有一个子结点, 进行压缩

- 空间复杂度 $O(s)$,其中S为array中的字符串数量, 和标准字典树相比优化了存储空间
- 

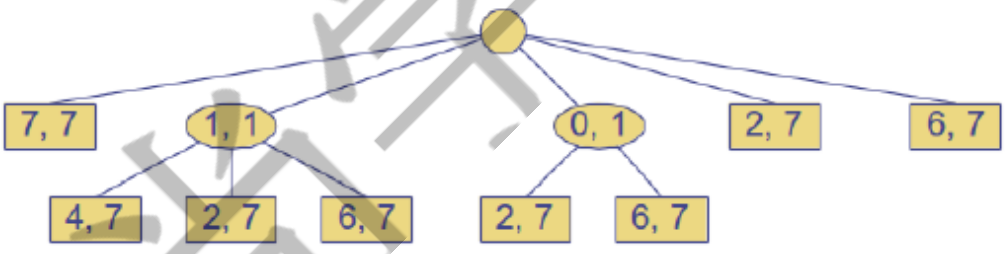
- 插入和删除

6.3 Suffie trie

- 对Suffix Tries的分析:
- n:字符串X的size;
- d:alphabet的size;
- m:Pattern的size;
- 空间复杂度 O（n）
  - 长度为n的串X的后缀总长度为n(n+1)/2
  - 若显式保存空间复杂度是平方级
  - 这里使用隐式保存(坐标), 优化了空间复杂度
- 字符匹配操作 O（dn）-》O(m)
- 构建字典树，但相对不容易构建, 如果是显式构建的话和空间复杂度一样都是平方级别



- 用坐标替换字符串：从头到尾,例如"nimize"=[2,7]



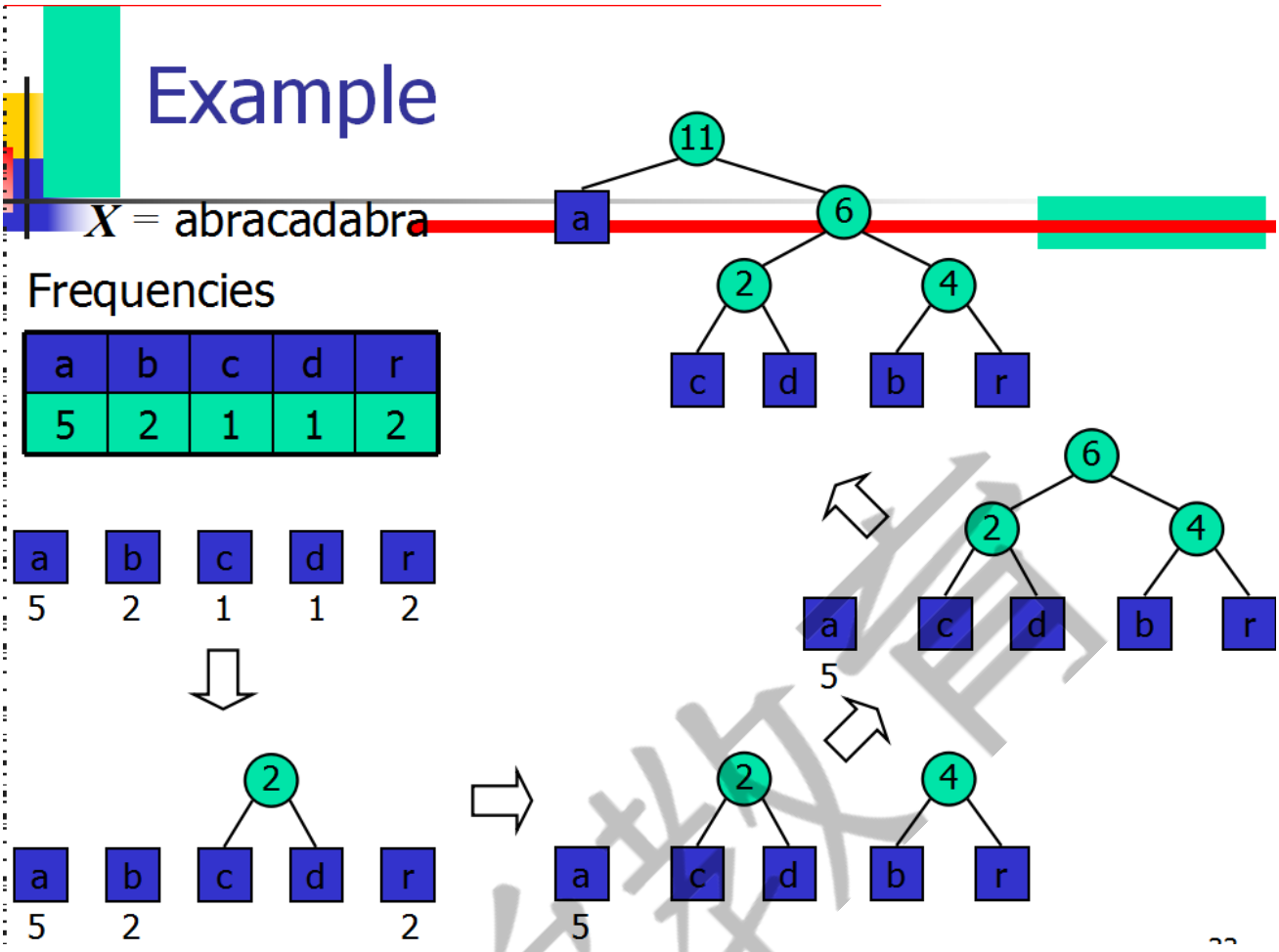

```
Algorithm suffix Trie Match
Input: Compact suffix trie T for a text X and pattern P;
Output: 若完全匹配，给出X中对应P开头的位置，否则返回-1;
{
    p=P.length;
    j=0;
    v=T.root();
    repeat{
        f=true;
        for (each child w of v){
            //已经匹配了j+1个字符
            i=start(w); //w的起始index(首字母)
```

```
            if(P[j]==T[i])
            {   //对子树w进行处理
                 x=end(w)-i+1;//end(w)->end index of w
                if(p<=x)
                {    //判断pattern是否是当前结点关键字的子字符串

                    if(P[j:j+p-1]==X[i:i+p-1]) return i-j;
                    else return -1;
                }
                else{
                // 判断当前节点关键字是否是pattern子字符串
                    if (P[j:j+x-1]==x[i:i+x-1]){
                        p=p-x //更新后缀长度
                        j=j+x;//更新后缀起始index
                        v=w;
                        f=false;
                        break for loop; //进行下一次repeat
                    }
                }
            }
        }
    }until (f= true || isExternal (v) ) //repat
    return -1
}


// ppt
Algorithm suffixTrieMatch(T, P)
  {   p = P.length;   j = 0; v = T.root();
      repeat
         { for  each child  w of v  do
              { // we have matched j + 1 char
                childTraversed=false; i = start(w);    // start(w) is the start index of w
                if ( P[j] = X[i] )    // process child w
                    {  childTraversed=true;
                       x = end(w) - i +1;   // end(w) is the end index of w
                       if  (  p ≤ x )
                        // suffix is shorter than or of the same length of the node label
                           { if  (  P[j:j+p-1] = X[i:i+p-1]  )   return i - j ;
                             else return "P is not a substring of X" ; }
```

```
                                    else // the pattern goes beyond the
substring stored at w
                                        { if  (  P[j:j+x-1] = X[i:i+x-1]  )
                                                {  p = p - x;  // update suffix
length
                                                   j = j + x;  // update suffix
start index
                                                   v = w;  break ; }
                                            else  return "P is not a substring o
f X" ; }}}}
        until childTraversed=false or T.isExternal(v);
        return "P is not a substring of X" ; }
```

7. 画huffman tree wk9

- 赫夫曼树的构建步骤如下：
  1、将给定的n个权值看做n棵只有根节点（无左右孩子）的二叉树，组成一个集合HT，每棵树的权值为该节点的权值。
  2、从集合HT中选出2棵权值最小的二叉树，组成一棵新的二叉树，其权值为这2棵二叉树的权值之和。
  3、将步骤2中选出的2棵二叉树从集合HT中删去，同时将步骤2中新得到的二叉树加入到集合HT中。
  4、重复步骤2和步骤3，直到集合HT中只含一棵树，这棵树便是赫夫曼树。

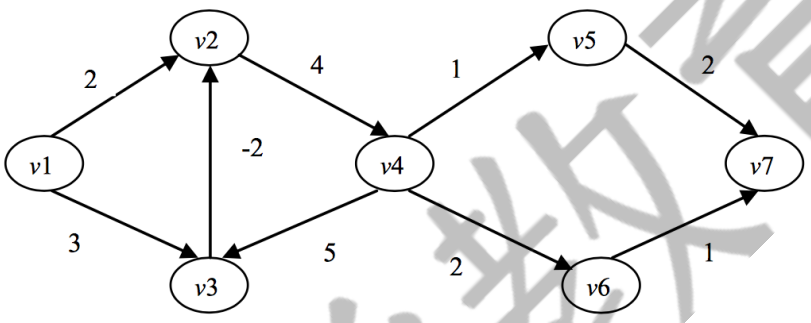



```
1.  Algorithm HuffmanEncoding(X)
2.      Input string X of size n
3.      Output optimal encoding trie for X  //最小化编码后的size
4.    {
5.      C = distinctCharacters(X);  //构建字母表
6.      computeFrequencies(C, X);   //计算字母表中每个元素的频率
7.
8.      Q = new empty heap;
9.      for all c in C
10.        { T = new single-node tree storing c;
11.          Q.insert(getFrequency(c), T);
12.         } //先将全部元素及其频率以键值对形式插入Heap
13.
14.     while   ( Q.size() > 1 )
15.         {    f1 = Q.minKey();
16.            T1 = Q.removeMin();
17.            f2 = Q.minKey();
18.            T2 = Q.removeMin();
19.             //每次移出并归并两个频率最低的键值对得到一个新键值对作为父结点(key为这两个键对key的和)
20.
```

```
                T = join(T1, T2);
                Q.insert(f1 + f2, T); //将新键值对再插入Q中

            }
        return Q.removeMin();
    }
```

8. dj算法

- dj算法会崩掉的情况homework 12 q2



Is there something going wrong? Explain.

Solution: Let's simulate the execution of Dijkstra's algorithm in the given graph with a negative weight:

Step 0: D($v1$) = 0, d($v2$) = d($v3$) = . . .= d($v7$) = +∞. cloud ={}, Q={$v1$, $v2$, …, $v7$}
Step 1: Remove $v1$ from Q. cloud ={$v1$}, Q={$v2$, $v3$, …, $v7$}, d($v2$) = 2, d($v3$) = 3, d($v4$) = d($v5$) = . . .= d($v7$) = +∞.
Step 2: Remove $v2$ from Q. cloud ={$v1$, $v2$}, Q={$v3$, $v4$, …, $v7$}, d($v3$) = 3, d($v4$) = 6, d($v5$) = d($v6$) = d($v7$) = +∞.
Step 3: Remove $v3$ from Q. cloud ={$v1$, $v2$, $v3$}, Q={$v4$, $v5$, …, $v7$}, d($v4$) = 6, d($v5$) = d($v6$) = d($v7$) = +∞.
Now Dijkstra's algorithm goes wrong here. In step 3, since $v2$ is already in the cloud, d($v4$) is not modified. As a result, the shortest path $v1$->$v3$->$v2$ is missed out. The shortest path $v1$->$v2$->$v4$->$v5$->$v7$ found by Dijkstra's algorithm is wrong. The real shortest path is $v1$-> $v3$->$v2$->$v4$->$v5$->$v7$.

- DJ 求最短路径

```
Algorithm DijkstraDistances(G, s)
    Input: A simple undirected weighted graph G with nonnegative edge
weights, and a distinguished vertex s.
```

```
    Output: A label D[u], for each vertex u, such that D[u] is the length of a shortest path from s to u in G.

   {  for each  v ∈ G  do
            if  ( v = s )
                 D[v] = 0;
            else
                 D[v] = +∞;
        Create a priority queue Q containing all the vertices of G using the D labels as keys;
   while  Q is not empty do
        {  u = Q.removeMin();
           for each  z ∈ Q such that  z is adjacent to u do
               if  ( D[u] + w((u,z)) < D[z] )  // relax edge e
                   {  D[z] = D[u] + w((u,z));
                        Change to D[z]  the key of vertex z in Q;
                     }
         }
   return the label D[u] of each vertex u;
}
```

- 有向图求加权最高
  - dj算法 相反

1. 求树中任意子树节点个数

```
//  先序遍历
Algorithm preOrder(v)
   { visit(v);
      if hasLeft(v)
          preorder(left(v));
      if hasRight(v)
          preorder(right(v));
   }
//  用以上先序遍历为例
Algorithm Solution (v)
Input tree
Ouput max value
{
    int max =0;

    S.push(v)
```

```
17.
18.      while(!S.isEmpty())
19.          v = S.pop();
20.          visit(v);
21.          max = max>v? max: V ;
22.          if hasLeft(v)
23.                S.push(left(v));
24.          if hasRight(v)
25.                S.push(right(v));
26.         }
27.       return max;
28.   }
29.
30.
31.   Algorithm inOrder(v)
32.     { if ( hasLeft (v) )
33.          inOrder(left (v));
34.       visit(v);
35.       if ( hasRight (v) )
36.           inOrder(right (v));
37.      }
38.
39.   //Algorithm inOrder(v)
40.     {
41.
42.
43.      while(!S.isEmpty() || v != null)
44.      {
45.          S.push(v)
46.
47.          if ( v != null )
48.              S.push(left(v));
49.              v= left(v);
50.          else
51.               {
52.                v= S.pop();
53.                visit(v)
54.                S.push(right(v));
55.               }
56.
57.      }
58.   }
59.
60.   Algorithm postOrder(v)
61.     { if ( hasLeft (v) )
```

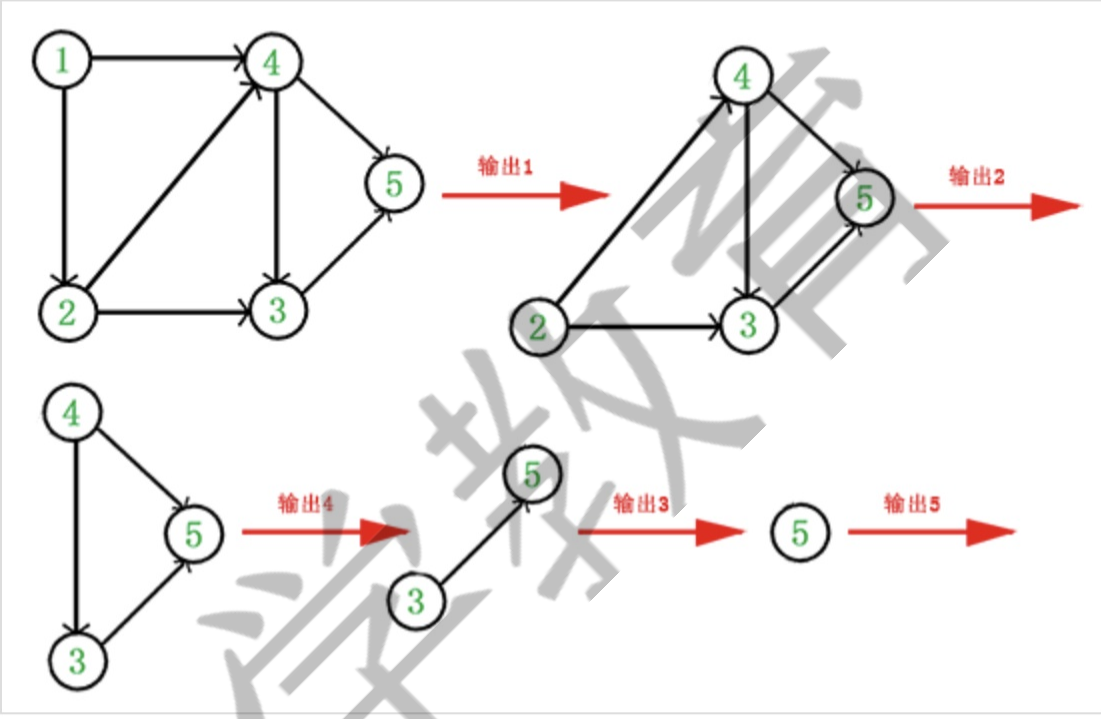
```
62.           postOrder(left (v));
63.       if ( hasRight (v) )
64.           postOrder(right (v));
65.        visit(v);
66.     }
67.
68.
69.  // 找last node
70.  {
71.      if v=null
72.          return null;
73.      if left(v) != null // v has a left child
74.      {
75.          v=left(v); // the rightmost descendant of left(v) is the IIP
     while right(v)!=null
76.              v=right(v);
77.          return v;
78.      }
79.      else
80.      {
81.          if v is root
82.              return null;
83.          while parent(v) != null
84.      {
85.          if right(parent(v))=v // v is the right child of its parent
86.          return parent (v); else
87.          v = parent(v);
88.      }
89.      return null ; // no IIP
90.  }
```

2. 求DAG最长路径 + 拓扑排序

2.2 拓扑排序

它是一个 DAG 图，那么如何写出它的拓扑排序呢？这里说一种比较常用的方法：

1. 从 DAG 图中选择一个 没有前驱（即入度为0）的顶点并输出。
2. 从图中删除该顶点和所有以它为起点的有向边。
3. 重复 1 和 2 直到当前的 DAG 图为空或**当前图中不存在无前驱的顶点为止**。后一种情况说明有向图在环。

- 

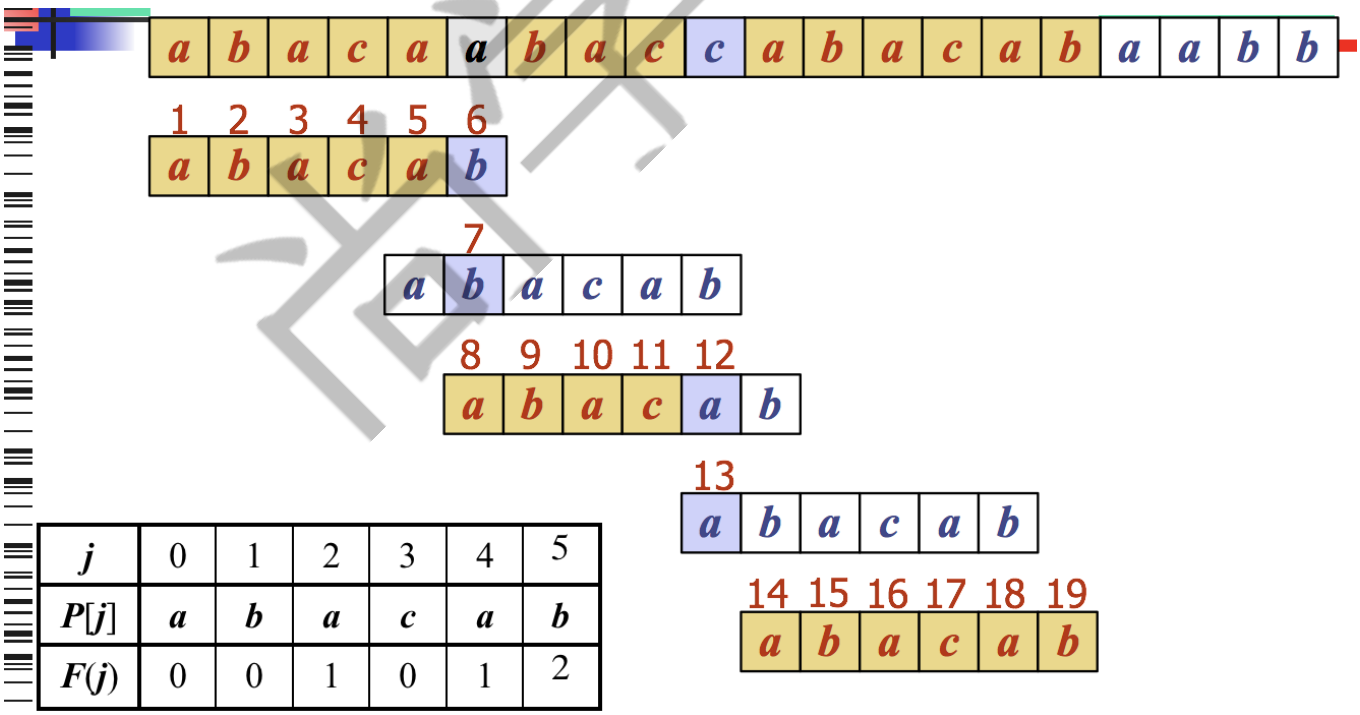
于是，得到拓扑排序后的结果是 { 1, 2, 4, 3, 5 }。

通常，一个有向无环图可以有**一个或多个**拓扑排序序列。

```
// part1
Algorithm topologicalDFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the vertices of G □      in the connected
component of v
 {  setLabel(v, VISITED);
    for all  e ∈ G.incidentEdges(v)
        if  ( getLabel(e) = UNEXPLORED )
            {  w = opposite(v,e);
              if  ( getLabel(w) = UNEXPLORED )
                  { setLabel(e, DISCOVERY);
```

```
                    topologicalDFS(G, w); }
                }
        Label v with topological number n;
        n = n - 1;
        return;
    }


    // part2
    Algorithm topologicalDFS(G)
        Input dag G
        Output topological ordering of G □ {   n = G.numVertices();
         for all  u ∈ G.vertices()
          setLabel(u, UNEXPLORED);
         for all  e ∈ G.edges()
          setLabel(e, UNEXPLORED);
         for all  v ∈ G.vertices()
         if   ( getLabel(v) = UNEXPLORED )
            topologicalDFS(G, v);
     }
```

3. KMP算法



j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
F(j)	0	0	1	0	1	2

```
    //part 1
```

```
Algorithm KMPMatch(T, P)
   {    F = failureFunction(P);
    i = 0;
    j = 0;
    // max_j =0;
    while ( i < n )
         if  ( T[i] = P[j] )
             { if  ( j = m - 1 ) // m = P.length
                     return  i - j ; // match
                     /*
                     如果是LCS 那题
                     return  (0,j)
                     */
                else
                     { i = i + 1; j = j + 1;}
              }
         else
             if (  j > 0 )
                   j = F[j - 1];
                   /*
                   如果是LCS 那题
                   if (max_j < j)
                        max_j = j
                   j = F[j - 1];
                   */
             else
                   i = i + 1;
         return  -1;  // no match
         /*
        如果是LCS 那题
        return (0,max_j)
        */
     }

// part 2
Algorithm failureFunction(P)
  {  F[0] = 0;
       i = 1;
       j = 0;
      while  ( i < m )
          if  ( P[i] = P[j] )
              { // we have matched j + 1 char
                  F[i] =  j + 1;
                   i = i + 1;
                   j = j + 1; }
```

```
            else if   (  j > 0 )
                 // use failure function to shift P
                         j = F[j - 1];
                   else
                        { F[i] = 0; //  no match
                          i = i + 1;}
    }


// KMP find LCS
// O(m(n+m)) 如果m<n
Algorithm Solution(T1,T2)
Input 2 text
Ouput start index and end index from shorter text
{
    if T1.length < T2.length
        T_long = T2
        T_short = T1
    else
        T_long = T1
        T_short = T2

    length = 0;
    LT =0; //LT记录loop_time
    while(!T_short.isEmpty)
    {
        (i,j) = KMPMatch(T_long, T_short)
        if (i-j+1 > length ) //长度
            length = i-j+1
            result(Start,End)= (i + LT, j + LT)
        LT = LT + 1
        T_short.RemoveFirst() //每次把T_short 的首字符去掉

    }
    return result(Start,End)
}
```

4. 木桶排序 wk8

```

Algorithm bucketSort(S, N)
    Input sequence S of (key, element)
      items with keys in the range [0, N - 1]  Output sequence S sorte
```

```
d by increasing
        keys
    { B = array of N empty sequences;
      while ( ¬S.isEmpty() )
          {  f = S.first();
             (k, o) = S.remove(f);
             B[k].insertLast((k, o)); }
        for ( i = 0; i++; i <=N - 1)
            while ( ¬B[i].isEmpty() )
                {     f = B[i].first();
                    (k, o) = B[i].remove(f);
                    S.insertLast((k, o)); }
      }

S is a sequence of n (key, element) entries with keys in the range [0, N - 1]
Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
Phase 1: Empty sequence S by moving each entry (k, o) into its bucket B[k]
Phase 2: For i = 0, …, N - 1, move the entries of bucket B[i] to the end of  sequence S
Analysis:
Phase 1 takes O(n) time
Phase 2 takes O(n + N) time
```

5. DFS找图的联通部分

5.1 DFs

```
// part 1
Algorithm DFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the edges of G □
            in the connected component of v □
            as discovery edges and back edges
  { setLabel(v, VISITED);
     for all  e ∈ G.incidentEdges(v)
         if  ( getLabel(e) = UNEXPLORED )
            {  w = opposite(v, e);
               if ( getLabel(w) = UNEXPLORED )
```

```
                    {  setLabel(e, DISCOVERY);
                       DFS(G, w);
                    }
                else
                     setLabel(e, BACK);
            }
   }

// part 2
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G □     as discovery edges and□
back edges
 {  for all  u ∈ G.vertices()
         setLabel(u, UNEXPLORED);
     for all  e ∈ G.edges()
         setLabel(e, UNEXPLORED);
     for all  v ∈ G.vertices()
         if  ( getLabel(v) = UNEXPLORED )
              DFS(G, v);
 }

// path finding
Algorithm pathDFS(G, v, z)
  {  setLabel(v, VISITED);
      S.push(v);
  if ( v = z )
     return S.elements();
  for all  e ∈ G.incidentEdges(v)
     if  ( getLabel(e) = UNEXPLORED )
       {  w = opposite(v,e);
          if  ( getLabel(w) = UNEXPLORED )
            {  setLabel(e, DISCOVERY);
               S.push(e);
                pathDFS(G, w, z);
               S.pop(e);
             }
          else
              setLabel(e, BACK);
       }
   S.pop(v);
}


// circle finding
```

```
Algorithm cycleDFS(G, v, z)
  {  setLabel(v, VISITED);
      S.push(v);
  for all  e ∈ G.incidentEdges(v)
     if  ( getLabel(e) = UNEXPLORED )
        {  w = opposite(v,e);
           S.push(e);
           if  ( getLabel(w) = UNEXPLORED )
              {  setLabel(e, DISCOVERY);
                  pathDFS(G, w, z);
                  S.pop(e);  }
           else
              {  T = new empty stack
                  repeat
                     {  o = S.pop();
                        T.push(o);  }
                  until ( o = w );
                  return T.elements();  }
   S.pop(v);
}

```

5.2 BFS

```
//part 1
Algorithm BFS(G, s)
  { L0 = new empty sequence;
 L0.insertLast(s);
 setLabel(s, VISITED);
 i = 0;
 while  ( ¬Li.isEmpty() )
   {  Li +1 = new empty sequence;
      for all  v ∈ Li.elements() □
         for all  e ∈ G.incidentEdges(v)
            if  ( getLabel(e) = UNEXPLORED )
              { w = opposite(v,e);
                /*
                //2- colorable 那题，如果bfs遍历的子节点与父节点颜色相同返回-1
                if (w.color == v. color)
                    return -1
                */
                if  ( getLabel(w) = UNEXPLORED )
                  {  setLabel(e, DISCOVERY);

```

```
                        setLabel(w, VISITED);
                        Li +1.insertLast(w);
                       //Q(w)=Q(v)+{v,w}; r
                    }
                    else
                        setLabel(e, CROSS);
                }
            i = i +1;
        }
    }

    // part 2

    Algorithm BFS(G)
        Input graph G
        Output labeling of the edges □      and partition of the □      ver
    tices  of G
      {
         for all  u ∈ G.vertices()
             setLabel(u, UNEXPLORED);
         for all  e ∈ G.edges()
             setLabel(e, UNEXPLORED);
         for all  v ∈ G.vertices()
             if  ( getLabel(v) = UNEXPLORED )
                  BFS(G, v);
      }

```

5.3 homework11 Q2