Notebooks

st out

- createStack(), which creates an empty stack.
- push(element), which adds an element to the collection, and
- pop(), which removes the top element from the stack.
- peek(), which returns the top element without modifying the stack.
- isEmpty(), which checks if the stack is empty.

空间复杂度 O(n)
每个操作时间复杂度O(1)

Bracket Matching (2/5)

Algorithm

```
#include "Stack.h"

bracketMatching(s):
|  Input  stream s of characters
|  Output TRUE if parentheses in s balanced, FALSE otherwise
|
|  for each ch in s do
|  |  if ch = open bracket then
|  |  |    push ch onto stack
|  |  else if ch = closing bracket then
|  |  |  if stack is empty then
|  |  |  |    return FALSE
|  |  |  else
|  |  |  |    pop top of stack
|  |  |  |    if brackets do not match then
|  |  |  |  |    return FALSE
|  |  |  |    end if
|  |  |  end if
|  |  end if
|  end for
|  if stack is not empty then return FALSE
```

Queue: First in first out

- enqueue(element): add a new element at the end of the queue
- dequeue(): remove the element at the front of the queue
- Other auxiliary operations:
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): indicates whether no elements are stored

Week3

Singly Linked List: two components, data and a pointer pointing to the next node
注意一点，最后一个点的指针是null

- Create a new linked list
- Create a new node
- Delete a node
- Insert a node
- Find a node containing particular data

Doubly linked List: three components, data, two pointers pointing to the previous node and next node.

Array:

Week4

- Control flow
  - **if** … [**else** …]
  - **while** …
  - **do** … **while** …
  - **for** …
- Method declaration

  Algorithm *method* (*arg* [, *arg*…])

  Input …

  Output …
- Method call

  var.method (*arg* [, *arg*…])
- Return value

  return *expression*
- Expressions

  = Assignment

  = Equality testing

  n^2Superscripts and other mathematical formatting allowed

Big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

Big-Omega
- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

Big-Theta
- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$
- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$
- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \bullet g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

Week5 and Week6

Map/Dictionary: store data with pair(key, value)
- find(item): find item in the collection
- insert(item): insert item into the collection
- remove(item): remove item from the collection

区别： map可以有重复key，dict不可以

Tree:

Notation:

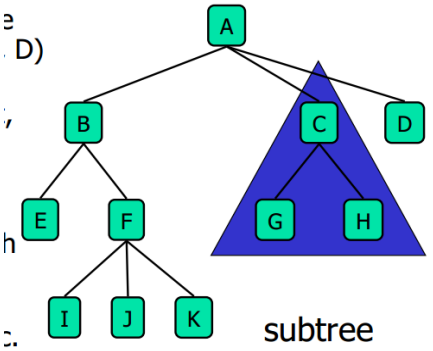
Root

Internal node

Leaf node

Depth: number of ancestors,根节点为0

Height: maximum depth (3)，叶节点为0



Traversal:

Preorder: 根左右

先根遍历(先访问后递归) : 始于根结点,依次遍历每棵子树(父结点优先,左子树优先)

Postorder:左右根

后根遍历(先递归后访问) : 始于叶结点,只有访问完某个结点的所有子树才会访问该结点

Inorder:左根右

某结点的左子树->该结点->该结点的右子树

Binary Tree

Arithmetic Expression Tree: internal为操作符,external为操作数

Decision Trees: internal为questions,external为decisions

searching

Proper binary tree: (完全二叉树) 指除了最后一层其他层都填满元素的二叉树

Full binary tree:(满二叉树) 所有层都填满，特殊的完全二叉树

一些概念:

- N : number of nodes;
- N_e : number of external nodes;
- N_i : number of internal nodes;
- H : height

各概念之间的关系 (可以互相推导)

1. $h+1 \le n \le 2^{h+1}-1$
2. $1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h-1$
4. $\log(n+1)-1 \le h \le n-1$

Also, if T is proper, then T has the following properties:

1. $2h+1 \le n \le 2^{h+1}-1$
2. $h+1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h-1$
4. $\log(n+1)-1 \le h \le (n-1)/2$

Binary Search tree

Nodes u,v,w : u 为 v 的左子结点 , w 为 v 的右子结点

$key(u) <= key(v) <= key(w)$

所有external node 都是空节点，不存储值

中序遍历BST，key是非降序排列

Operations:

Search:

Insertion:

- 先对key k进行搜索(TreeSearch);
- 若tree中不存在key k,插入到合适的位置;
- 若已经存在key k,更改储存内容为Node v;

Time complexity: O(h) h is the height of tree

Deletion:

- 先进行搜索, 确保树中存在key k并确定其位置Node v;
- 如果存在k, 对相应Node v 按以下几种情况讨论:
  - 删除v时将v连接的叶结点w一并删除;
  - 如果v只有一棵子树w , 则用这一棵子树替换v;
  - v有两棵子树 : 寻找 v 的predecessor 或者 successor 替换 v;

Predecessor and successor 就是中序遍历结果中被删除节点的前和后节点

Time complexity: O(h) h is the height of tree

best case h=logn

worst case h=n

Space complexity: O(n)

AVL tree

Balanced : 任何结点的两棵子树高度差不超过1

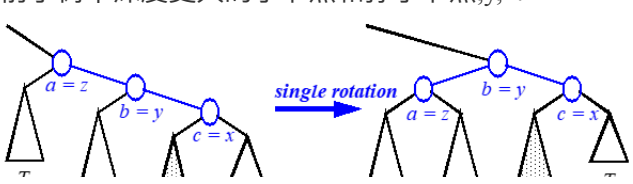
The height of an AVL tree storing n keys is O(log n).

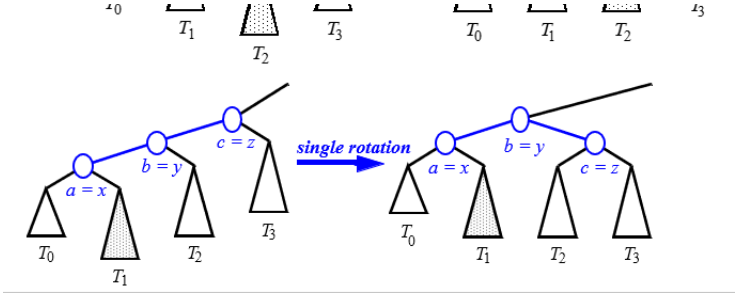
Insertion:

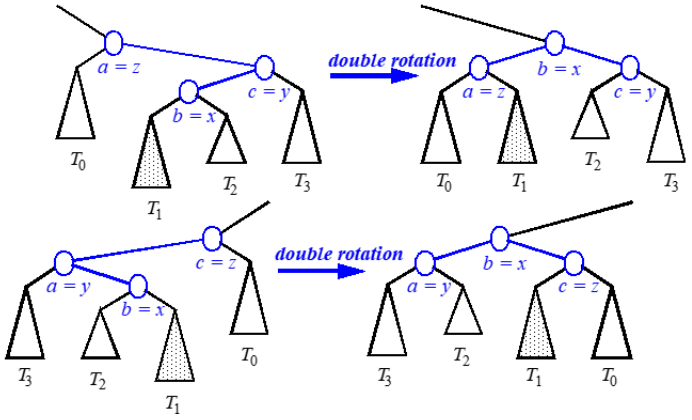
旋转:

先找到 不满足balanced的子树节点，z.

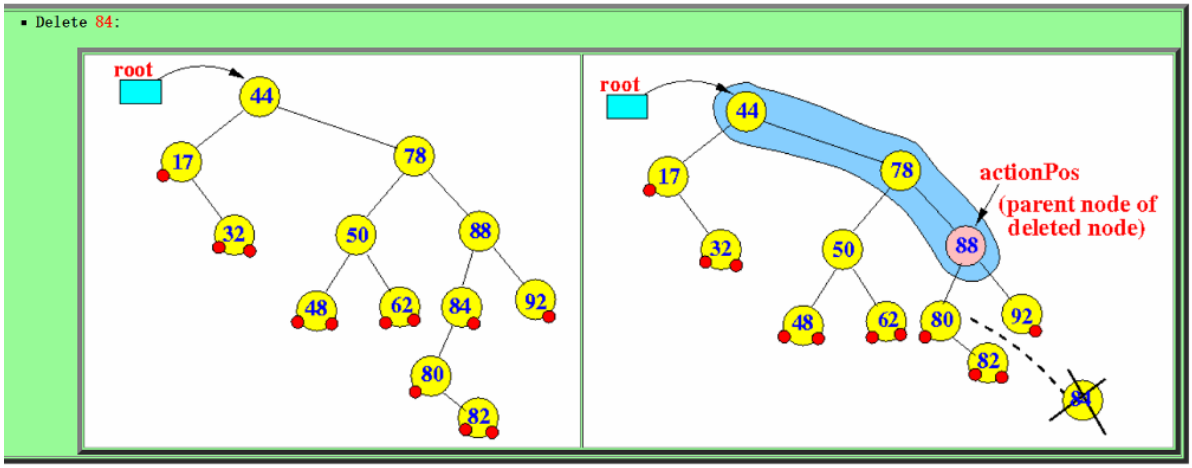
找到当前子树中深度更大的子节点和孙子节点,y,x.



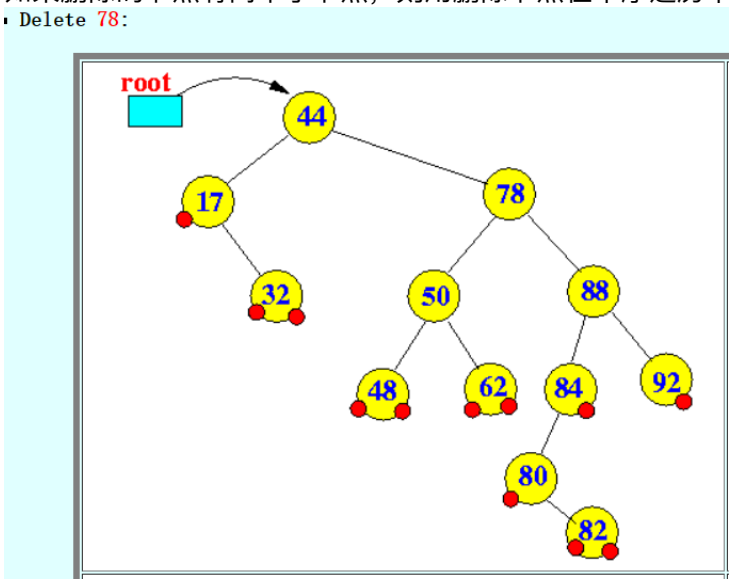


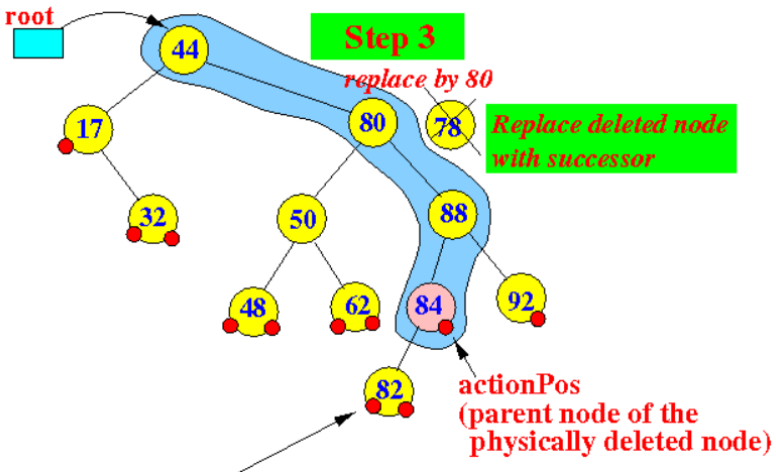


Deletion：跟insertion 一样，删除后需要重构。如果被删除节点没有子节点（internal node）那么就直接删除
如果删除的节点有一个子节点，则直接把删除节点的父节点和它的子节点连接在一起。



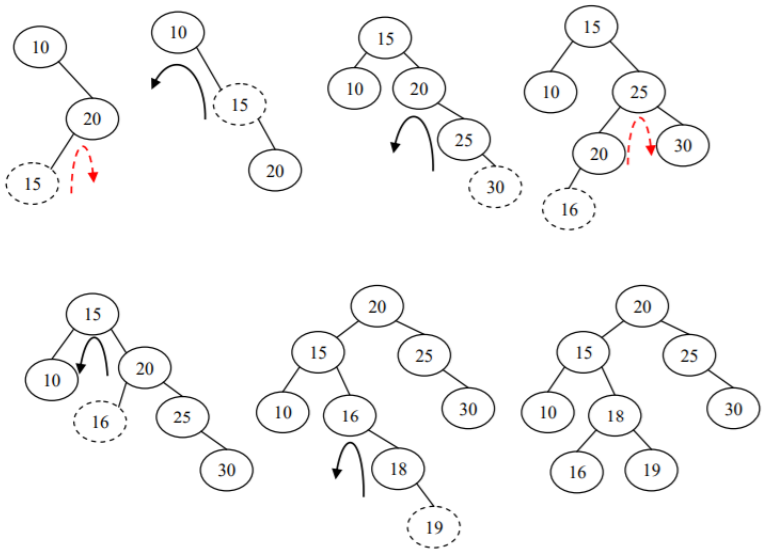
如果删除的节点有两个子节点，则用删除节点在中序遍历中的前节点或者后节点替换删除节点。



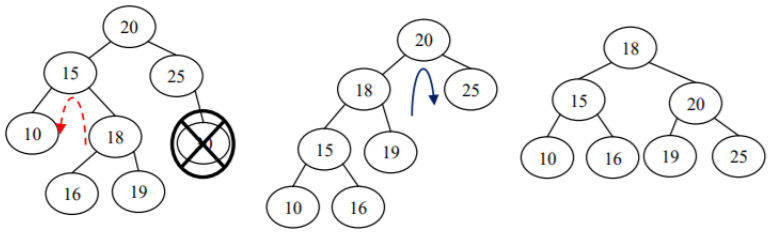


Exercise:

Insert the following sequence of elements into an AVL tree, starting with an empty tree: 10, 20, 15, 25, 30, 16, 18, 1



Delete 30 in the AVL tree that you got.



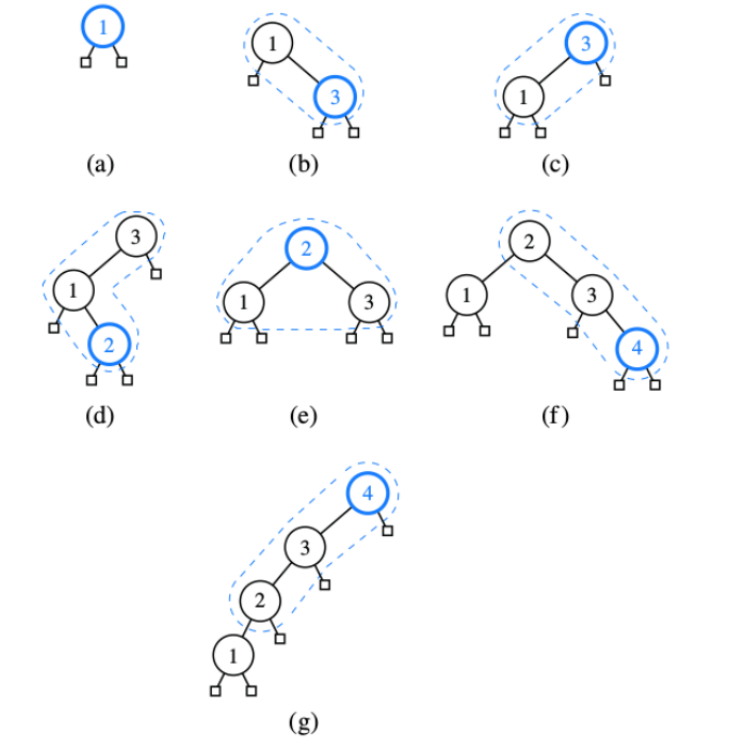
- 一次重构$O(1)$
- 查找$O(logn)$
- 插入(查找+重构) $O(logn)$
- 删除(查找+重构) $O(logn)$

Splay Trees

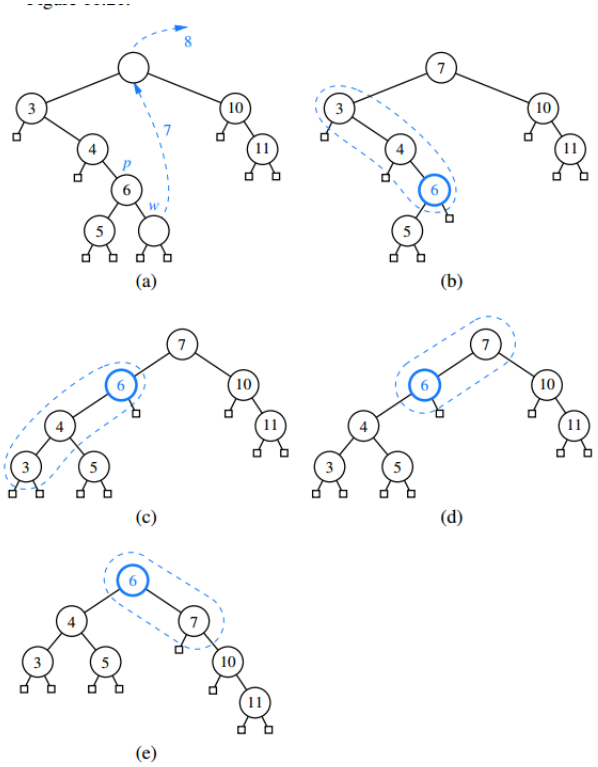
伸展树是一种特殊的BST,适合多次连续操作
类似AVL树,可以通过旋转来改变结点分布,进而减小深度;
差别在于, 伸展树不一定是平衡的,左右子树可以相差任意深度;
Goal : 指定某个结点x , 通过不断旋转将x传送到到root位置;

- find(k):
  - 找到该结点则从该结点开始splay;
  - 没找到该结点,则从搜索终止位置(叶结点)的父结点开始splay;
- insert(k,v):

- 使用被插入的新结点进行splay;

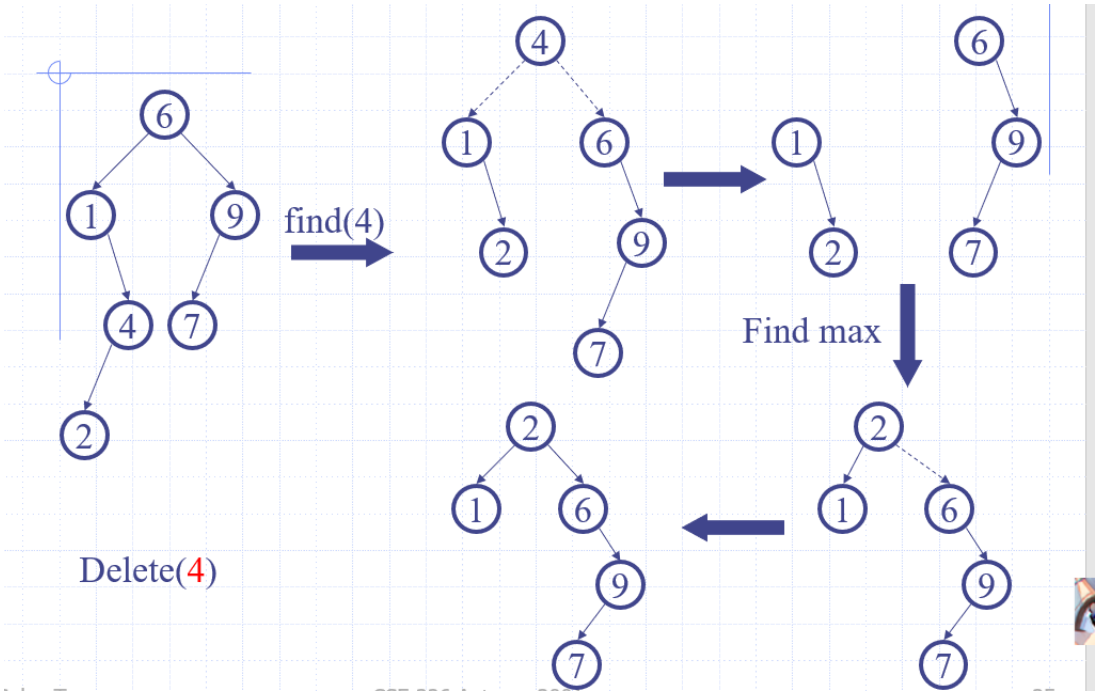


- remove(k):
  - 使用被删除结点的predecessor w替换此结点
  - 从w原来的父结点p开始进行splay过程



另一种方法：

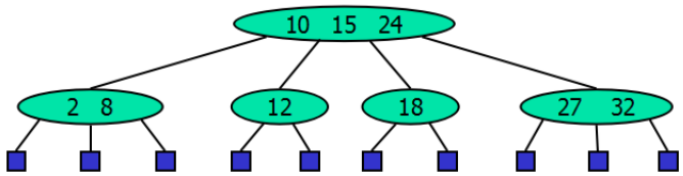
把要删除的节点splay到root节点，删除后，就剩下两个分开的子树，选取左子树中值最大的节点splay到置。



单次搜索时和BST类似可能需要$O(n)$
splay过程平均耗费$O(logn)$

2-4 Tree

每个结点最多包含3个结点(4个子结点),所有的external深度相同.



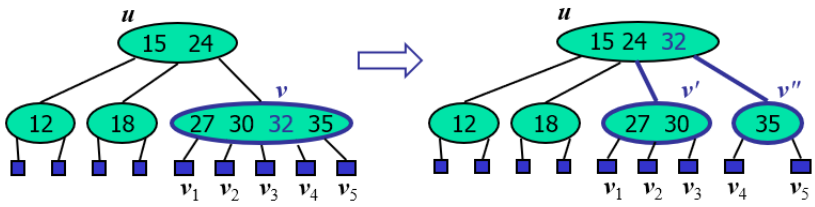
Height of a (2,4) Tree $O(logn)$

- 对于此的证明可以从以下不等式入手
  - 最高情况: 所有内部结点都是单结点
  - 最矮情况: 所有内部结点都是三结点

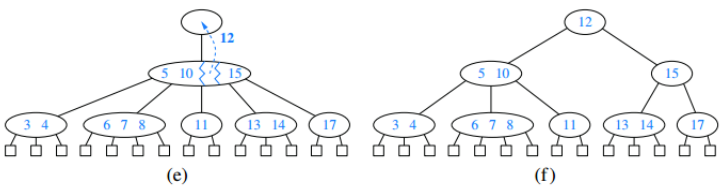
$$\frac{1}{2}log(n+1) \le h \le log(n+1)$$

- search/insert/remove $O(logn)$
- split/fusion/transfer $O(1)$

Insertion and Overflow then split:

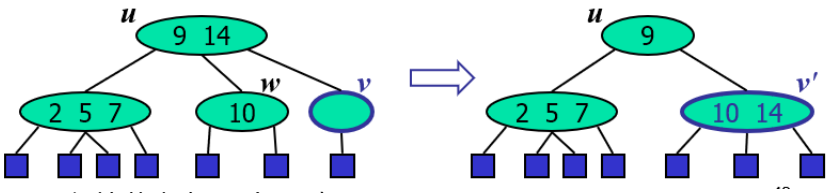


如果root节点也overflow，则拆分root节点

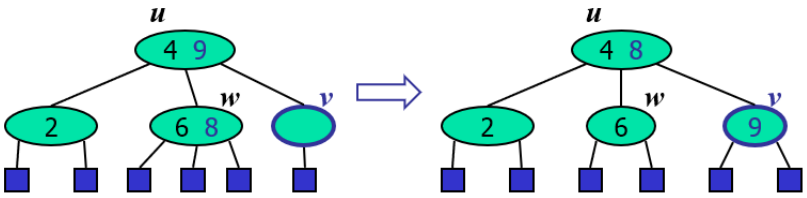


Deletion:

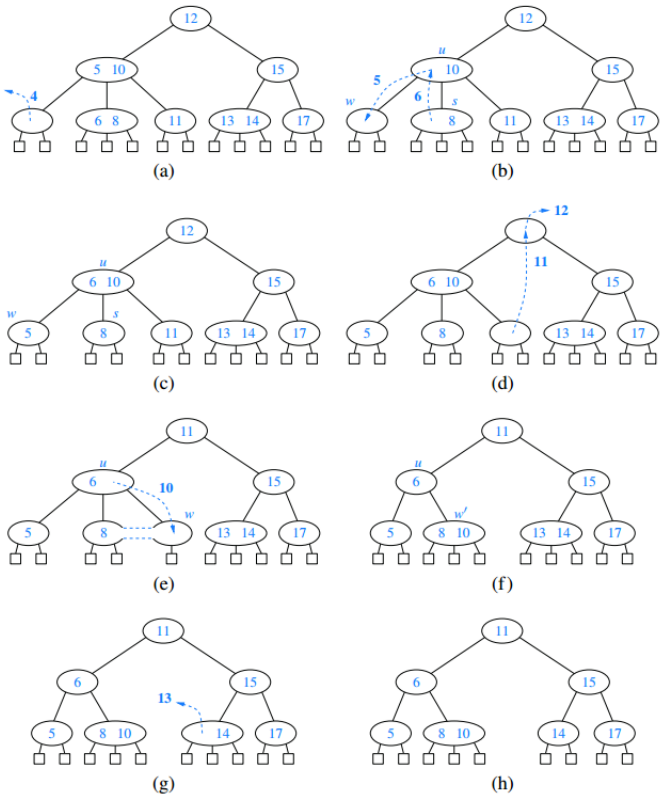
Fusion:（邻接节点只有一个data时）



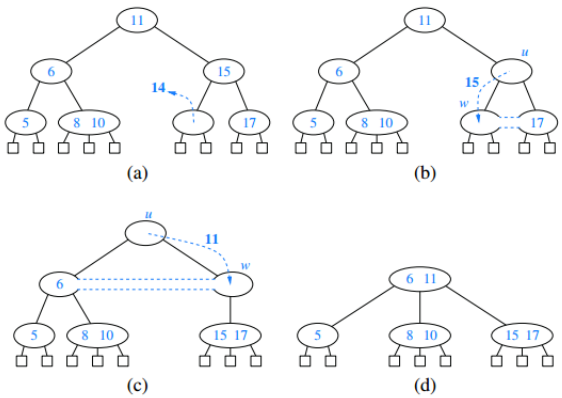
Transfer:(邻接节点有2-3个data)



Exercise:



Delete 4,12,13



Delete 14

Week7

Priority Queue :priority queue stores a collection of entries (key-value pairs)

- Insert(k, x)
  Inserts an item with key k and value x.
- RemoveMin() (RemoveMax())
  Removes and returns the item with smallest key (largest key). We consider RemoverMin() only. Implementation of RemoveMax() is similar.
- Min() (Max())
  returns, but does not remove, an entry with smallest key (largest key)
- Size(), IsEmpty()

  entry允许相同key存在;

List-based Priority Queue

- unsorted list : 4-5-2-3-1
  - insert : $O(1)$ (直接头尾插入)
  - removeMin()/min() : $O(n)$ (需要先找到最小的)
- sorted list : 1-2-3-4-5
  - insert : $O(n)$ (按顺序插入)
  - removeMin()/min() : $O(1)$ (有序队列)
  - 排序后的队列不适合插入，但很适合查找

Selection-Sort

- 基于未排序list的优先队列 $O(n^2)$
- 先随便插入P，每次移除P中最小的元素插入S
  - 插入n个元素 $O(n)$
  - 移除最小元素 $1+2+\ldots+n=O(n^2)$
- 选择排序：每次从剩余元素中选取最小的交换到相应位置

	List S		*Priority Queue P*
Input:	(7,4,8,2,5,3,9)		()
Phase 1			
(a)	(4,8,2,5,3,9)		(7)
(b)	(8,2,5,3,9)	(7,4)	
..	.. ..		
.	. .		
(g)	()		(7,4,8,2,5,3,9)
Phase 2			
(a)	(2)		(7,4,8,5,3,9)
(b)	(2,3)		(7,4,8,5,9)
(c)	(2,3,4)		(7,8,5,9)
(d)	(2,3,4,5)		(7,8,9)
(e)	(2,3,4,5,7)	(8,9)	
(f)	(2,3,4,5,7,8)		(9)
(g)	(2,3,4,5,7,8,9)		()

Insertion-Sort

- 基于有序数组的优先队列 $O(n^2)$
- 插入过程中排序(P时刻保持有序),移除时直接按顺序移除
  - 插入n个元素 $1+2+\ldots+n=O(n^2)$

- 插入n个元素1,2,3...n $= O(n^2)$
  - 移除最小元素 : $O(n)$
- 每次插入新元素，都把当前队列进行排序，保证插入下一个元素前队列是有序的

Insertion-Sort Example

	List S	*Priority queue P*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
.	.	.
(g)	(2,3,4,5,7,8,9)	()

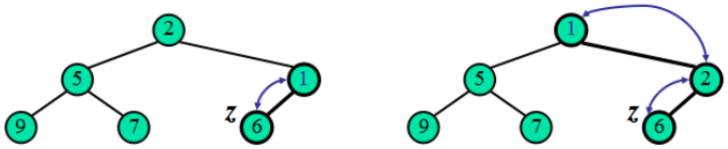
Heap

- Heap的形状类似二叉树，但规则不同
  - 每个子结点都保证不小于父结点，root最小 (最小堆);
  - Heap-order : 除了root的internal node v , key(v) >= key(parent(v));
- Complete Binary Tree:
  - 深度为h的这一层结点数为 2^h
  - 上一层没被填满绝对不在下一层添加子树
  - 尾结点是深度h(最底层)的最右结点;
- 包含n个结点的heap高度 $h = O(logn)$

Insertion:

找到对应位置插入数据，然后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 $O(logn)$



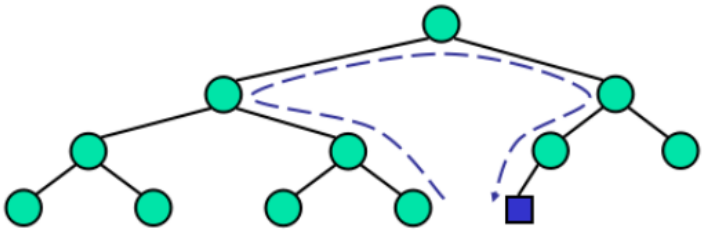
Deletion：

- 每次删除的都是根结点: removeMin()
- 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 $O(logn)$;

Updating the last node

- 每次插入或删除操作时都需要更新尾结点位置;
- 更新步骤:
  - 先从指定结点不断向上直到到达根结点或是根结点的左子结点;
  - 如果到达的是左子结点，切换到该结点的兄弟结点(根结点的右子结点);
  - 从根结点的右子结点开始，不断向左下移直到到达叶结点;
  - 该叶结点即为当前尾结点;
- 更新操作耗时 $O(logn)$



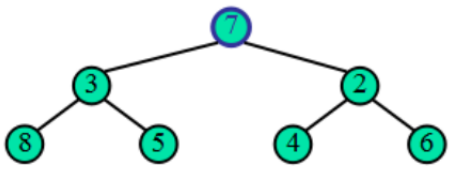
Heap-sort:

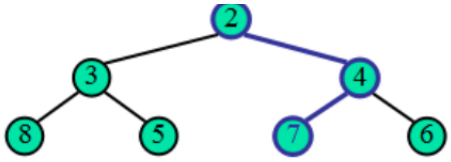
- 使用heap实现优先队列:
  - 空间复杂度$O(n)$;
  - insert()/removeMin() : $O(logn)$
  - size()/isEmpty()/min(): $O(1)$

- 基于heap的优先队列排序 :$O(nlogn)$
- 实现堆排序:
  - 给出一个堆和一个空队列;
  - 每次对堆进行removeMin()操作，提取堆顶元素;
  - 将提取出来的元素一次插入空队列，实现排序;

Merging Two Heaps

- Input 2 heaps and a key k;
- 创建一个新heap,根结点key为k,两棵子树为打算merge的heaps;
- 检验各结点是否满足heap-order;
- 对破坏规则的使用downheap来重构;





Bottom-up Heap Construction

- $\log n$ phases,构造时间$O(n)$，比从上到下省时间
- 其实就是不断归并子堆的过程
- 对于phase i : 将一对拥有2^i-1 元素的堆被归并成一个$2^{(i+1)}-1$ 元素的堆

每次merge之后会进行downheap

8.7 Union-Find Partition Structures

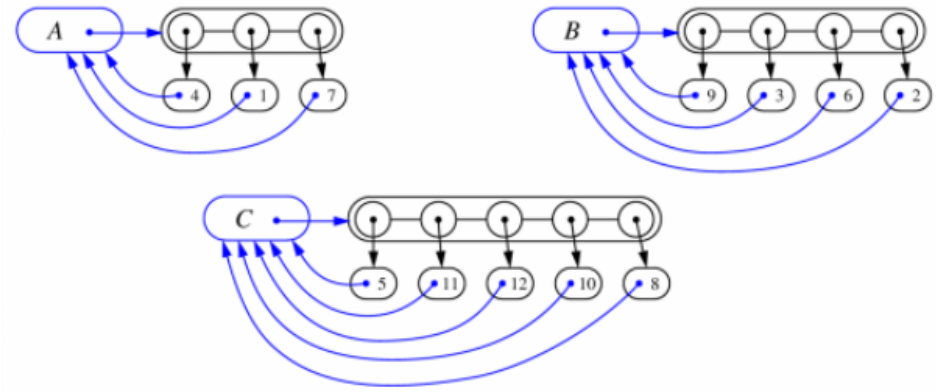
- 用于处理一些不相交集合（Disjoint Sets）的合并及查询问题
- 时间复杂度 $O(nlogn)$
- 举个栗子: 给出一组结点, 判断是否连通

8.7.1 Partitions with Union-Find Operations

- makeSet(x) : 创建一个包含x的singleton set(单元素集)并返回该set中存储x的位置;
- union(A,B): 返回set($A \cup B$) 并删除两个旧数据集;
- find(p): 确定元素p属于哪些子集，返回包含元素p的sets;

8.7.2 基于list的实现

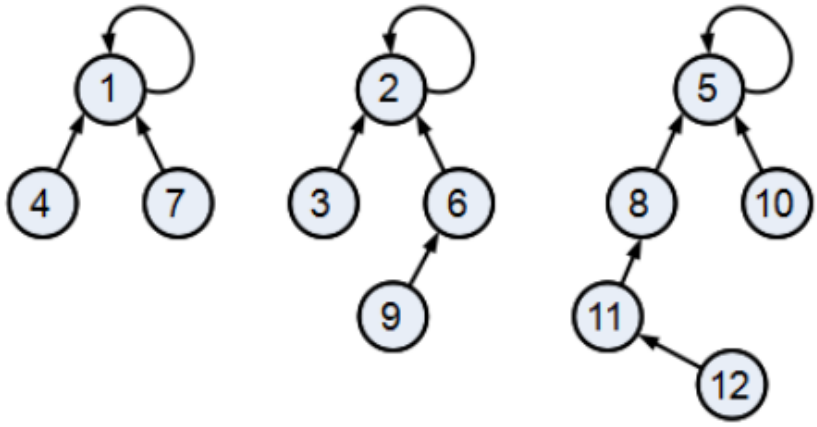
- 每个set储存在链表序列内;
- 每个node储存一个object(element,reference)，reference都指向链表的名字;



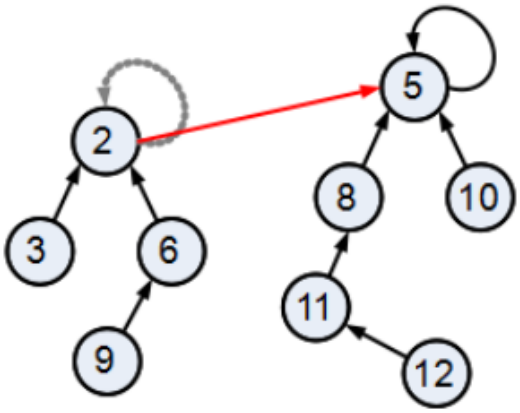
- 分析:
  - 每次union操作 : 将元素从小set移动到大set;
  - 这个大set的size至少是小set的两倍;
  - 因此一个元素最多被移动$O(logn)$时间;
  - n次union-find耗时$O(nlogn)$;

8.7.3 基于tree的实现

- 每个元素储存在node里,且包含一个指向set名的指针;
- 对于根结点v,set指针指向自身,那么v也是一个set名;
- 子结点的链接都是指向父结点的;
- 每个set都是一棵树:根为指针指向自身的node;



- Union-Find Operations
  - union: 将小树的root指向大树的root ,小树被并入大树;



- find : 返回这个node所在的root(set name)
  - 从指定node开始向上查找,直到到达指向自身的那个node(root);
  - 找到了根结点就可以得知set的名字了;

Path compression:

- After performing a find, compress all the pointers on the path just traversed so that they all point to the root

Week8

Pattern Matching:

Let P be a string of size m

A substring P[i .. j] of P is the subsequence of P consisting of the characters with ranks between i and j

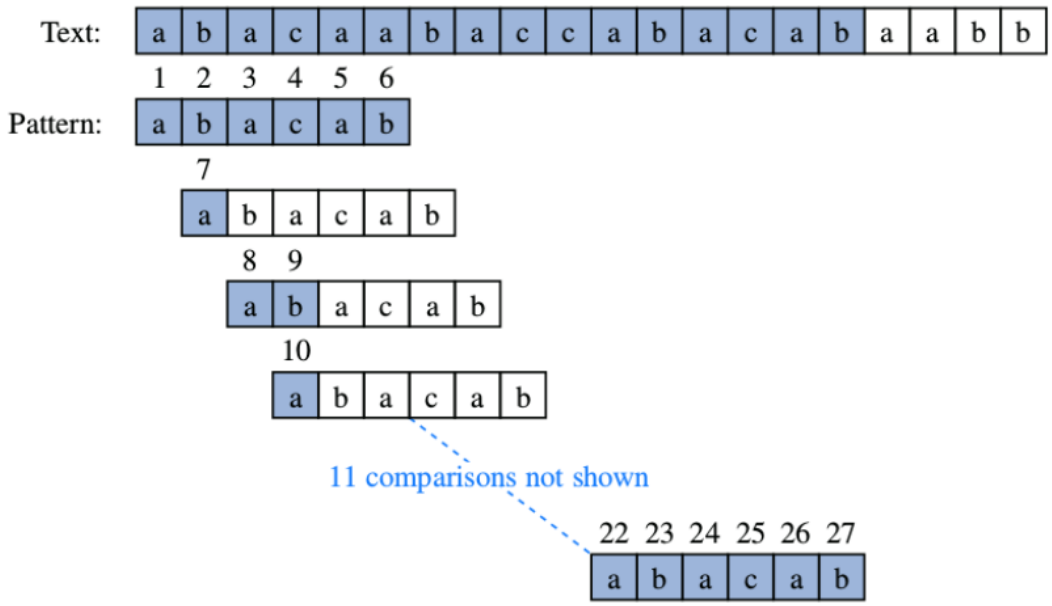
A prefix of P is a substring of the type P[0 .. i]

A suffix of P is a substring of the type P[i ..m - 1]

Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal t

- Brute-Force : 从P头比较,找不到严格后移一位 , $O(nm)$
- Boyer-Moore : 从P尾比较+Last-occurrence function , $O(nm+s)$
- Knuth-Morris-Pratt : 从P头比较,出现冲突时可以跳过一些已经比较过的位置;

Brute force



Boyer-Moore:

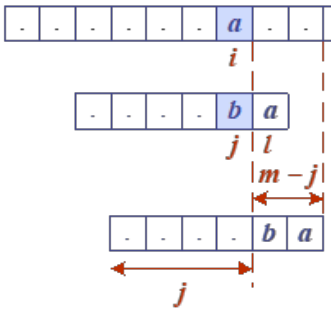
Boyer-Moore算法基于Last-Occurance Function

- 该函数返回字符集中每个字符在P中最后出现的位置
- 栗子:
  - 字符集S={a,b,c,d}
  - P=abacab
  - L(a)=4,L(b)=5,L(c)=3,L(d)=-1
- 该函数操作时间取决于size(P)=m,以及size(S)=s : $O(m+s)$

Algorithm *BoyerMooreMatch(T, P, Σ)*

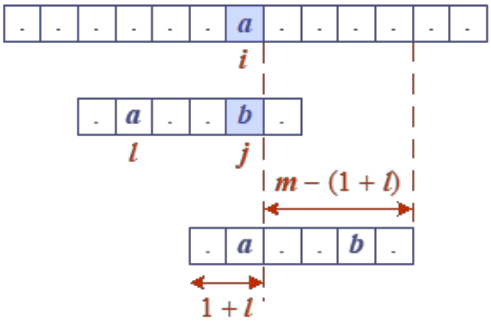
```
{ L = lastOccurrenceFunction(P, Σ)
  i = m − 1
  j = m − 1
  repeat
     if ( T[i] = P[j] )
        { if (j = 0 )
             return i // match at i
          else
             { i = i − 1;
               j = j − 1; }
        }
     else // character-jump
        { l = L[T[i]];
          i = i + m − min(j, 1 + l);
          j = m − 1; }
  until (i > n − 1)
  return −1 // no match
}
```

Case 1: $j \le 1 + l$



如果l的最后出现位置在当前j的后面，那么i的坐标直接跳过已经检测过的字符串m−1−j+1，即m−j

Case 2: $1 + l \le j$



如果l的最后出现位置在当前j的前面，那么就把这两个字符对齐，然后从尾开始检测。

KMP:

使用KMP Failure Function求取部分匹配值

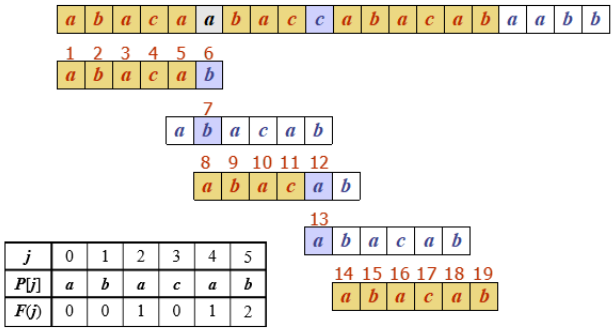
- 前缀 : 除了最后一个字符外的全部头部组合
- 后缀 : 除了第一个字符外的全部尾部组合
- F(j) : 前缀和后缀共有的最长组合的长度
- 注意要从第2位开始才会计算这个, 即F[1], 忽略F[0]

举个栗子说明failure function :

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

```
ab:前缀[a],后缀[b]-->F[1]=0
aba:前缀[a,ab],后缀[a,ba]-->F[2]=1
abaa:前缀[a,ab,aba],后缀[a,aa,baa]-->F[3]=1
abaab:前缀[a,ab,aba,abaa],后缀[b,ab,aab,baab]-->F[4]=2
abaaba:前缀[a,ab,aba,abaa,abaab],后缀[a,ba,aba,aaba,baaba]-->F[5]=3
```

```
Algorithm KMPMatch(T, P)
{ F = failureFunction(P);
  i = 0;
  j = 0;
  while ( i < n )
    if ( T[i] = P[j] )
      { if ( j = m − 1 )
          return i − j ; // match
        else
          { i = i + 1; j = j + 1;}
      }
    else
      if ( j > 0 )
        j = F[j − 1];
      else
        i = i + 1;
  return −1; // no match
}
```



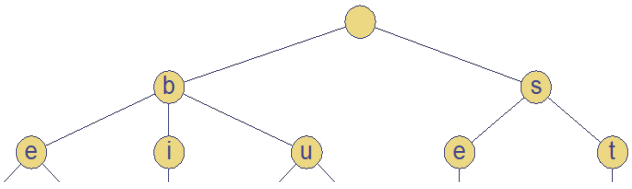
j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
F(j)	0	0	1	0	1	2

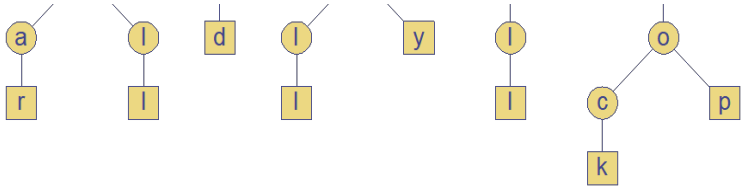
- Failure function : $O(m)$
- while-loop : 不大于2n次迭代
- KMP的运行时间$O(m+n)$, 优于暴力查找 $O(mn)$

Tries:

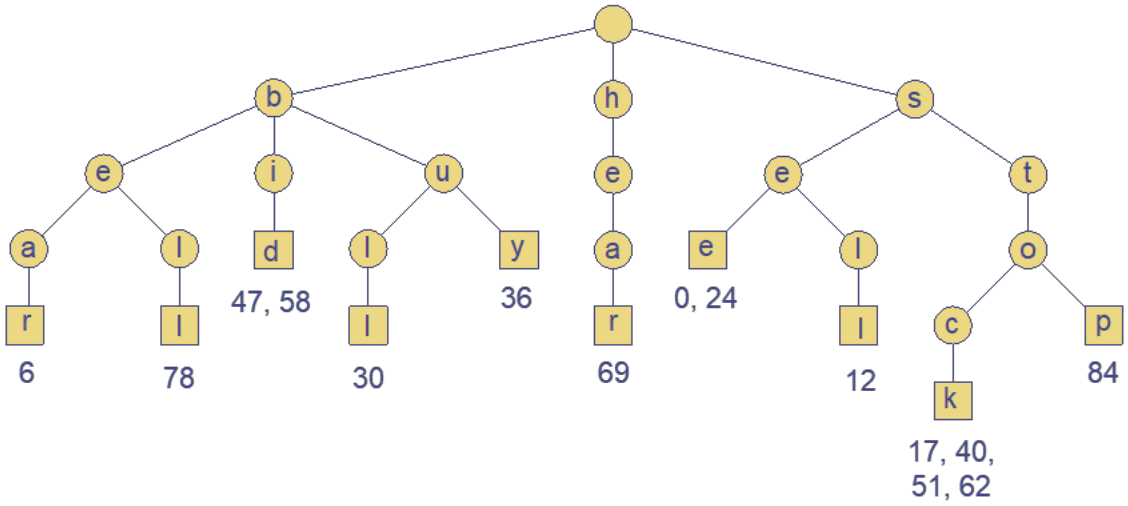
Standard Tries:

- 基本构造:
  - 除了root外每个node都配有一个字母标签, root 代表空格;
  - 结点的几棵子树按字母表顺序排序 , root的子结点为单词首字母;
  - 从root到external的路径构成S中一个字符串
  - 每个leaf还储存了这条路径代表单词在总字符串中的位置(首字母)



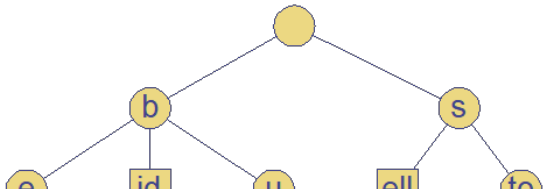


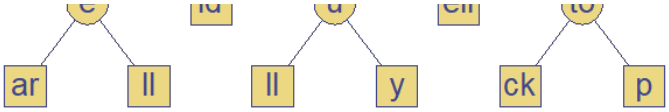
- 对于总长度为n,存储了s个字符串的标准字典树T:
  - n : 总长度;
  - m : 每次操作动用字符串参数的数量;
  - d : 字母表的size;
  - 空间复杂度 : $O(n)$
  - 搜索/插入/删除操作 : $O(dm)$
  - T的高度为s个字符串中最长的那个的长度;
  - 每个internal最多d个子结点;
  - T有s个external;
  - T的结点数量最多为n+1(所有单词不共享路径,再加上根结点);
- 使用字典树进行文本匹配
  - 将文本插入trie;
  - 每次操作始于搜索:先查询首字母,然后一步步查询单词;
  - 到达叶结点还能给出这个单词首字母的位置;

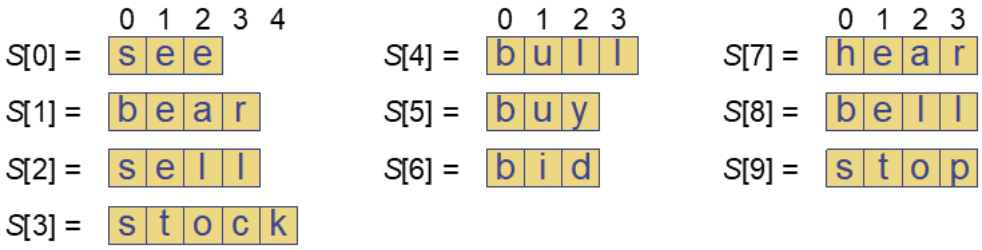


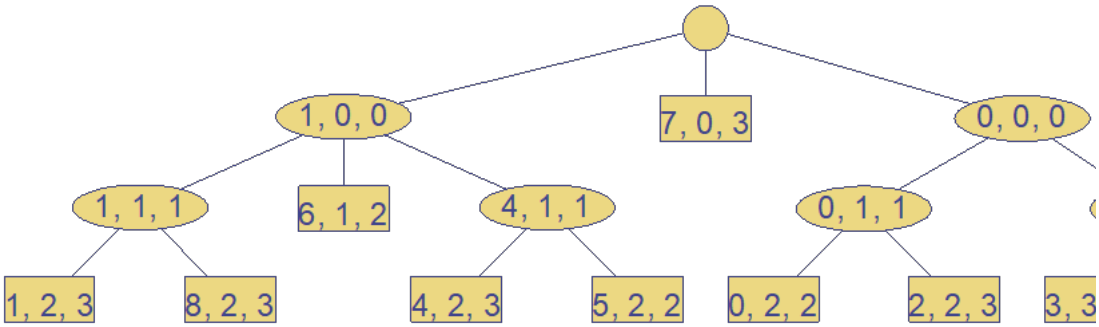
Compressed Tries:

- 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



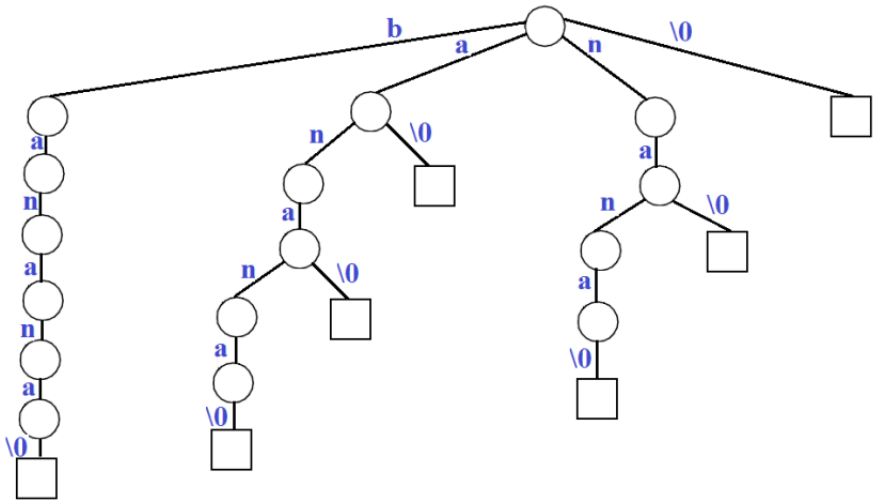


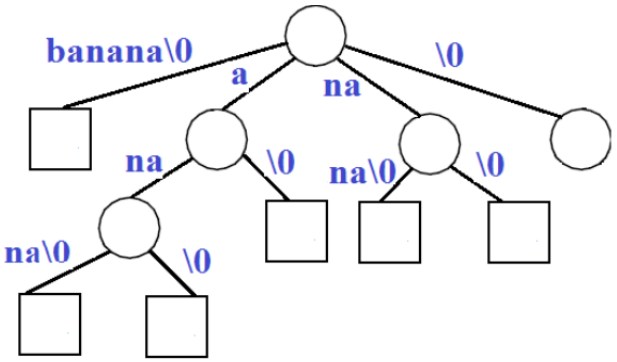




Suffix Tries:

- 后缀Trie构建: 以"banana为例"
  - 首先获得当前字符串的全部后缀
  - 将所有后缀根据其公共前缀构建成前缀树: 例如以"b"为前缀的后缀只有"anana",以"a"为前缀的后缀有"na"和"nana"...





- 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 $O(dm) \rightarrow O(m)$;
- 构建字典树 $O(n)$, 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

```
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 of X" ; }}}}
     until childTraversed=false or T.isExternal(v);
     return "P is not a substring of X" ; }
```

算法对比:

	Preprocess Pattern	Preprocess Text	Space	Search Time
Brute Force			O(1)	O(mn)
Boyer Moore	O(m+d)		O(d)	O(n) *
Suffix Trie		O(n)	O(n)	O(m)

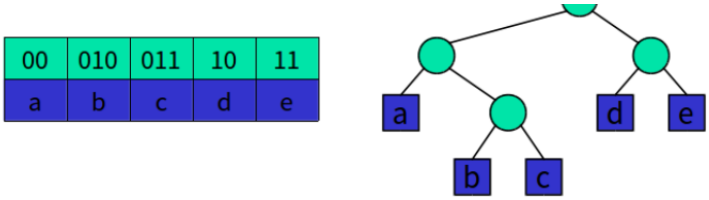
n = text size
m = pattern size

* on average

Huffman Tree:

Encoding Tree:

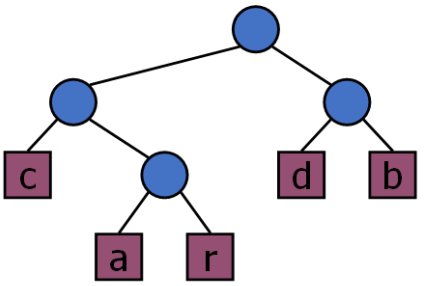
- code:将字母表内每个字母都转化为二进制编码;
- prefix code: code的一种,没有code-word是另一个code-word的前缀;
- encoding tree: 用于展示prefix code
  - 每个external储存一个字母;
  - 每个字母的code word取决于从root到叶结点的路径;
  - 0为左子树,1为右子树
  - 例如从root某个字母的路径是左-右-左,则该字母的codeword为"010"

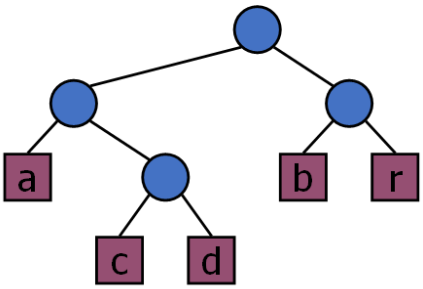


- optimal encoding tree:让字符串X的code尽可能短
  - 高频字母的code-words很短;
  - 低频字母code-words较长;
  - 栗子 : T2优于T1;

- Example
  - X = abracadabra
  - T_1 encodes X into 29 bits

T_2 encodes X into 24 bits





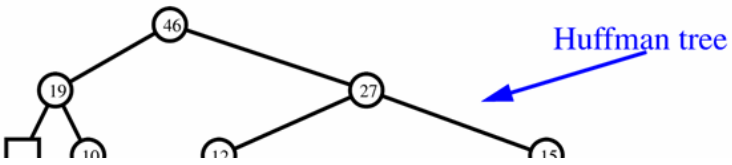
Huffman's Algorithm

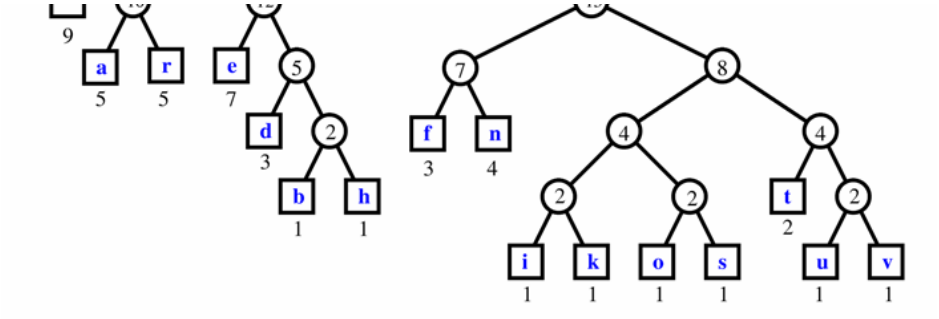
- 构建optimal encoding trie : 高频浅低频深
- 基于Heap(反正只要是优先队列就好咯):
  - 首先将全部元素插入Heap, key为频率
  - 每次removeMin()两次得到两个低频元素,将其归并作为这俩元素的父结点再插入Heap, 重复这一过程直到堆里就剩下一个键值对
  - 最后这个键值对的key就是优化后的总size, 使用removeMin()输出

- 算法分析:
  - 运行时间 $O(n + d\log d)$,d为字母表size;
  - 基于heap PQ:
    - 先全塞进去
    - 每次拿出来两个放回去一个直到堆里就剩下最后一个

String: **a fast runner need never be afraid of the dark**

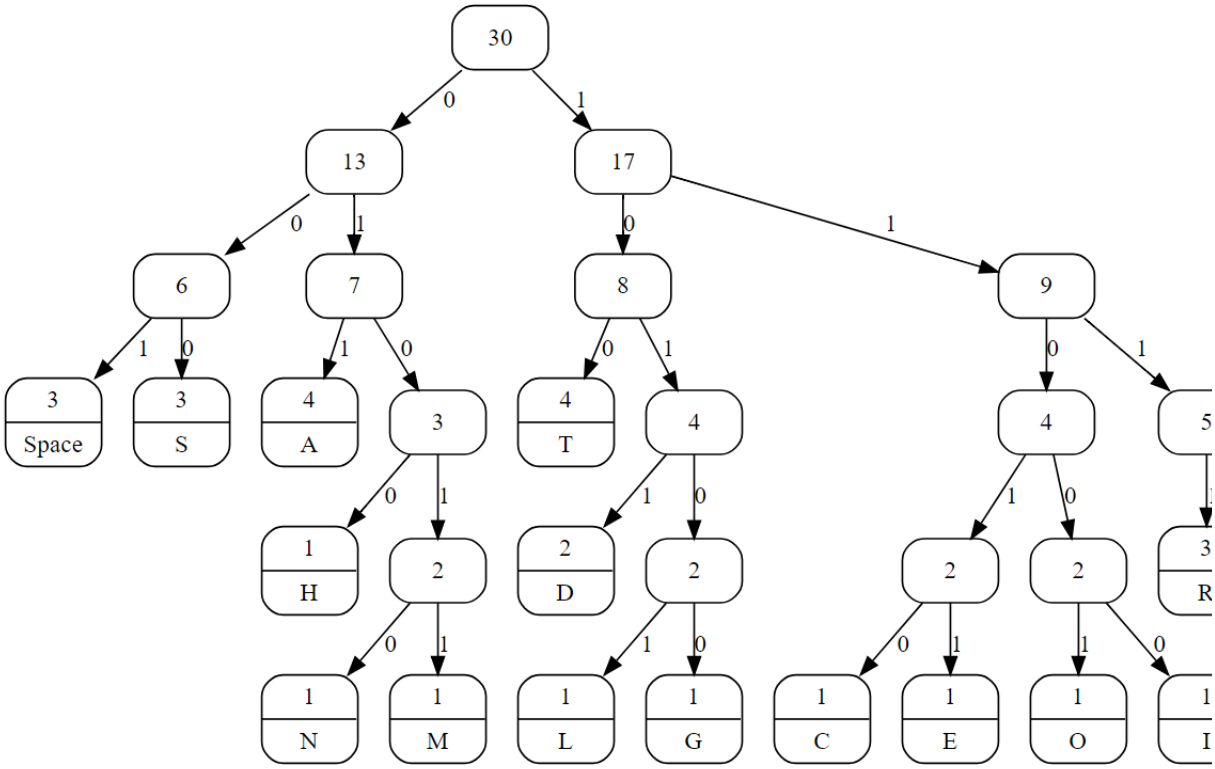
Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1





Exercise:

DATA STRUCTURES AND ALGORITHMS



Fractional Knapsack Problem

- 本课程不考察!
- Given: Set S with n items,每个item i包含
  - b_i : positive benefit
  - w_i : positive weight
- Goal: 在weight <= W的前提下尽可能最大化benefit
- fractional knapsack problem:
  - x_i : 取走item i的比例 , 比如某个item只取一半
  - 目标 : maximize $\sum_{i \in S} b_i(x_i/w_i)$

Fractional Knapsack Problem

- Given: Set S with n items,每个item i包含
  - b_i : positive benefit
  - w_i : positive weight

- Goal: 在weight <= W的前提下尽可能最大化benefit
- fractional knapsack problem:
  - x_i : 取走item i的比例 , 比如某个item只取一半
  - 目标 : maximize $\sum_{i \in S} b_i(x_i/w_i)$
  - Constraint: $\sum_{i \in S} x_i \leq W$

```
Algorithm fractionalKnapsack(S,W)
Input:set S(benefit bi,weight wi), max weight W;
Output:返回得到最优化的xi;
{
    for i in S{//初始化xi,value
        xi=0;
        vi=bi/wi;//value
    }
    w=0;//total weight
    while(w<W){
        remove item i with the largest vi;
        xi=min(wi,W-w);
        w=w+xi;
    }
}
```

- 算法分析: 运行时间$O(nlogn)$

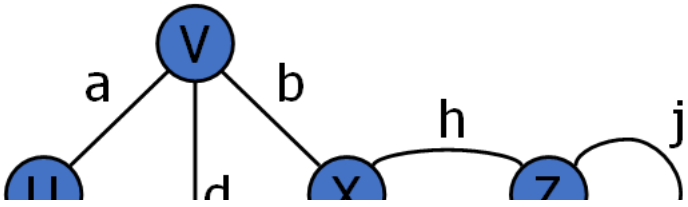
Week9-11
Graph

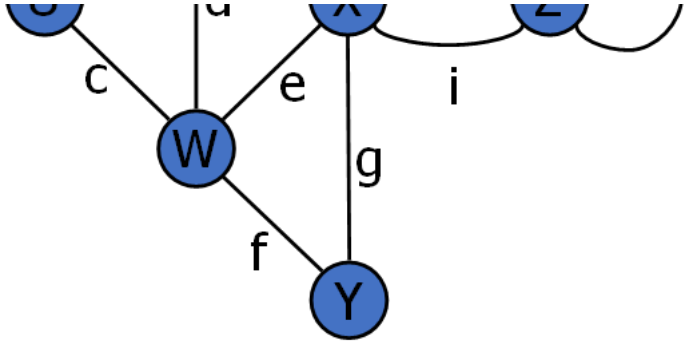
- 定义:graph=pair(V,E)
  - V(vertices) : a set of nodes;
  - E(edges) : a collection of pairs of vertices;
- 端点和边都是store element的positions

Edge Type:

- Directed edge:
  - 有序顶点对(u,v);
  - 始于u,止于v,逆过程不成立;
  - 栗子 : flight;
- Undirected edge:
  - 无序顶点对(u,v),uv可互相访问;
  - 栗子 : flight route;
- Directed graph:
  - 所有的边都是directed edges;
  - 栗子 : route network;
- Undirected graph:
  - 所有的边都是undirected edges;
  - 栗子 : flight network;

Terminology:





End vertices(end points) of an edge : 某条边的两个端点;
Adjacent vertices : 与某个端点间隔一条边的端点;
Edge incident on a vertex : 与某个端点连接的所有边
Degrees of a vertex : 某个端点连接边的数量;
Parallel edges : 起止端点相同的两条边;
Self-loop : 起止端点是同一个点的某条边;
Simple path : 经过的端点和边不重复;
Simple cycle : 经过的端点和边不重复;
Tree: connected graph with no cycles
Spanning tree: tree containing all vertices
Clique: complete subgraph

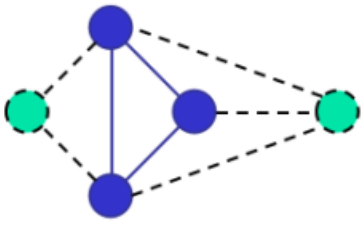
Properties:

Given:

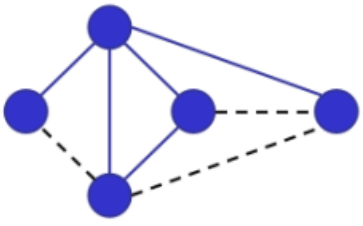
n : number of vertices;
m : number of edges;
deg(v) : degree of vertex v;

- 度与边的关系:
  - 无向图: $\sum_v deg(v) = 2m$ (每条边两端点，计算两次)
  - 有向图: $\sum indeg(v) + \sum outdeg(v) = 2m$
- 点与边的关系:
  - 对于无向的简单图:
    - $m \leq n(n-1)/2$;
    - $deg(v_i) \leq n-1$;
  - 其他无向图:
    - 连通图: $m \geq n-1$
    - 树: $m = n-1$
    - 森林: $m \leq n-1$
  - 有向图:
    - $m \leq n(n-1)$

- Spanning subgraph:包含G所有的端点,但边要少
  - 子图的端点不多于母图,边少于母图
  - Spanning Tree的边数为n-1



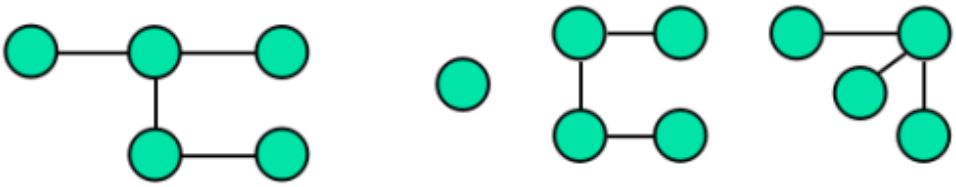
Subgraph



Spanning subgraph

11.2.3 Trees and Forests

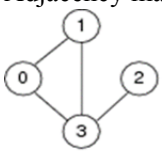
- Tree : 完全连通的无向图 , no cycles;
- Forest : 不完全连通的无向图,no cycles , 其中每个连通的部分都是tree;
- 树和森林的区别在于是否connected;



Tree Forest

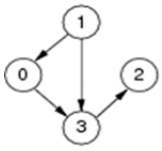
Simple graph: no loops and Parallel edges
Representation:

Adjacency matrix



Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0



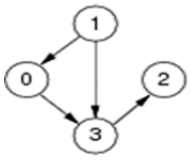
Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

Adjacency List

Undirected graph

A[0] = <1, 3>
A[1] = <0, 3>
A[2] = <3>
A[3] = <0, 1, 2>



Directed graph

A[0] = <3>
A[1] = <0, 3>
A[2] = < >
A[3] = <2>

space usage initialise insert edge remove edge adjacency matrix 712 712 1 1 adjacency list n+m n 1

- Space complexity: O(n^2
  - if a graph is sparse, most
- Time complexity:
  - initialisation: O(n^2) (init
  - insert an edge: O(1) (se
  - delete an edge: O(1) (u

- Space complexity: O(n+ of edges
- Time complexity:
  - initialisation: O(n) (initia
  - insert an edge: O(1) (ins

adjacency matlix disconnected(v)? n isPath(x,y)? copy graph destroy graph 172 172 adjacency list 1 n+m n+m n+m

(digraph) or two lists (un
duplicates

- delete edge: O(m) (nee
- If vertex lists are sorted
  - insertion requires search
  - deletion always requires

无向图：

DFS:

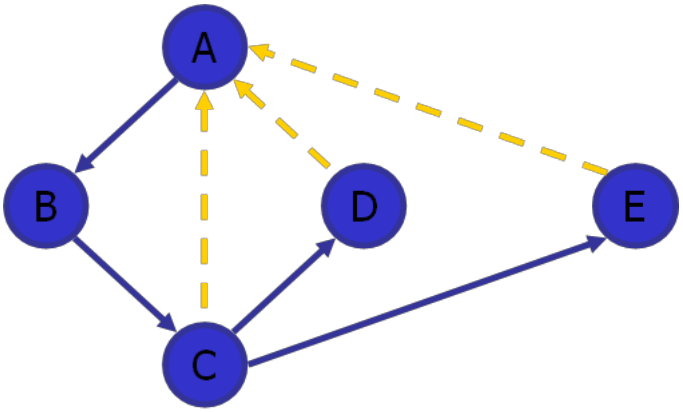
```
// Part I
Algorithm DFS(G,v)
Input: graph G and a start vertex v;
Output: 给以v为端点的连通元素的几条边做标记,前进边DISCOVERY,后退边BACK;
{
    setLabel(v,VISITED);//将起始端点设为VISITED
    for e in G.incidentEdge(v){//循环遍历以v为端点的所有未被访问过的边
        if(getLabel(e)==UNEXPLORED){
            w=opposite(v,e); //找到另一个端点w
            if(getLabel(w)==UNEXPLORED){
                //若该端点也没有被探索过-->将边标签更改DISCOVERY
                setLabel(e,DISCOVERY);
                DFS(G,w);//以w为起点递归这一过程直到遇到的新w已经被探索-->进入e
            }
            else{ //将e标签设置为BACK-->返回边
                    setLabel(e,BACK);
                }
        }
    }
}
```

```
// Part II
Algorithm DFS(G)
Input:graph G
Output: 深度优先遍历, 将G的标签更改为合适的状态
{
//初始化所有端点和边的标签为"UNEXPLORED"
    for u in G.vertices(){
         setLabel(u,UNEXPLORED);
    }
    for e in G.edges(){
    setLabel(e,UNEXPLORED);
    }

    //对每一个未被探索过的端点, 调用之前的Part I算法
    //这样做的原因是图不一定是完全连通的, 从一个端点不一定能探索所有的端点
    for v in G.vertices(){
        if(getLabel(v)==UNEXPLORED) DFS(G,v);
    }
}
```

分析:

第一个算法DFS(G,v)访问给定端点v能到达的所有边和端点,并对边做标记;
DFS(G,v)的DISCOVERY边构成了v的spanning tree;
运行时间$O(n+m)$;



应用

```
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);
  }
```

分析两个pop()的作用:

- 一开始根据递归将点和边加入栈:$\{A, AB, B, BC, C, CD, D, \ldots\}$
- S储存从A到某点的一条路径, 链状
- 如果找到了比如说$A \rightarrow D$, 则返回S的所有元素
- 如果遍历到这条链的头都没找到, 按相反顺序将点和边pop出S: $D \rightarrow CD \rightarrow C \rightarrow \ldots$, 留下起始点A, 对条链进行遍历(如果多条边联结A的话)
- 还是需要注意递归这个过程
- 全过程无环(BACK边不加入S)

```
Algorithm cycleDFS(G, v)
 { 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);
          cycleDFS(G,w);
          S.pop(e); }
      else
        { T = new empty stack
          repeat
            { o = S.pop();
              T.push(o); }
          until (o = w);
          return T.elements(); }
  S.pop(v);
}
```

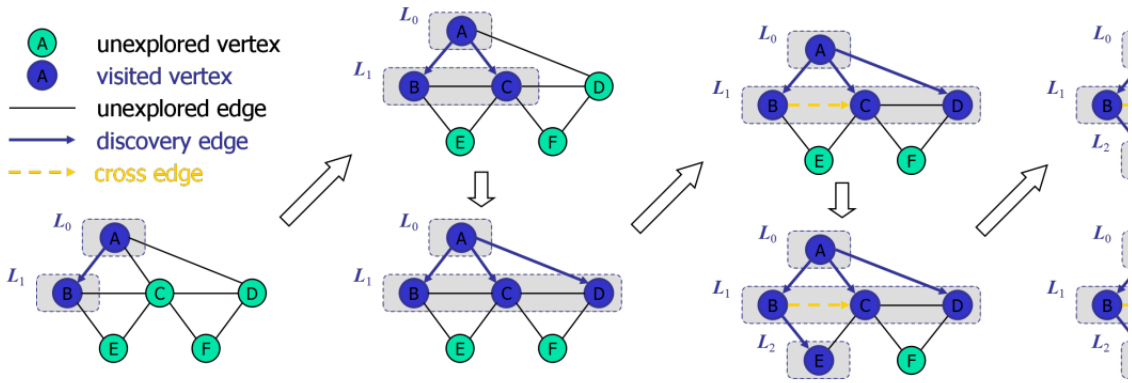
BFS：

```
//Part I
//每一层的端点放一层，通过遍历当前层的端点查找下一层的端点并加入新队列
Algorithm BFS(G,s){//基于队列，和DFS(G,v)相比不用递归
    L0=new empty sequence;
    L0.insertLast(s);
    setLabel(s,VISITED)//起始端点
    i=0;
    while(!Li.isEmpty()){//当前层为空说明已经遍历完毕(上一层为最底层)
        L(i+1)=new empty sequence;//构建一个新序列用于存放下一层的点
        for v in Li.elements(){
            for e in G.incidentEdges(v){
                if(getLabel(e)==UNEXPLORED){//这说明是不同层的端点
                    w=opposite(v,e);
                    if(getLabel(w)==UNEXPLORED){
                        setLabel(e,DISCOVERY);
                        setLabel(w,VISITED); //和DFS的一个不同点：在这里给点加
                        L(i+1).insertLast(w); //将加好标签的点放进新序列(这个新
                    }
                    else setLabel(e,CROSS);//同一层的端点
                }
            }
        }
        i++;
    }
}
```

```
//Part II
Algorithm BFS(G)
Input:graph G;
Output:给G所有的边都加上标签
{
    for u in G.vertices(){
        setLabel(u,UNEXPLORED);
    }
    for e in G.edges(){
```

```
        setLabel(e,UNEXPLORED);
    }
    for v in G.vertices(){
        if(getLabel(v)==UNEXPLORED) BFS(G,v);
    }
}
```

运行时间：$O(n+m)$



Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	

有向图：

If G is simple, m ≤ n*(n-1).

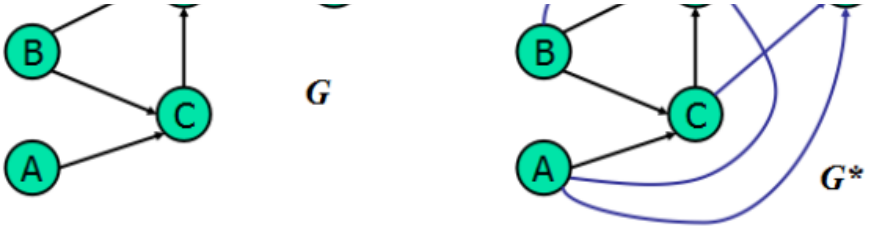
Reachability : 给定端点v,经过有向路径可到达的端点数量

Strong Connectivity： 每个端点都可以到达其余所有端点

```
Algorithm IsStrongConnectivity
{
    Random pick a vertex in G;
    DFS(G,v);
    if (存在未被访问的端点w): return false
    G'=端点与G相同,但所有边的方向相反7. DFS(G',v);
    if (存在未被访问的端点w):return false;
    else: return true;
    }
```

Transitive Closure:

- 传递闭包: 某两个端点之间是否存在一条路径
- 定义 : 给定有向图G , G的传递闭包G*为满足以下条件的有向图
  - G*与G的端点数相同(spanning);
  - 如果uv为G中的连通端点,则在G*中(u -> v)为一条有向边;

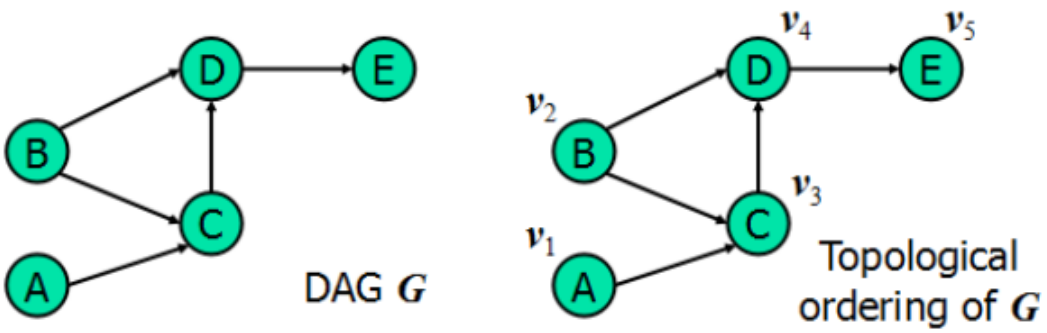


Floyd-Warshall:
若两个端点是连通的, 从一个端点到另一个端点有两种可能:
两端点相邻, 可以直接到达
从一个端点开始, 经过若干端点到另一个端点

Running time: O(n3)

```
Algorithm FloydWarshall(G)
Input:digraph G;
Output:transitive closure G* of G;
{
//将每个端点编号
    i = 1;
    for v in G.vertices(){
        denote v as vi ;
        i = i + 1;
    }
    G0 = G;
    for (k in [1,n]){
        Gk = Gk-1; //新图初始化为和原图一样
        for (i in [1,n]) and (i!=k){
            for (j in [i,n]) and (j!= i,k){ //这两个循环的意思是等价于for i,j
                // 若边(vi,vk),(vk,vj)都在图里,且当前(vk,vj)不在图里, 新增边(vj
                if (G(k-1).areAdjacent(vi,vk)) and (G(k-1).areAdjacent(vk,vj
                    if(!Gk.areAdjacent(vi,vj)){
                        Gk.insertDirectedEdge(vi,vj,k);
                    }
                }
            }
        }
    }
    return Gn;//返回最后一个图
}
```

DAGs and Topological Ordering



- DAG(directed acyclic graph) : 有向无环图,没有cycle的有向图;
- 在DAG中，任何一个端点都没法回到自身

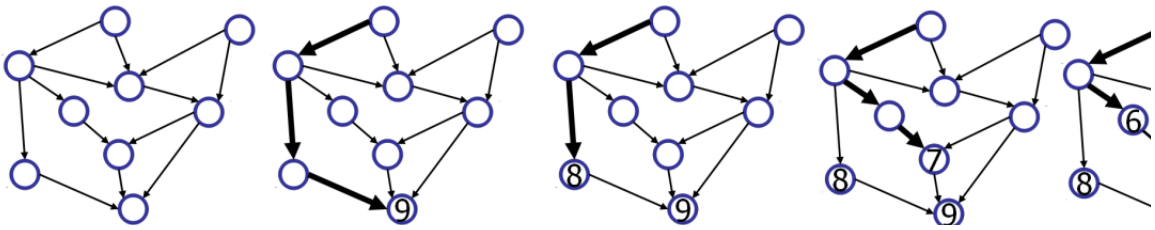
- 拓扑有序：结点编号,对于每条边$(v_i, v_j), i < j$
- 有且仅有DAG才能实现拓扑有序

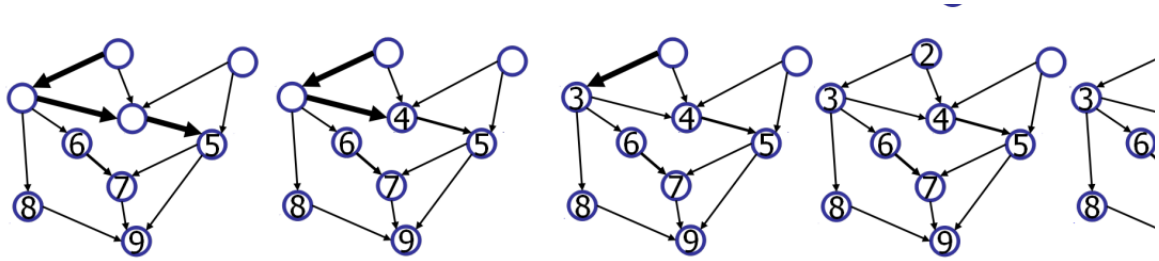
DFS实现拓扑排序 O(n + m).

```
Algorithm topologicalDFS(G,v)
Input:graph G and a start vertex v in G;
Output: 给v所有的connected component加上标签;
{
    setLabel(v,VISITED);
    for e in 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 index n;//v的所有端点都处理完了，赋予v编号n
    n--;
    return;
}
```

```
Algorithm topologicalDFS(G)
Input: digraph G;
Output: topological ordering of G;
{
    n=G.numBertices();
    for u in G.vertices(){
        setLabel(u,UNEXPLORED);
    }
    for e in G.edges(){
        setLabel(e,UNEXPLORED);
    }
    for v in G.vertices(){
        if(getLabel(v)==UNEXPLORED){
            topologicalDFS(G,v);
        }
    }
}
```

栗子：





Weighted Graphs
给每条边都加上权重 : 价格,距离等

Short Path:给定权重图中两个端点u / v ,得到边权重总和最小的路径;
最短路径的子路径也是最短路径;
从起始端点到其他任意端点的最短路径组成一棵树;

Dijkstra Algorithm :
distance(s,v):两端点之间的最短路径;
该算法的目标:给定起始点s,计算s到每个端点vi的距离label D(vi);
预设:
connected graph;
undirected edges;
nonnegative edge weight;
Edge relaxation:
edge e=(u,z) : u是最近被加入输出集合的端点,z还未被加入;
relaxation:以如下形式对D(z)进行更新

$$D(z) = \min\{D(z), D(u) + weight(e)\}$$

```
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 k
  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;
}
```

- 算法分析:
  - 创建优先队列Q:
    - adaptable PQ: $O(n\log n)$;

- bottom-up heap constructure: $O(n)$;

- while循环的每一步:
  - Q.removeMin() : $O(logn)$
  - relaxation : $O(deg(u)logn)$, 对于每个点比较$O(1)$, 在优先队列中更新并更改优先级C
  - while循环总共需要 : $O((n+m)logn)$
- **整个算法: $O((m+n)logn)$, 相当于优先队列中的每个端点都被更新了标签/移出**
- 该算法基于贪婪 : 距离升序增加端点
- 不能有负的边权重

Bellman-Ford Algorithm :

即使边的权重是负的也能运行

要求G是有向图 , 运行时间O(nm) 长于Dijkstra , 但可计算负权重

```
Algorithm BellmanFordDistance(G,s)
{
    for v in G.vertices(){
        if(v==s) D[v]=0;
        else D[v]=infinity;
    }
    for(i=1;i<n;i++){
        for e in G.edges(){
            u=G.origin(e);//有向边的起始端点
            z=G.opposite(u,e);//该边的终止端点
            r=D[u]+weight(e);
            D[z]=min{D[z],r};
        }
    }
}
```

Minimum Spanning Trees:

Spanning tree: 包括了图中所有的点, 边组合可变

MST:找出边权重加和最小的spanning tree;

MST的边的数量为: n-1, 两个端点一条边

Kruskal Algorithm

算法描述: 用于求取MST

假设原图G有v个顶点, e条边

新建图Gnew有v个顶点,但暂时不包括边

将原图G中的e条边按权重进行排序

在Gnew的所有点连通前, 从最小权重边e开始进行以下循环:

若e的两个端点u,v在Gnew中不连通

将e加入Gnew中

最后返回Gnew

```
Algorithm KruskalMST(G){
//每个cloud(v)代表v连通的点, 初始化为v自身
    for v in G.vertices(){
        define a Cloud(v) of {v};
```

```
    }
    T=null;//初始化最小生成树为09. //创建一个优先队列Q，和Prim/Dijkstra不同，该优先队
    Create priority queue Q{
        element:edge e in G;
        key:weight of e;
    }
    while (T.edges().length()<n-1){//注意MST边数为n-1
        //移出权重最小的边并判断两端点是否在同一个类内
        edge e =Q.removeMin();
        u,v=endpoints(e);
        //若不在同一个类内，合并两个云，相当于用e将两个云连通
        if(Cloud(v)!=Cloud(u){
            T.insertLast(e);
            Merge Cloud(v) and Cloud(u);
        }
        //否则就当这条边被扔了，进入下一次循环
    }
    return T;//等到T储存的边数量达到了MST的边数n-1，执行输出
}
```

Running time: O((m+n) log n)

Prim-Jarnik's Algorithm :

Algorithm *PrimJarnikMST(G)*

```
{
  Pick any vertex v of G;
  D[v] = 0;
  for each u ∈ G with u ≠ v do
     D[u] = +∞;
  T = ∅;
  Create a priority queue Q with an entry
  ((u, null), D[u]) for each vertex u, where
  (u, null) is the element and D[u] is the key;
  while Q is not Empty do
   {(u, e) = Q.removeMin();
     add vertex u and edge e to T;
    for each vertex z in Q such that z is adjacent to u do
        if ( w((u, z)) < D[z] )
           { D[z] = w((u, z));
             Change to (z, (u,z)) the element of vertex z in Q;
             Change to D[z] the key of vertex z in Q;
           }
   }
  return T;
}
```

- Graph operation : 对每个端点调用一次 incidentEdges;
- Label operation:
  - set/get z的距离/parent/标签: $O(deg(z))$;
  - set/get a label: $O(1)$;
- PQ operation:
  - 每个端点都被插入一次 + 删除一次: $O(logn)$;
  - 端点w的key每次变化耗时$O(logn)$，变化次数最多$O(deg(w))$;

- 总运行时间: 和Dijkstra类似
  - 如果G的结构是adjacency list structure : $O((n+m)\log n)$;
  - 如果G是连通图 : $O(m\log n)$;

Baruvka's Algorithm :

```
Algorithm BaruvkaMST(G)
  { T = V;  // just the vertices of G
    while T has fewer than n-1 edges do
      for each connected component C in T do
        { Let edge e be the smallest-weight edge from C to another compon
          if e is not already in T then
```

Week12:

Quick select:

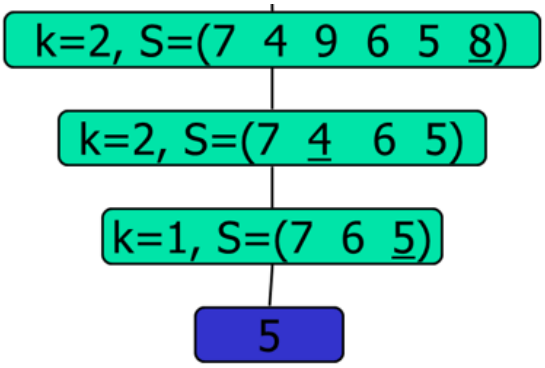
```
Algorithm quickSelect(S, k)
  Input Sequence S of |S| comparable elements and an integer k in [1, |S|]
  Output The k-th smallest element of S
  { if |S|=1
      return the (first) element of S;
    pick a random integer i in [1, |S|]; // |S| is the size of S;
    (L, E, G)=partition(S, i);
    if k < l
      quickSelect(L, k);
    else if k <= |L|+|E|
           return S[i]; // Each element in E is equal to S[i]
         else // Find the k-|L|-|E|-th element in G
           quickSelect(G, k-|L|-|E|);
  }
```

```
Algorithm partition(S, i)
  Input sequence S, index i of the pivot
  Output subsequences L, E, G of the
  elements of S less than, equal to,
  or greater than the pivot, resp.
 { L, E, G = empty sequences;
   x = S[i];
   while ( ¬S.isEmpty() )
     { y = S.remove(S.first());
       if ( y < x )
         L.insertLast(y);
       else if ( y = x )
         E.insertLast(y);
       else // y > x
         G.insertLast(y); }
   return L, E, G;
  }
```

14

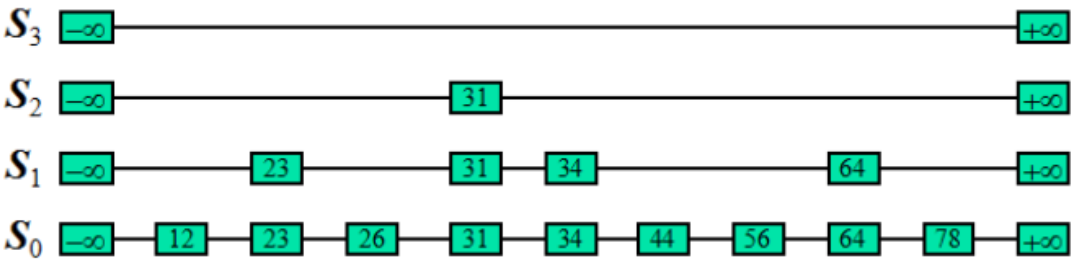
Running time:O(n)

k=5, S=(7 4 9 3 2 6 5 1 8)



Skip lists

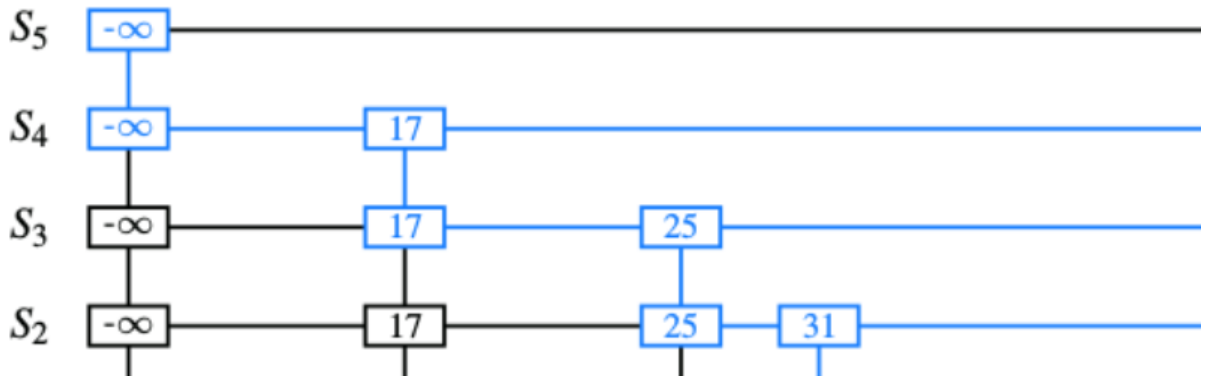
- Skip list : 含有不同数量键值对的层 $S_0, S_1, \ldots S_h$
  - 第一层S_0包含所有元素
  - 每层都是有序链表
  - 每个S_i包含special keys(sentinels) $+\infty, -\infty$
  - 每个list是上一个list的子序列 : $S_h \in S_{h-1} \ldots \in S_1 \in S_0$, 若某个元素出现再第i层,则所有比i小的
  - top list(最后一个序列) S_h只含两个特殊keys $+\infty, -\infty$;

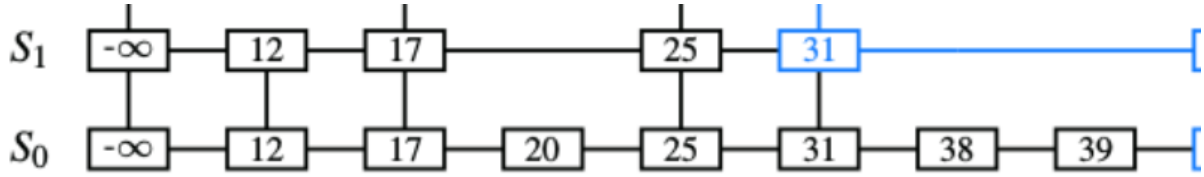


- 基本移动操作 $O(1)$: 假定p为S_1中的31
  - next(p):与p相同level的下一个位置(S_1中的34)
  - prev(p):与p相同level的上一个位置(S_1中的23)
  - above(p):上一个level的相同位置(S_2中的31)
  - below(p):下一个level的相同位置(S_0中的31)

Search:

- We start at the first position of the top list
- At the current position p, we compare x with $y \neg key(next(p))$



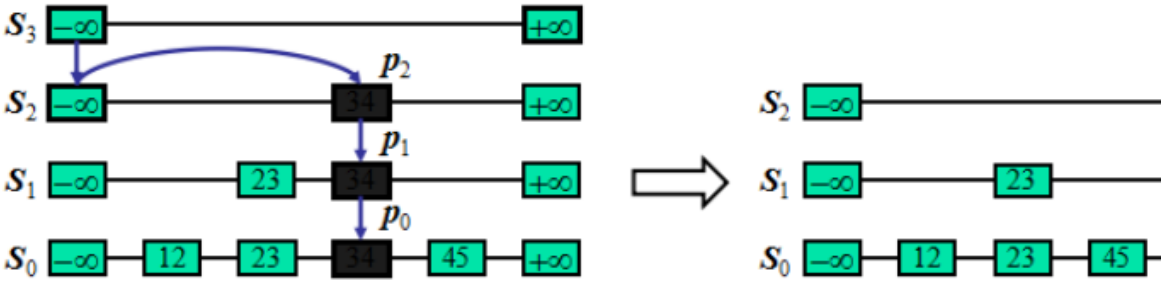


Insertion:

- 插入操作需要用到随机化算法决定插入最后插入的层
  - 类似抛硬币 ,重复抛直到得到背面 ,在这期间得到正面的次数为i
  - S_i为新元素被插入的最上层level
  - 确定S_i:
    - if(i>h) 增加一些只含$+\infty, -\infty$的空序列 $S_{h+1}, \ldots S_i$;
    - else 排除多余的list $S_{i+1}, \ldots S_h$, 最后剩下$S_0, S_1, \ldots S_i$;
  - 使用SkipSearch(k)查询是否已经存在entry:
    - 如果已经存在,return p并将value替换为v;
    - 如果不存在,返回的是小于k的最大key所在位置 p0,p1,...pi;
  - 将新entry(k,v)插入到p0,p1,...pi后面
- 举个栗子 : i=2,插入15

Deletion

- SkipSearch(k),从最高层开始寻找删除点p0,p1,...pi;
- 从每个list Sj中删除相应位置pj;
- 若删除点位于最大层, 需要对层数进行更新(多个只有最大最小边界的层最终只保留一个)
- 举个栗子: 删除34



平均空间复杂度 O(n)
高度O(logn)
搜索操作的时间复杂度O(log n),插入和删除操作差不多