Visualization

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Red-Black Tree

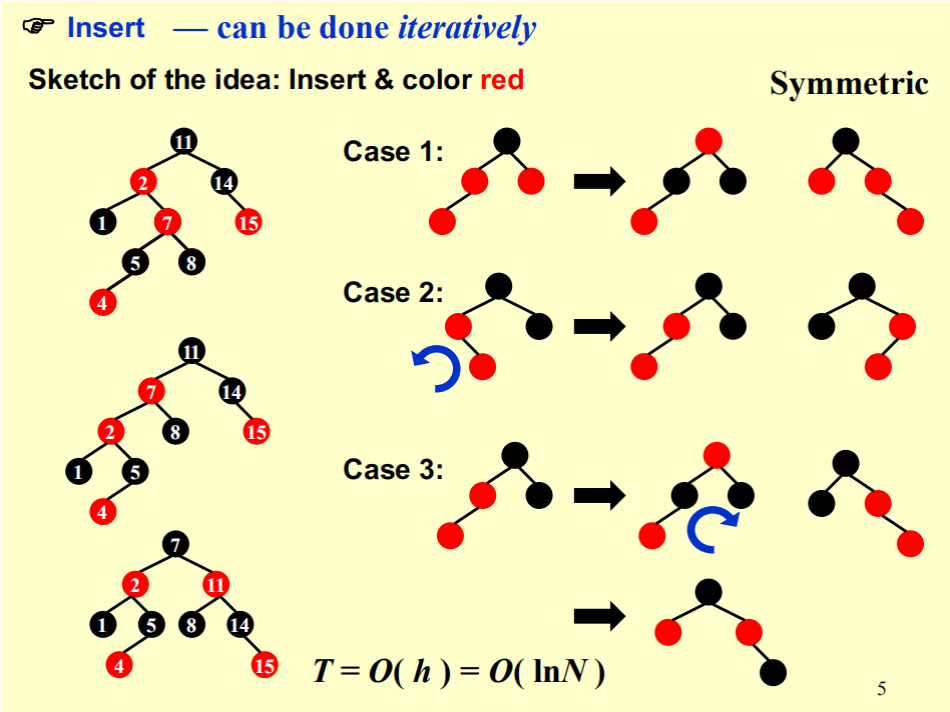
Property

- Every node is either red or black.
- The root is black.
- Every leaf (NIL) is black.
- If a node is red, then both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

Lemma:

- A red-black tree with n internal nodes has height at most 2lg(n+1)
- A subtree rooted at node x contains at least $2^{bh(x)-1}$ internal nodes

Insert



The following properties always hold:

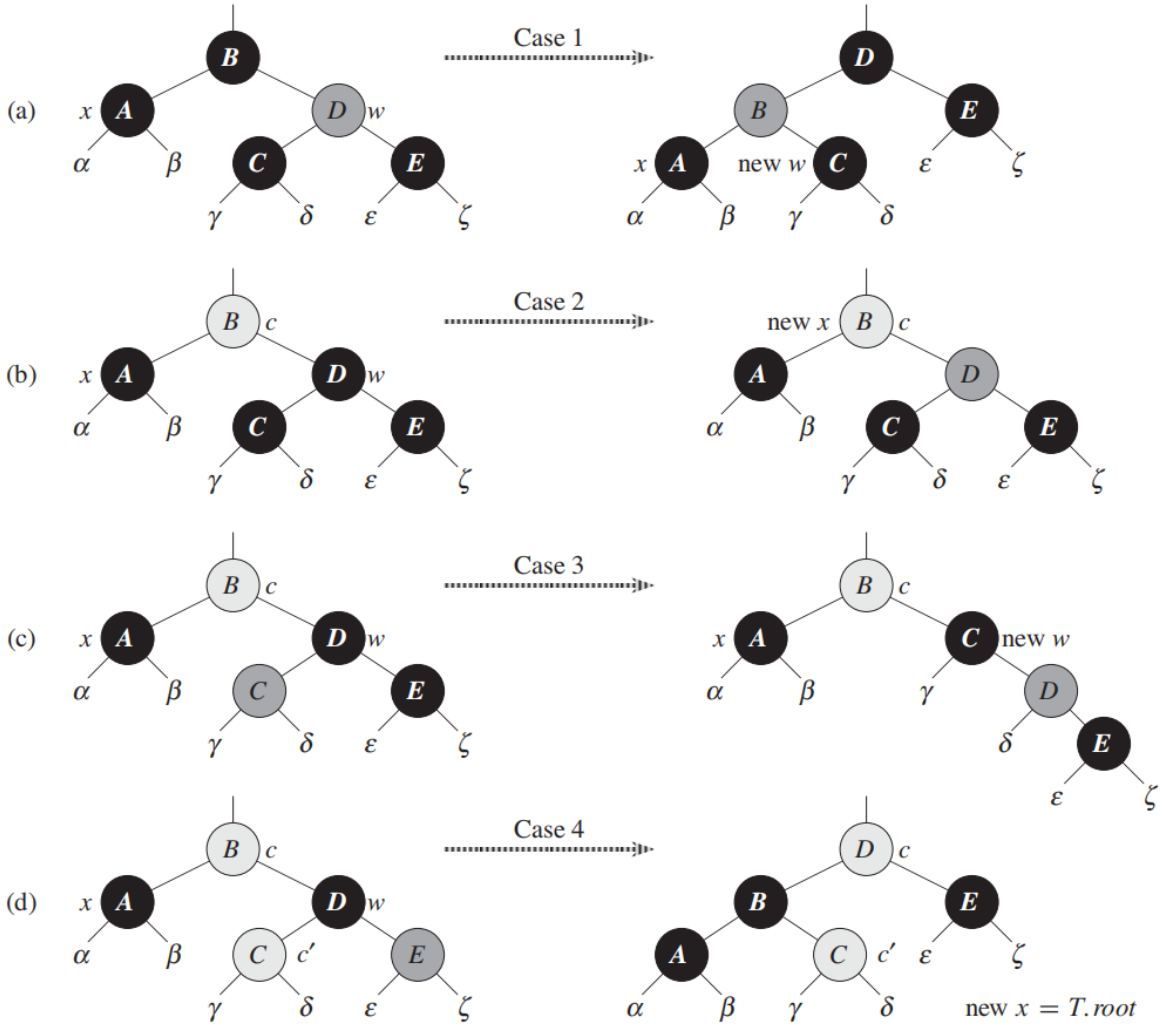
- Node z is red.
- If $z.p$ is the root, then $z.p$ is black.
- At most one black-red property may be damaged, either property (2) or (4)
- The process only goes on after case 1 and terminates after case 3

彻底理解红黑树（二）之 插入

Delete

If the root is to be deleted, simply remove the root. (the tree has only a root)

When x is not the root, we want to remove x (in fact x here should be the child of the node that is actually deleted):



case 1 (x's sibling is red)$\longrightarrow$ 2/3/4

case 2 (sibling is black and both children are black)

- push up: parent is black
- terminate: parent is red

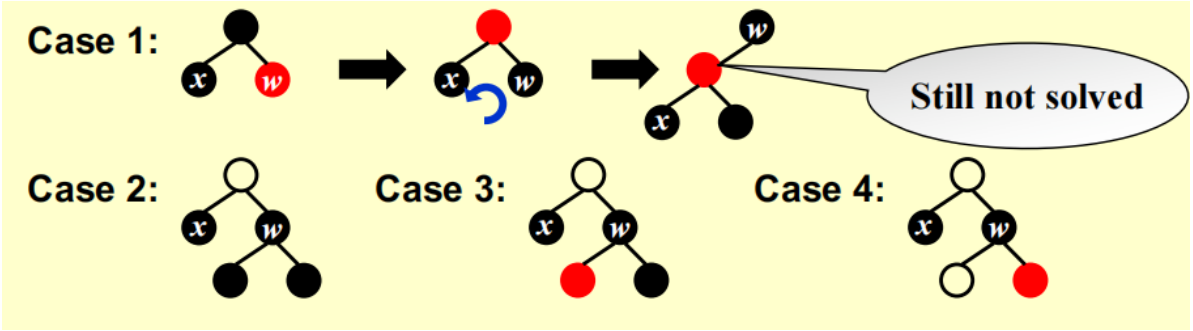
case 3 (sibling is black and the inner children is red but the outer one is black)

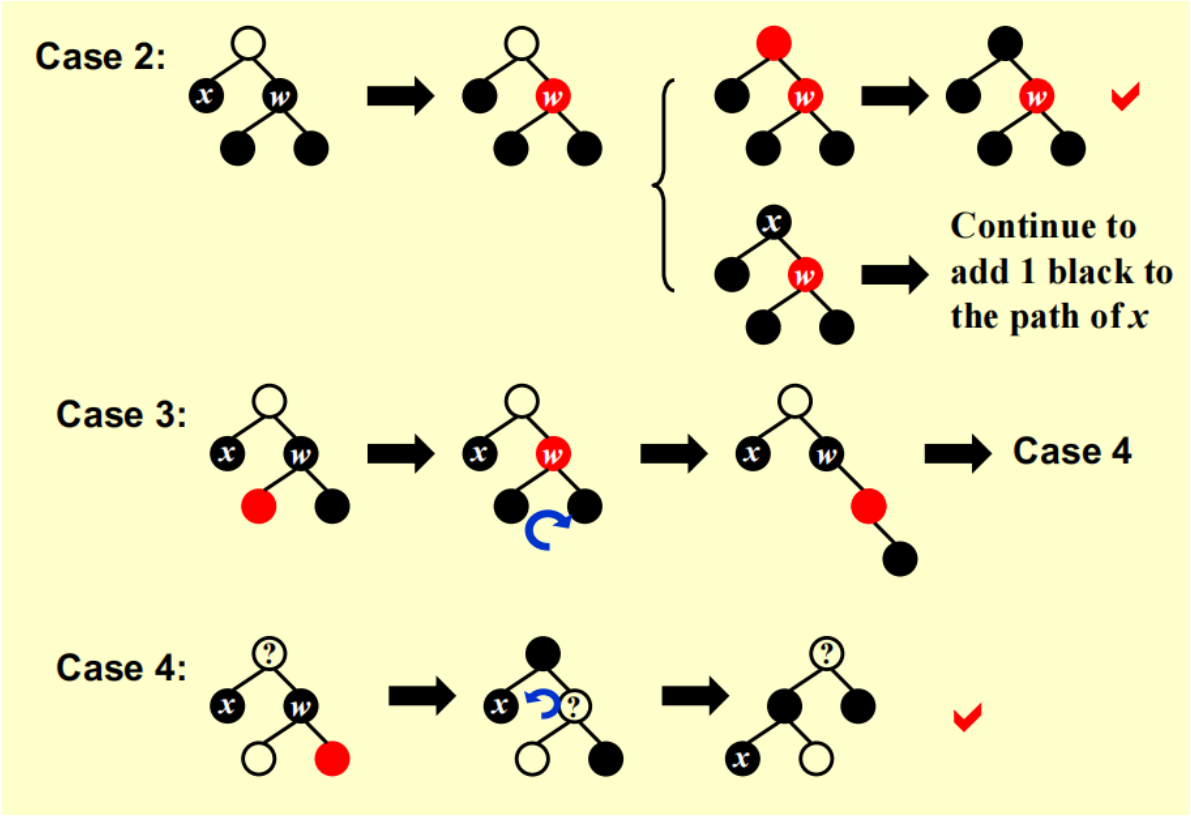
- 4: rotate and color

case 4 (sibling is black and the outer children is red)

- terminate: rotate and color

彻底理解红黑树（三）之 删除





AA-Tree

- To simplify a red-black tree
- Similar performance
- A red node can only be a right child
- *level* is exactly *black height* of a red-black tree

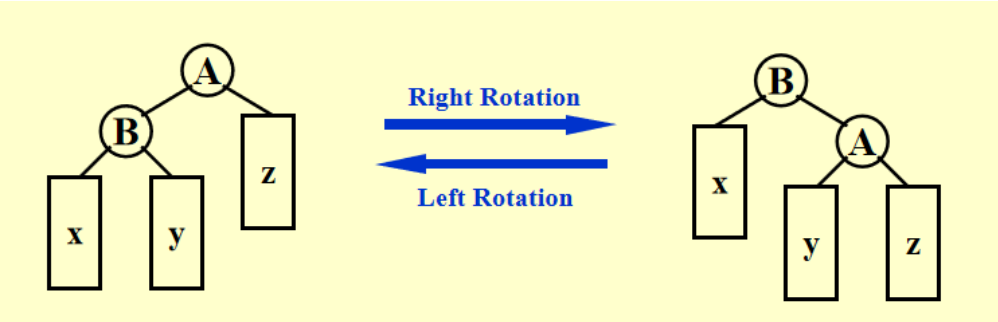
reference 1

reference 2

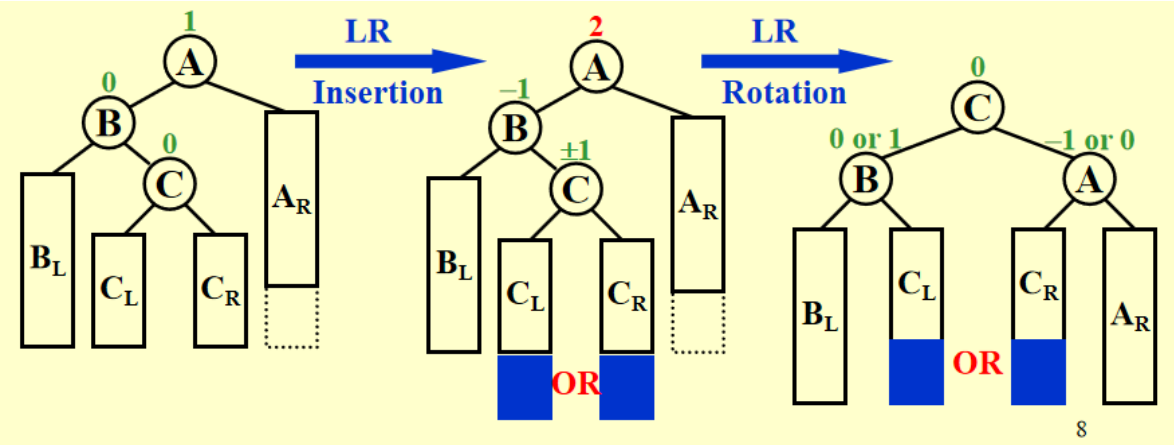
AVL

Rotation

Single rotation:



Double rotation:



Insertion & Deletion

先按照BST的方式插入，然后再维护

- 自底向上寻找第一个违反定义的点
- 从该点往新插入的点方向走，看走的是直线还是折线

先按照BST的方式删除了，然后再维护

- 自底向上寻找第一个违反定义的点设为A
- 设被删除点在A的子树B内，则设其兄弟为C。如果A和C的平衡因子异号，则double rotation；否则single rotation

Splay

插入：和二叉搜索树一样操作，不同的是每次要把新加入的元素旋转到根结点

删除：

- `Find X;` (Rotate)
- `Remove X;`
- `FindMax(TL);` (Rotate)
- `Make TR the right child of the root of TL`

Rotation:

- zig: the same as AVL
- zig-zig: single rotate the **grandparent**, then single rotate the **parent**

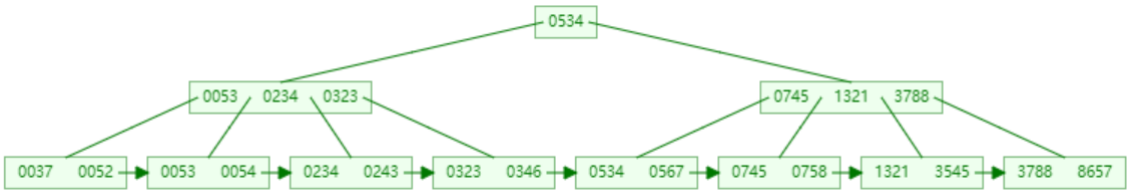
- zig-zag: double rotate, the same as AVL

B+ (2-3-4)

Definition:

1. The root is either a leaf or has between 2 and M children
2. All non-leaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children
3. All leaves are at the same depth, containing $\lceil M/2 \rceil$ to M keys

> Adopted from B-trees, but more suitable for external storage index structure
>
> A page is almost fully used for indexing, thus a tree of height 3 could index more data than a B-tree s maximizing the branching factor of the internal nodes



Depth: $O(\lceil \log_{\lceil M/2 \rceil} N \rceil)$

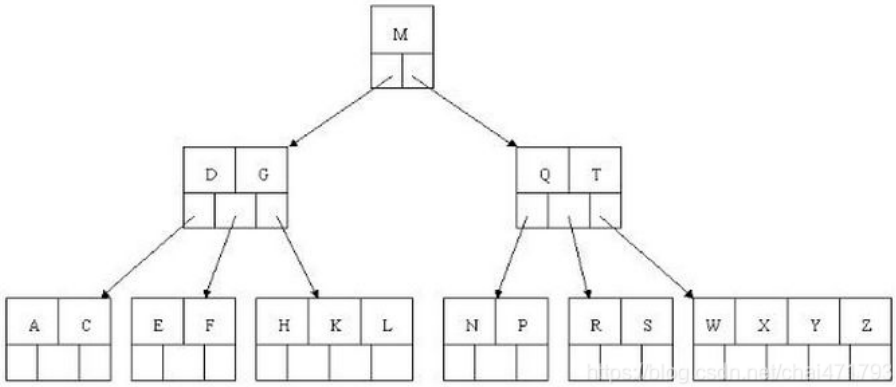
Time complexity for splitting one node: $O(M)$, thus insertion:
$T(M,N) = O(M) \cdot O(\lceil \log_{\lceil M/2 \rceil} N \rceil) = O(M \frac{\log N}{\log M})$

Time complexity for (binary) searching in one node: $O(\log M)$, thus finding:
$T_{find}(M,N) = O(\log M) \cdot O(\lceil \log_{\lceil M/2 \rceil} N \rceil) = O(\log N)$

> The best choice of M is 3 or 4

B-Trees

- The origin of B+trees
- When the data is of large structure, a page may be filled **majorly with the data(*satellite information*) but not *keys* for indexing**, which may slow down the speed
- Well compatible with disks
- Since in most systems the running time of a B-tree algorithm depends primarily on the number of *DISK-READ* and *DISK-WRITE* operations it performs, we typically want each of these operations to read or write as much information as possible. Thus, a B-tree node is usually **as large as a whole disk page**, and this size limits the number of children a B-tree node can have.
- For a large B-tree stored on a disk, we often see **branching factors between 50 and 2000**, depending on the size of a key relative to the size of a page
- Insertion can go in a single-pass-down way(for internal nodes contain data too), which happens in the leaf level at first



Insertion

1. Find the position
2. Insert the new key
3. If the leaf is full:
    - split the leaf
    - push the new key to a sibling
4. Check upwards

```
Btree  Insert ( ElementType X,  Btree T )
{
   Search from root to leaf for X and find the proper leaf node;
   Insert X;
   while ( this node has M+1 keys ) {
         split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
   respectively;
         if (this node is the root)
                create a new root with two children;
         check its parent;
   }
}
```

$T(M, N) = O((M/\log M) \log N)$

Deletion

1. Find the key
2. Remove the key
3. If the leaf breaks definition 2:
    - merge 2 leaves
    - get one key from a sibling
4. Check upwards

Amortized Analysis

Usually: worst >= amortized >= average

Aggregate Analysis

For a sequence of n operations, the worst time is T, so the amortized time is $\frac{T}{n}$

Idea: Show that for all n, a sequence of n operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore $T(n)/n$.

〖Example〗 Stack with MultiPop(int k, Stack S)

```
Algorithm {
   while ( !IsEmpty(S) && k>0 ) {
      Pop(S);
      k - -;
   } /* end while-loop */
}
```

$T = \min (\text{sizeof}(S), k)$

Consider a sequence of n Push, Pop, and MultiPop operations on an initially empty stack.

$$\text{sizeof}(S) \le n$$

$$T_{amortized} = O(n)/n = O(1)$$

Accounting Method

- design the amortized cost (amortized credit for each operation), the closer to the actual bound the better
- not allowed to pay on credit $\sum \hat{c}_i \ge \sum c_i$
- save the credit for each pay
- For all sequences of n operations, we must have $\sum \hat{c}_i \ge \sum c_i$
- $T_{amortized} = \frac{\sum \hat{c}_i}{n}$

〖Example〗 Stack with MultiPop(int k, Stack S)

c_i for Push: 1 ; Pop: 1 ; and MultiPop: $\min (\text{sizeof}(S), k)$

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: +1 ; Pop: –1 ; and MultiPop: –1 for each +1

$\text{sizeof}(S) \ge 0 \Rightarrow$ *Credits* ≥ 0

$$\Rightarrow O(n) = \sum_{i=1}^{n} \hat{c}_i \ge \sum_{i=1}^{n} c_i$$

$$\Rightarrow T_{amortized} = O(n)/n = O(1)$$

Potential Method

- design a potential function to signal the potential (credit)
- In general, a good potential function should always assume its **minimum at the start** of the sequence. (or the operation may pay on credit)
- The potential **gains in drops for cheap operations** and may **release tremendously for expensive operations**
- for each operation, amortized cost: $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- for the whole: $\sum \hat{c_i} = \sum c_i + \Phi(D_n) - \Phi(D_0)$

〖Example〗 Stack with MultiPop(int k, Stack S)

D_i = the stack that results after the i-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)+1) - sizeof(S) = 1$

$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)-1) - sizeof(S) = -1$

$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1-1 = 0$

MultiPop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)-k') - sizeof(S) = -k'$

$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k'-k' = 0$

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} O(1) = O(n) \geq \sum_{i=1}^{n} c_i \Rightarrow T_{amortized} = O(n)/n = O(1)$$

Inverted File Index

reference

Definition:

Inverted file contains a list of pointers to all occurrences of that term in the text, like

No.	Term	<Times; (Documents;Index in text)>
1	a	<3; (1;2), (2;5), (4;16)>
2	silver	<2; (4;11), (5;77)>
...		

Pro:

- Very fast algorithm that is easy to implement.

Con:

- Doesn't work at all if you run out of main memory. Once memory runs out you start swapping.
- For small document collections this is a great algorithm, for anything realistic it requires **a lot of memory**.

Read a term

1. word stemming
    - process the word to gain its stem or root form
2. stop words
    - some words are too common that they don't help searching
    - a table is used to record stop words
    - Query Threshold
    - vary for different engines

Query Threshold

1. Sort the terms queried by their frequencies across the collection
2. Define a threshold as the percentage of terms taken in the original query in a newly created reduced query.

Pro

- Avoids large posting lists
- Dramatic savings on efficiency when large posting list is not retrieved
- Effectiveness does not degrade (as long as we do not threshold too much) because we are omitting only those terms with long posting lists

Con

- Still can have some very long posting lists
- Inappropriate threshold may degrade the accuracy

Store the table

1. Term-partitioned index
    - A~C
    - D~F
    - ...
2. Document-partitioned index
    - 1~10000
    - 10001~20000
    - ...

Dynamic Indexing

Double buffer:

- Maintain two index tables
- One for reading only and one for writing only
- Adding and deleting occurs to the table for writing
- Periodically renew the index for reading

Temporary Index:

> 在内存中实时建立的倒排索引，当有新文档进入系统时，实时解析文档并将其追加进这个临时索引结构中

Deleted List:

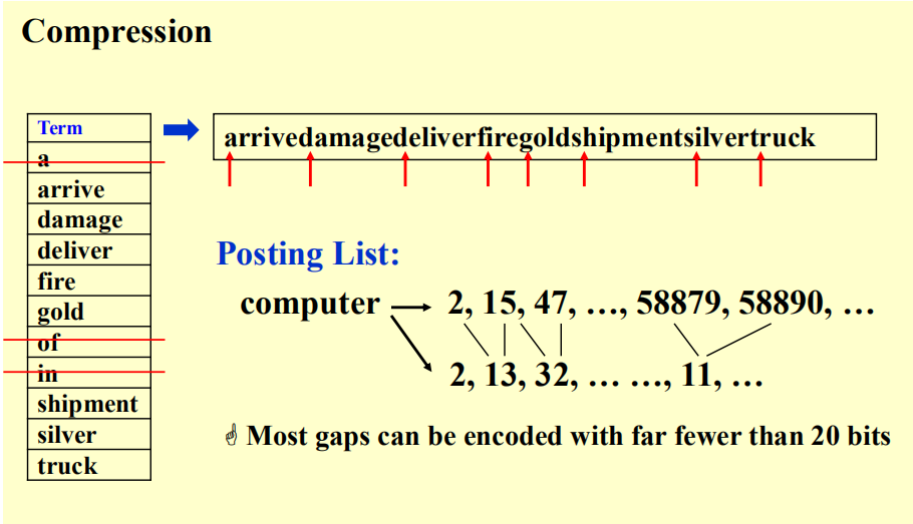
> 存储已被删除的文档的相应文档ID，形成一个文档ID列表。当文档被修改时，可以认为先删除旧文档，然后向系统增加一篇新文档，通过这种间接方式实现对内容更改的支持

Strategy of updating:

> 当系统发现有新文档进入时，立即将其加入临时索引中。有新文档被删除时，将其加入删除文档队列。文档被更改时，则将原先文档放入删除队列，解析更改后的文档内容，并将其加入临时索引。这样就可以满足实时性的要求。

- Complete Re-Build
- Re-Merge
    - when new documents accumulate to a certain number or memory is full
    - Efficiency: When traversing the old inverted index, we can read the contents of the file in order **which reduces the disk seek time** because we have arranged the order from low to high according to the lexicographic order of the terms
    - Disadvantage: **I/O costs a lot** because those posting lists unchanged need to be read and written into the new index
- In-Place
    - Available space needs to be saved, which requires to maintain and manage the available disk space and **costs a lot**
    - When data is transferred (especially some terms are deleted), the order of the terms and the posting lists needs to managed
- Hybrid
    - Classify the terms and adopt different strategies
    - Terms with long posting lists: In-place
    - Terms with short posting lists: Re-merge

Compression



Delta Compression

The structure should efficiently store each term:

- The list of documents that contain the term
- For each occurrence of a term in a document
  - The frequency the term appears in the document (tf)
  - The position in the document for which the term appears (only needed if proximity queries will be supported), which may be expressed as section, paragraph, sentence, location within sentence

- I/O to read a posting list is reduced if the inverted index takes less storage
- Stop words eliminate about half the size of an inverted index. “the” occurs in 7 percent of English text.
- Other compression
  - Posting List
  - Term Dictionary
- Half of terms occur only once *(hapax legomena)* so they only have one entry in their posting list
- Problem is some terms have very long posting lists -- in Excite’s search engine 1997 occurs 7 million times.

To compress:

- Term name in the term list
- Term Frequency in each posting list entry
  - Elias Encoding
- Document Identifier in each posting list entry
  - Generalized Elias Encoding
  - Bitmap Compression
  - Delta Compression
- Data
  - Applied to posting lists, each term: $(d_1, t_{f1}), (d_2, t_{f2}), \ldots (d_n, t_{fn})$
  - Documents are ordered, so d_i is replaced by the interval difference

    - Numbers are encoded using fewer bits for smaller, common numbers
    - Index is reduced to 10-15% of database size

Pro

- Can reduce I/O for query of inverted index.
- Reduce storage requirements of inverted index.

Con

- Takes longer to build the inverted index.
- Software becomes *much* more complicated.
- Uncompress required at query time -- note that this time is usually offset by dramatic reduction in I/O.

Top Docs

- Instead of retrieving the whole posting list, we might want to only retrieve the top *x* documents where the documents are ranked by weight.
- A separate structure with sorted, truncated posting lists may be produced.

Pro

- Avoids need to retrieve the entire posting list
- Dramatic savings on efficiency for large posting lists

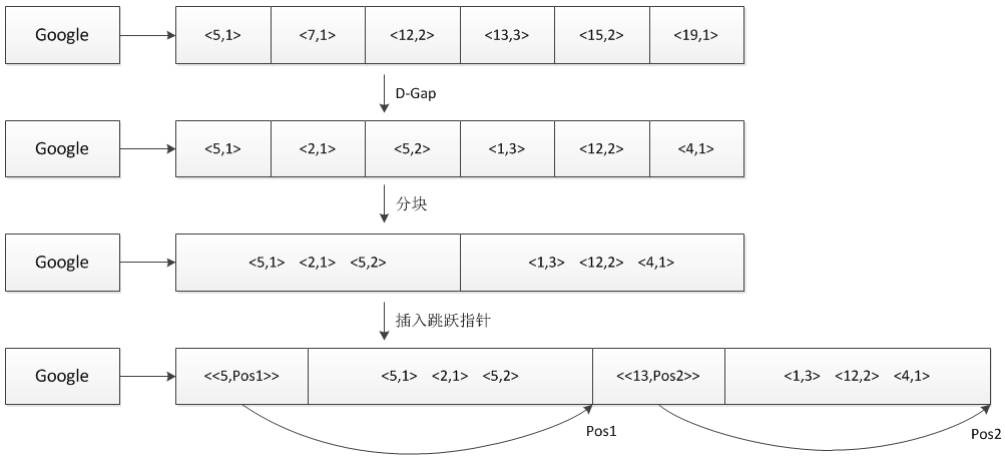
Con

- Not feasible for Boolean queries
- Can miss some relevant documents due to truncation

Query

- Doc at a Time
    - Calculate the score of a document at a time
- Term at a Time
    - Calculate the score of all documents on one queried term at a time
- Skip Pointers

> 基本思想：将一个倒排列表数据化整为零，切分为若干个固定大小的数据块，一个数据块作为一组，对于每个数据块，增加元信息来记录关于这个块的一些信息，这样即使是面对压缩后的倒排列表，在进行倒排列表合并的时候也能有两个好处：
>
> 1、无须解压所有倒排列表项，只解压部分数据即可
>
> 2、无须比较任意两个文档ID。
>
> 下图是将“Google”这个查询词对应的倒排列表加入跳跃指针后的数据结构。



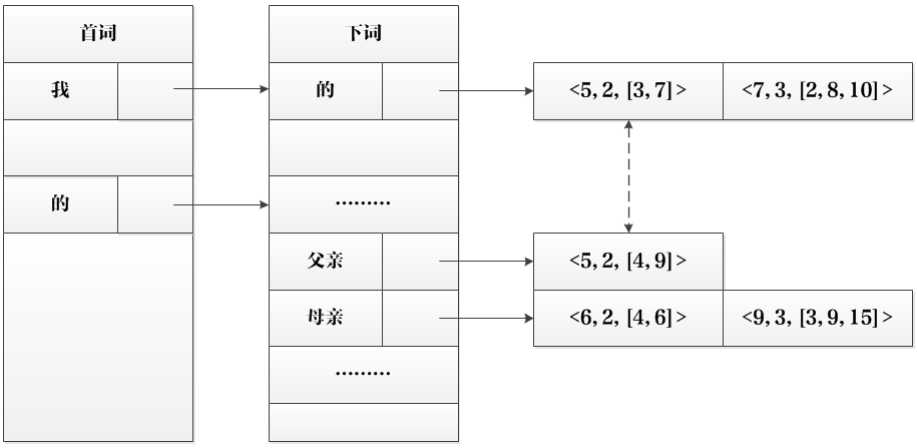
假设对于“Google”这个单词的倒排列表来说，数据块的大小为3。然后在每块数据前加入管理信息，比如第一块的管理信息是<<5,Pos1>>，5表示块中第一个文档ID编号，Pos1是跳跃指针，指向第2块的起始位置。假设要在单词“Google"压缩后的倒排列表里查找文档ID为7的文档。首先，对倒排列表前两个数值进行数据解压缩，读取第一组的跳跃指针数据，发现其值为<5,Pos1>，其中Pos1指出了第2组的跳跃指针在倒排列表中的起始位置，于是可以解压缩Pos1位置处连续两个数值，得到<13,Pos2>。5和13是两组数据中最小的文档ID（即每组数据的第一个文档ID），我们要找的是7，那么如果7号文档包含在单词”Google“的倒排列表中的话，就一定会出现在第一组，否则说明倒排列表中不包含这个文档。解压第1组数据后，根据最小文档编号逆向恢复其原始的文档编号，此处<2,1>的原始文档ID是：5+2=7，与我们要找的文档ID相同，说明7号文档在单词”Google“的倒排列表中，于是可以结束这次查找。

从上面的查找过程可知，在查找数据时，只需要对其中一个数据块进行解压缩和文档编号查找即可获得结果，而不必解压所有数据，很明显加快查找速度，并节省内存空间。

缺点：增加指针比较操作的次数。

实践表明：假设倒排列表的长度为L（即包含L个文档ID），使用根号L作为块大小，则效果较好。

Phrase Query

- Position Index
  - compare the positions of the queried terms to check whether they are adjacent
- Next-word Index
  - 

  - drastically increase the index table, thus only built for those words needing costly computation
- Phrase Index
  - build index for a phrase just like a single word
  - frequently used

- Hybrid
  - phrase index $\rightarrow$ next-word index $\rightarrow$ position index

Measurement

- How fast does it index
  - Number of documents/hour
- How fast does it search
  - Latency as a function of index size
- Expressiveness of query language
  - Ability to express complex information needs
  - Speed on complex queries
- User happiness
  - Data Retrieval Performance Evaluation (after establishing correctness)
    - Response time
    - Index space
  - Information Retrieval Performance Evaluation
    - How relevant is the answer set?
    - Precision: $P = \frac{R_R}{R_R+I_R}$ (The percentage of relevant information to retrieved information)
    - Recall: $R = \frac{R_R}{R_R+R_N}$ (The percentage of retrieved information to relevant information)

	Relevant	Irrelevant
Retrieved	R_R	I_R
Not Retrieved	R_N	I_N

Inversion (Build Index)

1. parse and build temp file
   - for each document: Parse text into terms, assign a term to a termID (use an internal *index* for this)
   - for each term: Write an entry to a temporary file with only triples <termID, docID, tf>
2. make sorted *runs*, to prepare for merge
   Do Until End of Temporary File:
   - Read as much of the temp file that will fit into memory
   - Sort the triples in memory
   - Write them out in a sorted run
3. merge the runs
   - Repeat until there is only one *run*
   - Merge two sorted pairs of runs into a single run
4. for each distinct term:
   - Read all triples for a given term (these will be in sorted order)

- Build the posting list (feel free to use compression)
- Write this to the inverted index

Pro:

- Not as fast as memory based, but at least is usable. Sort and merge time dominates cost.

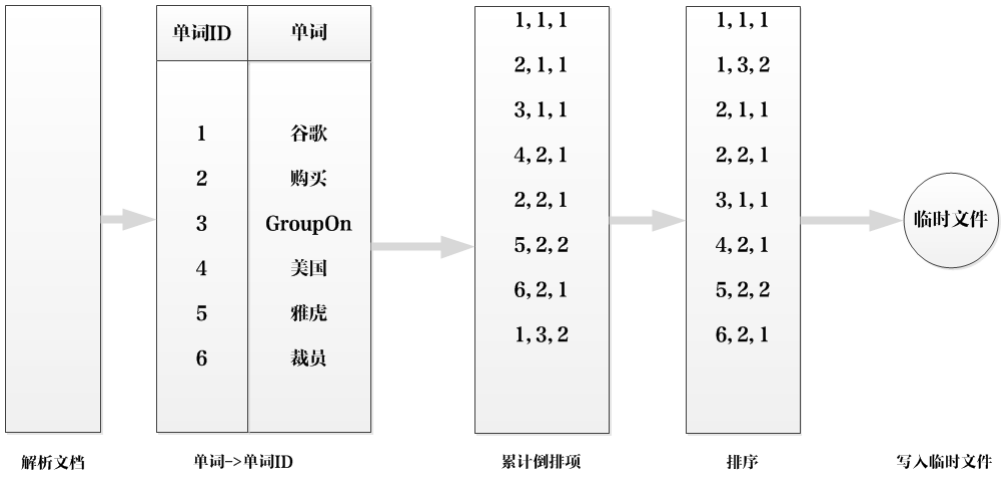
Con:

- Requires twice the amount of disk space as the size of the original text.

2-Pass In-Memory Inversion:

> 要求内存要足够大。
> **第一遍**
> 收集一些全局的统计信息。包括文档集合包含的文档个数N，文档集合内所包含的不同单词个数M，每个单词在多少个文档中出现过的信息DF。
> 将所有单词对应的DF值全部相加，就可以知道建立最终索引所需的内存大小是多少。
> 获取信息后，根据统计信息分配内存等资源，同时建立好单词相对应倒排列表在内存中的位置信息。
>
> **第二遍**
> 逐个单词建立倒排列表信息。获得包含某个单词的每个文档的文档ID，以及这个单词在文档中的出现次数TF，然后不断填充第一遍扫描时所分配的内存。当第二遍扫描结束的时候，分配的内存正好被填充满，每个单词用指针所指向的内存区域"片段"，其起始位置和终止位置之间的数据就是这个单词对应的倒排列表。

Sort-based

> 在建立索引过程中，始终在内存中分配固定大小的空间，用来存放词典信息和索引的中间结果，当分配的空间被消耗光的时候，把中间结果写入磁盘，清空内存里中间结果所占空间，以用做下一轮存放索引中间结果的存储区。参考下图：
>
> 
>
>
> 上图是排序法建立索引中间结果的示意图。建立过程：
> 读入文档后，对文档进行编号，赋予唯一的文档ID，并对文档内容解析；
> 将单词映射为单词ID；
> 建立（单词ID、文档ID、单词频率）三元组；
> 将三元组追加进中间结果存储区末尾；
> 然后依次序处理下一个文档；

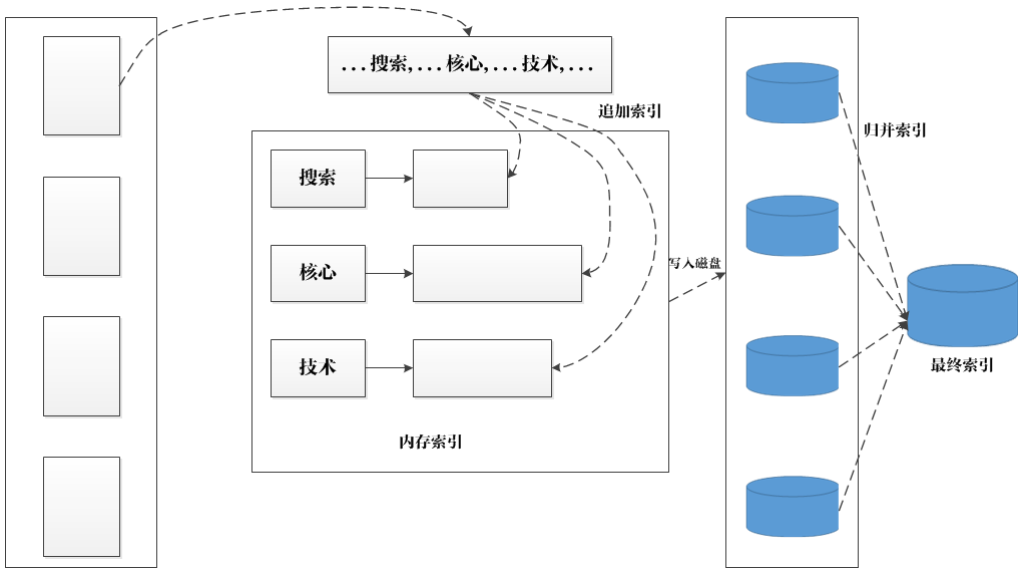
当分配的内存定额被占满时，则对中间结果进行排序（根据单词ID->文档ID的排序原则）；
将排好序的三元组写入磁盘文件中。

注：在排序法建立索引的过程中，词典是一直存储在内存中的，由于分配内存是固定大小，渐渐地词典占用内存越来越大，那么，越往后，可用来存储三元组的空间越来越少。

建立好索引后，需要合并。
合并时，系统为每个中间结果文件在内存中开辟一个数据缓冲区，用来存放文件的部分数据。将不同缓冲区中包含的同一个单词ID的三元组进行合并，如果某个单词ID的所有三元组全部合并完成，说明这个单词的倒排列表已经构建完成，则将其写入最终索引中，同时将各个缓冲区中对应这个单词ID的三元组内容清空。缓冲区继续从中间结果文件读取后续的三元组进行下一轮合并。当所有中间结果文件都依次被读入缓冲区，并合并完成后，形成最终的索引文件。

Merge-based

归并法与排序法类似，不同的是，每次将内存中数据写入磁盘时，包括词典在内的所有中间结果都被写入磁盘，这样内存所有内容都可以被清空，后续建立索引可以使用全部的定额内存。归并法的示意图如下所示：



与排序法的差异：
1、排序法在内存中存放的是词典信息和三元组数据，词典和三元组数据并没有直接的联系，词典只是为了将单词映射为单词ID。归并法则是在内存中建立一个完整的内存索引结构，是最终文章索引的一部分。
2、在将中间结果写入磁盘临时文件时，归并法将这个内存的倒排索引写入临时文件，随后彻底清空所占内存。而排序法只是将三元组数据排序后写入磁盘临时文件，词典作为一个映射表一直存储在内存中。
3、合并时，排序法是对同一单词的三元组依次进行合并；归并法的临时文件则是每个单词对应的部分倒排列表，所以在合并时针对每个单词的倒排列表进行合并，形成这个单词的最终倒排列表。

Leftist Heap

The null path length, Npl(X), of any node X is the length of the **shortest path** from X to a node without two children. Define $Npl(NULL) = -1$

Npl(X) = min { Npl(C) + 1 for all C as children of X }

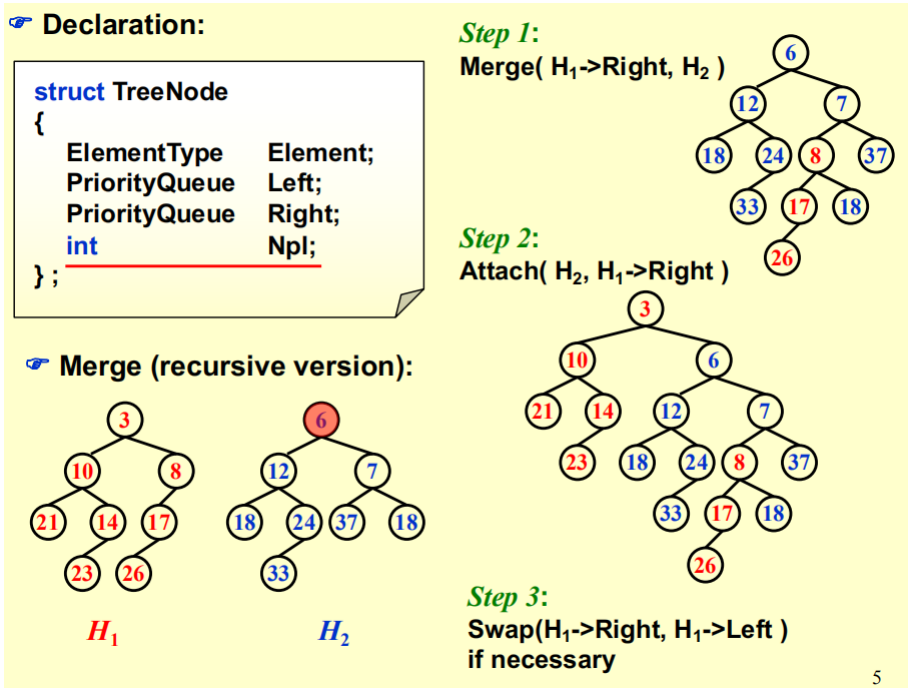
Leftist heap:

- The leftist heap property is that for every node X in the heap, the null path length of the left child is **at least as large as** that of the right child
- A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes
  - right path: always go right until there is no right children, $\leq \lfloor \log(N+1) \rfloor$
  - The right path is the shortest
  - The right path of left subtree is no less than the one of right subtree (?)
- Insertion is merely a special case of merging.

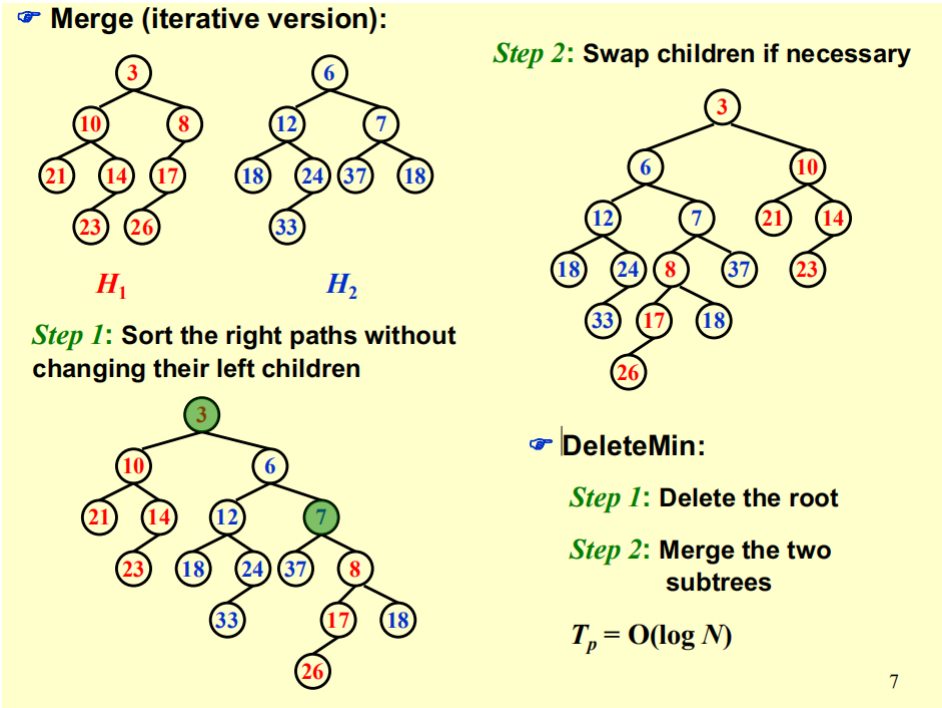
Recursion

Recursive pseudo-code: $T_p = O(\log N), N = \text{total nodes}$

```
PriorityQueue Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL ) return H2;
    if ( H2 == NULL ) return H1;
    if ( H1->Element < H2->Element ) return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}
// H1->root < H2->root
static PriorityQueue Merge1( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1->Left == NULL ) /* single node */
        H1->Left = H2; /* H1->Right is already NULL and H1->Npl is already 0 */
    else {
        H1->Right = Merge( H1->Right, H2 ); /* Step 1 & 2 */
        if ( H1->Left->Npl < H1->Right->Npl )
            SwapChildren( H1 ); /* Step 3 */
        H1->Npl = H1->Right->Npl + 1;
    } /* end else */
    return H1;
}
```



Iteration & DeleteMin

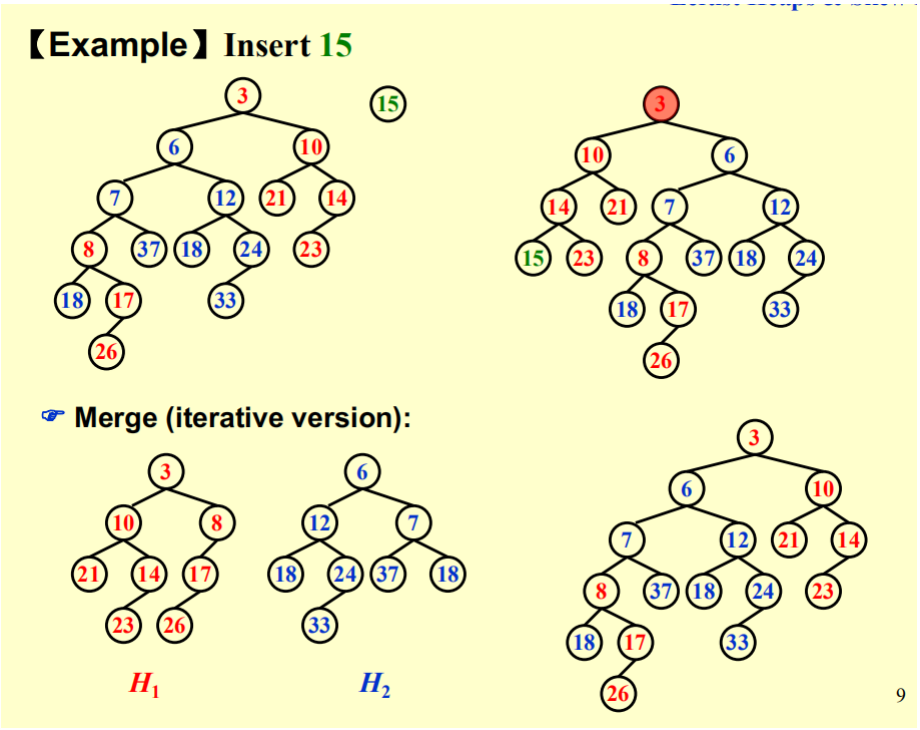


Skew Heap

- Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children.
- It is an open problem to determine precisely the expected right path length of both leftist and skew heaps

Merge

Always swap the left and right children **except that the largest of all the nodes on the two right paths** does not have its children swapped



Amortized Analysis of Merge

> A node p is **heavy** if the number of descendants of p's right subtree is no less than half of the number of descendants of p, and **light** otherwise.
>
> The nodes whose status can be changed between heavy and light are those **initially on the right path**

$$T_{amortized} = O(\log N)$$
$$\Phi(D_i) : \text{concerning the number of heavy nodes in the whole tree}$$

We define:

$$H_1, H_2 : \text{the length of the right path}$$
$$l_1, l_2 : \text{the number of light nodes on the right path, which possibly become heavy nodes after swaps}$$
$$h_1, h_2 : \text{the number of heavy nodes on the right path, which become light nodes after swaps}$$
$$h : \text{heavy nodes not on the right path, whose status never changes}$$

Thus

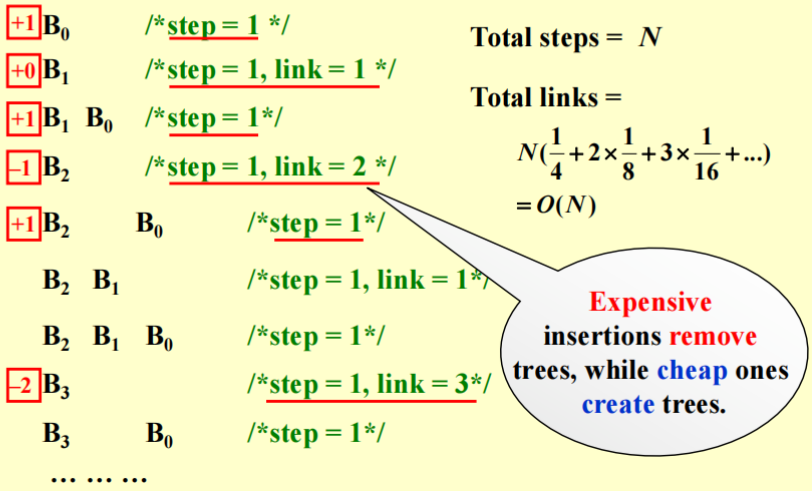
$$H_i = l_i + h_i$$
$$l = O(\log N)$$
$$\text{Before merge:} \Phi_i = h_1 + h_2 + h$$
$$\text{After merge:} \Phi_{i+1} = l_1 + l_2 + h$$
$$T_{worst}(\text{Traverse the whole right path}) \Phi_i = l_1 + h_1 + l_2 + h_2$$
$$T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i \leq 2(l_1 + l_2)$$
$$\Rightarrow \quad T_{amortized} = O(\log N)$$

Binomial Queue

- Better insertion, which takes **constant time** on average

【Claim】A binomial queue of N elements can be built by N successive insertions in O(N) time.

Proof 1 (Aggregate):



Proof 2: An insertion that costs c units results in a net increase of $2 - c$ trees in the forest.

C_i ::= cost of the ith insertion

Φ_i ::= number of trees *after* the ith insertion ($\Phi_0 = 0$)

$C_i + (\Phi_i - \Phi_{i-1}) = 2$ for all $i = 1, 2, \ldots, N$

Add all these equations up $\Rightarrow \sum_{i=1}^{N} C_i + \Phi_N - \Phi_0 = 2N$

$$\sum_{i=1}^{N} C_i = 2N - \Phi_N \le 2N = O(N)$$

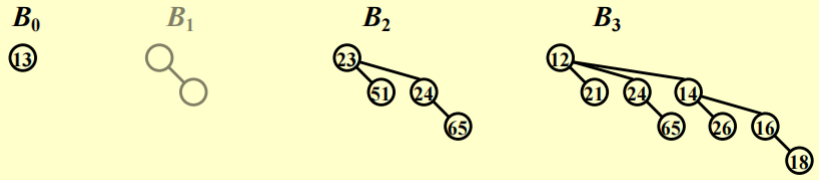
$\blacksquare$

$T_{worst} = O(\log N)$, but $T_{amortized} = 2$

- Heap order
- B_k has k children, $B_0, B_1, \ldots, B_{k-1}$
- B_k has 2^k nodes
- The number of nodes in height d is C_k^d
- Considering the binary representation, the representation of a certain size is unique
  - at most $\lceil \log N \rceil$ binomial trees
  - the representation can be calculated by the binary calculation
- Considering the last feature, it takes $O(\log N)$ to find the minimum key, or we directly maintain a variable storing the minimum key

【Example】Represent a priority queue of size 13 by a collection of binomial trees.

Solution: $13 = 2^0 + 0\times2^1 + 2^2 + 2^3 = 1101_2$



Merge

- Merge B_0 (if existed)
- Merge B_1, where there are multiple choices, simply we can merge the two existed B_1 and leave the newly generated B_1
  - Alternative: merge the two with large roots and leave the smallest
- Continue the process
  - just like binary addition
- Worst case: $O(\log N)$

Insert

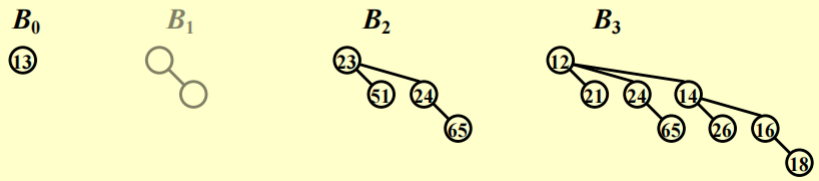
> A special case of merge

DeleteMin

1. Find B_k containing the minimum key
   - $O(\log N)$
2. All the queues except B_k make H'
   - $O(1)$
3. Removing the minimum key (root) from B_k makes H''
   - $O(\log N)$
4. Merge H' and H''
   - $O(\log N)$

【Example】Represent a priority queue of size 13 by a collection of binomial trees.

Solution: $13 = 2^0 + 0\times2^1 + 2^2 + 2^3 = 1101_2$



Implementation

- Binomial queue: a pointer array
- Binomial tree: a linked tree in left-child-next-sibling
  - **the leftmost subtree is the one with the most nodes**, in convenience of insertion
  - > 为了操作简单，二项树的子树采用递减的顺序（二项队列存储二项树仍是按照高度递增排序的）。因为新的子树将会是最大的子树，将它作为第一个儿子比将它作为最后一个儿子（要遍历孩子链表将新的子树放在链表末尾）更加方便。

Combine trees:

```
BinTree CombineTrees( BinTree T1, BinTree T2 )
{ /* merge equal-sized T1 and T2 */
    if ( T1->Element > T2->Element )
        /* attach the larger one to the smaller one */
        return CombineTrees( T2, T1 );
    /* insert T2 to the front of the children list of T1 */
    T2->NextSibling = T1->LeftChild;
    T1->LeftChild = T2;
    return T1;
}
```

Merge:

```
BinQueue Merge( BinQueue H1, BinQueue H2 )
{ BinTree T1, T2, Carry = NULL;
    int i, j;
    if ( H1->CurrentSize + H2-> CurrentSize > Capacity ) ErrorMessage();
        H1->CurrentSize += H2-> CurrentSize;
    for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
        T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
        switch( 4*!!Carry + 2*!!T2 + !!T1 ) {
            case 0: /* 000 */
            case 1: /* 001 */ break;
            case 2: /* 010 */ H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL;
break;
            case 4: /* 100 */ H1->TheTrees[i] = Carry; Carry = NULL; break;
            case 3: /* 011 */ Carry = CombineTrees( T1, T2 );
            H1->TheTrees[i] = H2->TheTrees[i] = NULL; break;
            case 5: /* 101 */ Carry = CombineTrees( T1, Carry );
            H1->TheTrees[i] = NULL; break;
            case 6: /* 110 */ Carry = CombineTrees( T2, Carry );
            H2->TheTrees[i] = NULL; break;
            case 7: /* 111 */ H1->TheTrees[i] = Carry;
            Carry = CombineTrees( T1, T2 );
            H2->TheTrees[i] = NULL; break;
         } /* end switch */
    } /* end for-loop */
    return H1;
}
```

DeleteMin:

```
ElementType DeleteMin( BinQueue H )
{ BinQueue DeletedQueue;
    Position DeletedTree, OldRoot;
    ElementType MinItem = Infinity; /* the minimum item to be returned */
    int i, j, MinTree; /* MinTree is the index of the tree with the minimum item
*/
    if ( IsEmpty( H ) ) { PrintErrorMessage(); return -Infinity; }
    for ( i = 0; i < MaxTrees; i++) { /* Step 1: find the minimum item */
        if( H->TheTrees[i] && H->TheTrees[i]->Element < MinItem ) {
            MinItem = H->TheTrees[i]->Element; MinTree = i;
         } /* end if */
    } /* end for-i-loop */
    DeletedTree = H->TheTrees[ MinTree ];
    H->TheTrees[ MinTree ] = NULL; /* Step 2: remove the MinTree from H => H' */
    OldRoot = DeletedTree; /* Step 3.1: remove the root */
    DeletedTree = DeletedTree->LeftChild; free(OldRoot);
    DeletedQueue = Initialize(); /* Step 3.2: create H" */
    DeletedQueue->CurrentSize = ( 1<<MinTree ) - 1; /* 2MinTree - 1 */
    for ( j = MinTree - 1; j >= 0; j - - ) {
        DeletedQueue->TheTrees[j] = DeletedTree;
        DeletedTree = DeletedTree->NextSibling;
        DeletedQueue->TheTrees[j]->NextSibling = NULL;
    } /* end for-j-loop */
    H->CurrentSize - = DeletedQueue->CurrentSize + 1;
    H = Merge( H, DeletedQueue ); /* Step 4: merge H' and H" */
    return MinItem;
}
```

Fibonacci heap

1. Support a set of operations that constitutes what is known as a 'mergeable heap'
   Mergeable:
   - *MAKE-HEAP*
   - *INSERT*
   - *MINIMUM*
   - *EXTRACT-MIN*
   - *UNION*

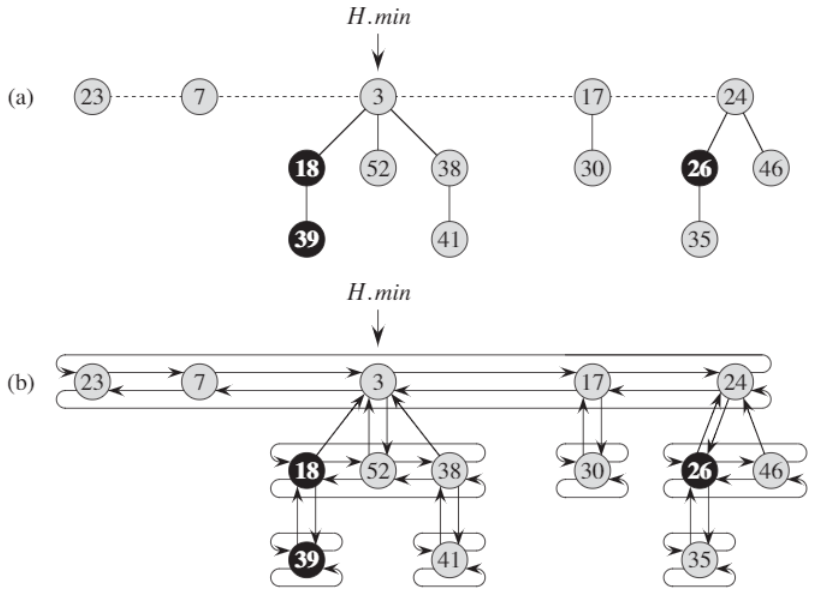
   Specially:
   - *DECREASE-KEY*
   - *DELETE*
2. Several operations run in constant amortized time

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

> especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed
>
> high constant factor because of complex operations
>
> works better for a dense graph (*decrease_key* updating the shortest distance is used frequently)

> Proof: P523, *Introduction to Algorithm*

Definition



1. Circular, doubly linked lists
    - Each node x contains a pointer $x.p$ to its parent and a pointer $x.child$ to **any one** of its children
    - The children of x are linked together in a circular, doubly linked list, which we call the **child list** of x
    - Siblings may appear in a child list in any order
2. Each node has two other attributes
    - store the number of **direct children** in the child list of node x in $x.degree$
    - $x.mark$ indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node.
3. The root of a tree containing the minimum key is called **minimum node** of the Fibonacci heap

4. $H.n$ records the nodes currently in the heap
5. $H.min$ acts like the real root pointing to the root list

- $t(H)$: the number of trees in the root list of H
- $m(H)$: the number of marked nodes in H
- Potential function: $\Phi(H) = t(H) + 2m(H)$
- $D(n)$: maximum degree, exactly $O(\lg n)$

INSERT

```
x.degree = 0
x.p = NIL
x.child = NIL
x.mark = FALSE
if H.min == NIL
    create a root list for H containing just x
else insert x into H's root list
    if x.key < H.min.key
        H.min = x
H.n = H.n + 1
```

- Some initialization
- Simply add one root

MINIMUM

Since $H.min$ records the minimum key, it's easy to implement in $O(1)$ time

UNION

```
H = MAKE_FIB_HEAP()
H.min = H1.min
concatenate the two root lists
if(!H1.min) or (H2.min and H2.min.key < H1.min.key)
    H.min = H2.min
H.n = H1.n + H2.n
return H
```

- merge the two root lists
- determine the new minimum node

EXTRACT-MIN

- Where the delayed work of consolidating trees in the root list finally occurs
- The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged.

- $O(D(n))$

```
z = H.min
if z
    for each child x of z
        add x to the root list of H(pointers changed)
        x.p = NIL
    remove z from the root list of H(pointers unchanged)
    if z == z.right
        H.min = NIL
    else H.min = z.right
        consoidate(H)
    H.n = H.n - 1
return z
```

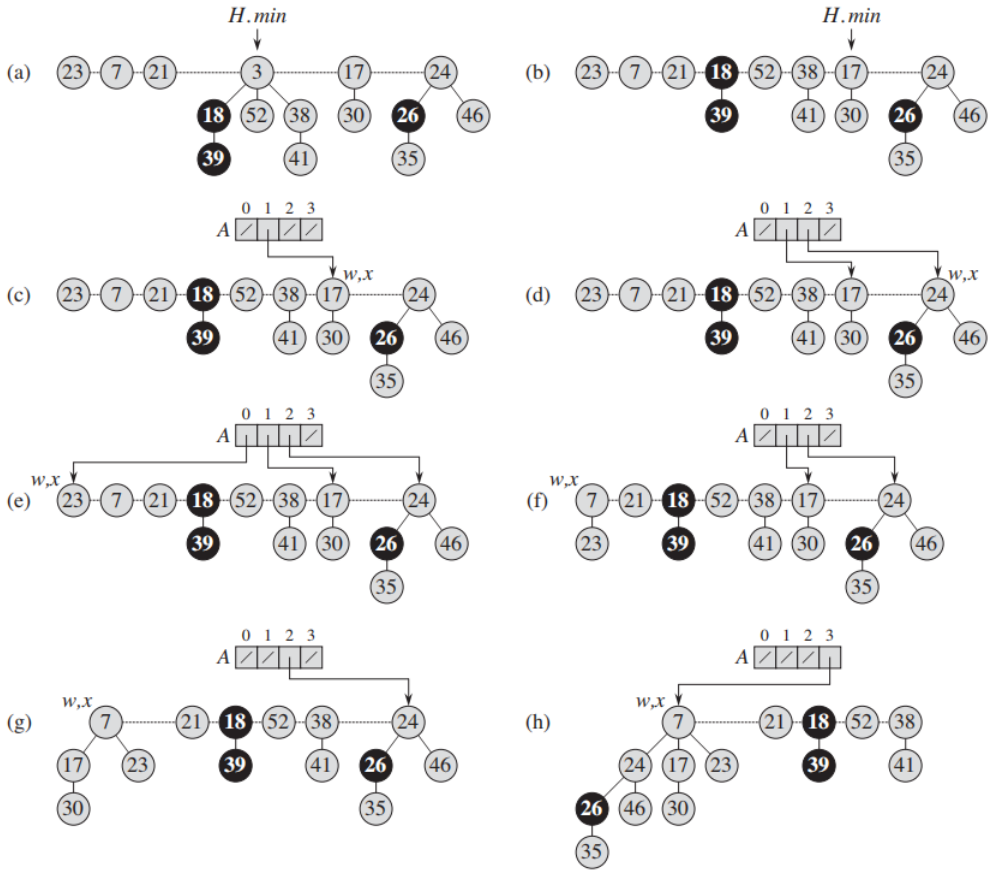
`Consolidate()` consists of repeatedly executing the following steps until every root in the root list has a distinct $degree$ value:

1. Find two roots x and y in the root list with the same degree, where $x.key \leq y.key$, using an auxiliary array of size $D(H.n)$
2. Remove y from the root list, and make y a child of x by calling the LINK procedure, which increments $x.degree$ and clears $y.mark$

```
ElementType A[D(H.n)]
for i=0 to D(H.n)
    A[i] = NIL
for each node w in the root list of H
    x = w
    d = x.degree
    while A[d]
        y = A[d]
        if x.key > y.key
            swap(x, y)
        LINK(H, y, x) // x.degree++ here
        A[d]=NIL
        d = d + 1
    A[d] = x
H.min = NIL
for i=0 to D(H.n)
    if A[i]
        if !H.min
            create a root list for H containing just A[i]
            H.min = A[i]
        else insert A[i] into H's root list
            if A[i].key < H.min.key
                H.min = A[i]
```

LINK:

```
remove y from the root list of H
make y a child of x
x.degree++
y.mark = FALSE
```



DECREASE-KEY

```
if k > x.key
    error
x.key = k
y = x.p
if y and x.key < y.key
    cut(H, x, y)
    cascade_cut(H, y)
if x.key < H.min.key
    H.min = x
```

cut:

```
remove x from the child list of y
y.degree--
x.p = NIL
x.mark = FALSE
```

cascade_cut:

```
z = y.p
if z
    if !y.mark
        y.mark = TRUE
    else cut(H, y, z)
        cascade(H, z)
```

$mark$ is used to obtain the desired time bound, keeping the general height not too large:

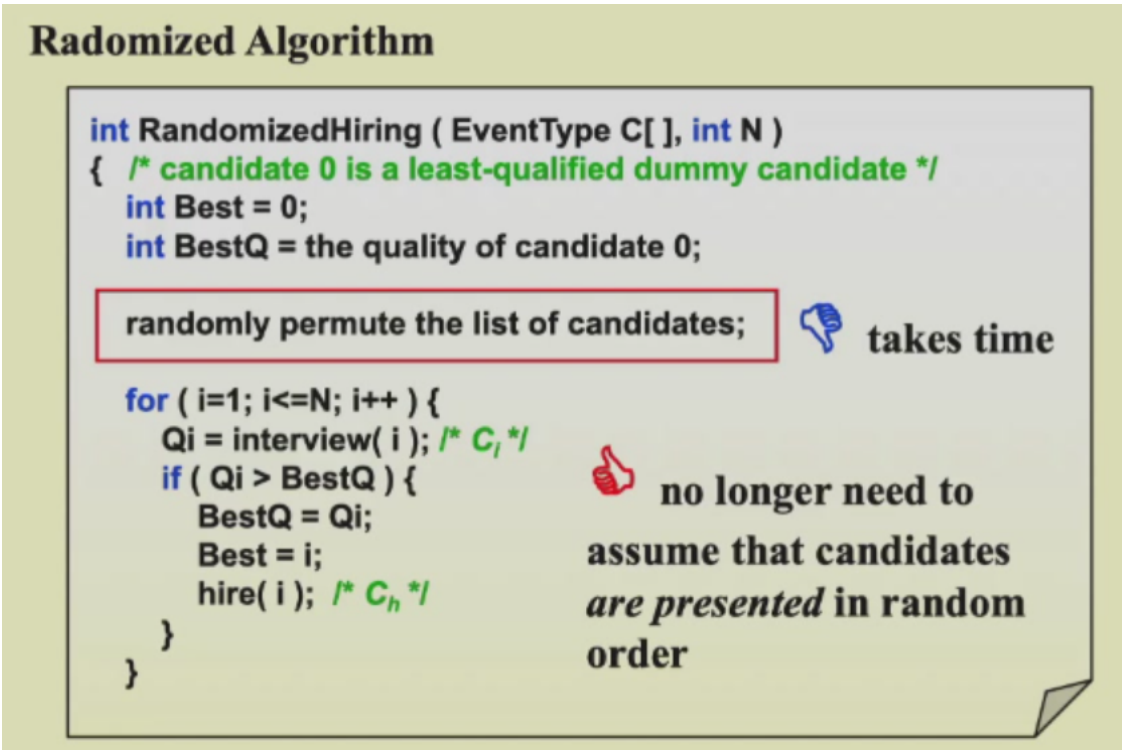
1. at some time, x was a root (`FALSE`)
2. then x was linked to(made the child of) another node(`FALSE`)
3. then two children of x were removed by cuts(`TRUE` right after the first was removed)

Randomized

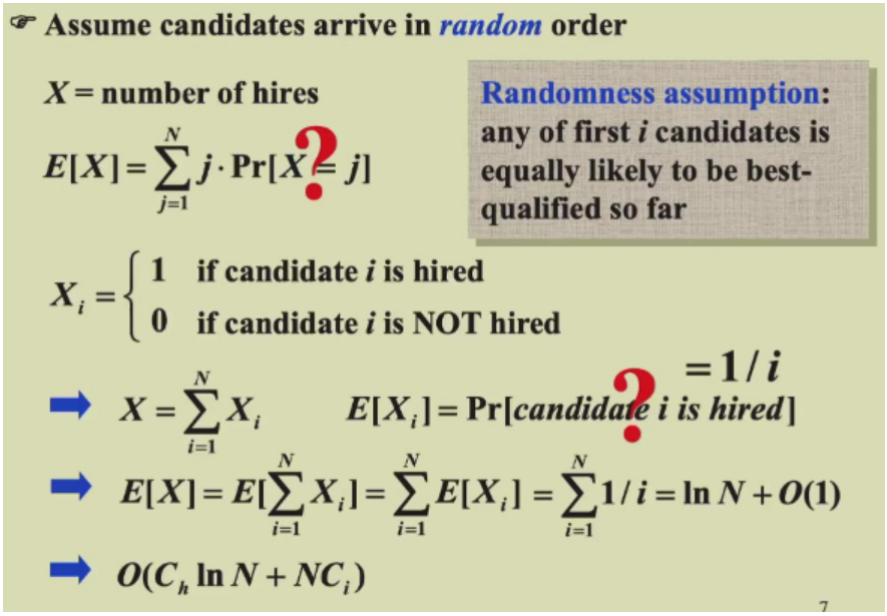
- always correct and run efficiently in expectation

Hiring

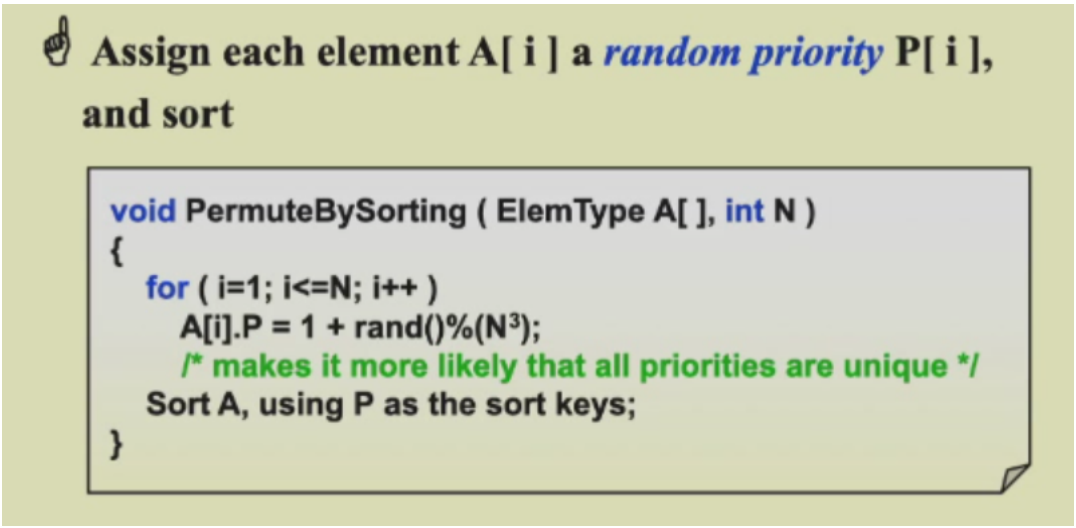
Offline



- Hire an employee from interviewees recommended by students
- Interviewing cost C_i for each student << Hiring cost C_h
- M people are hired and the cost: $O(NC_i + MC_h)$
- cost analysis:

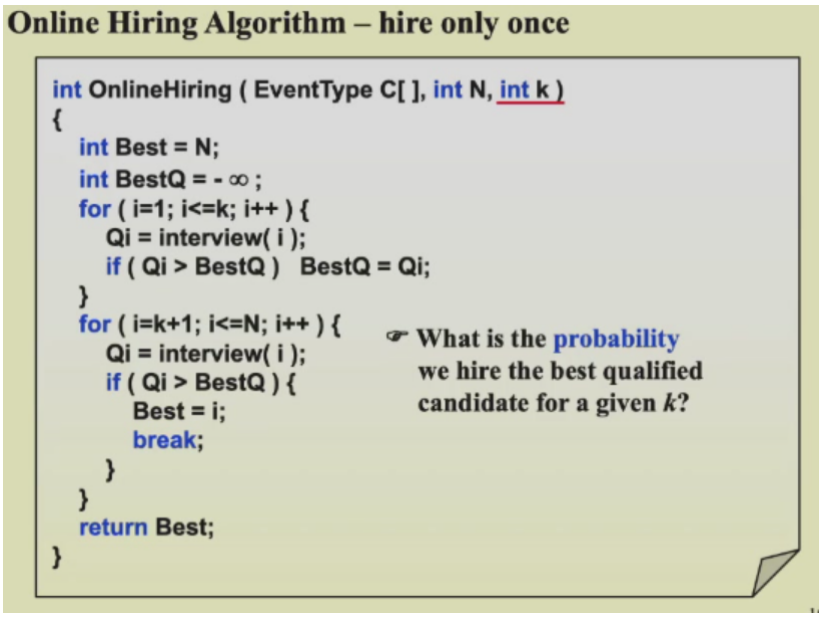


- permute the order at first:

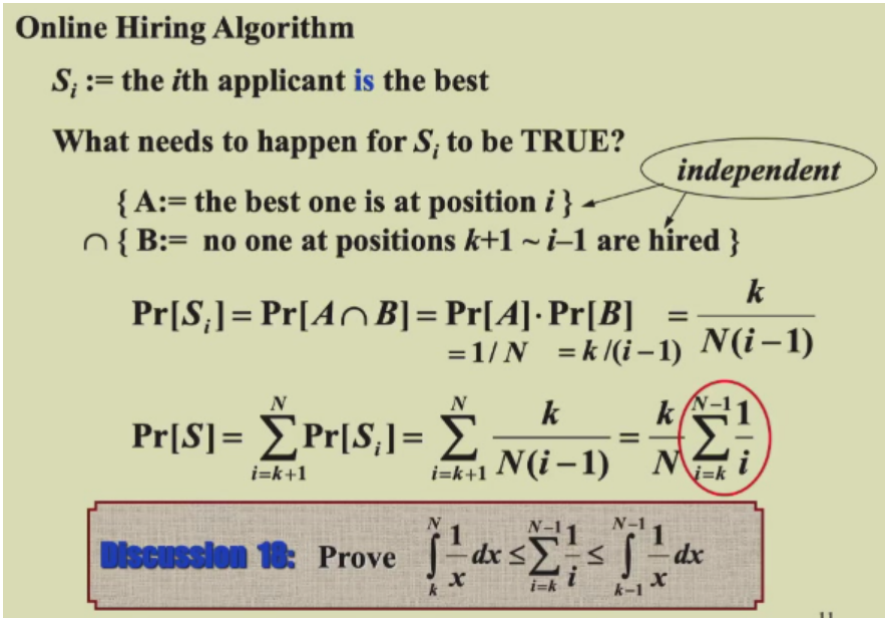


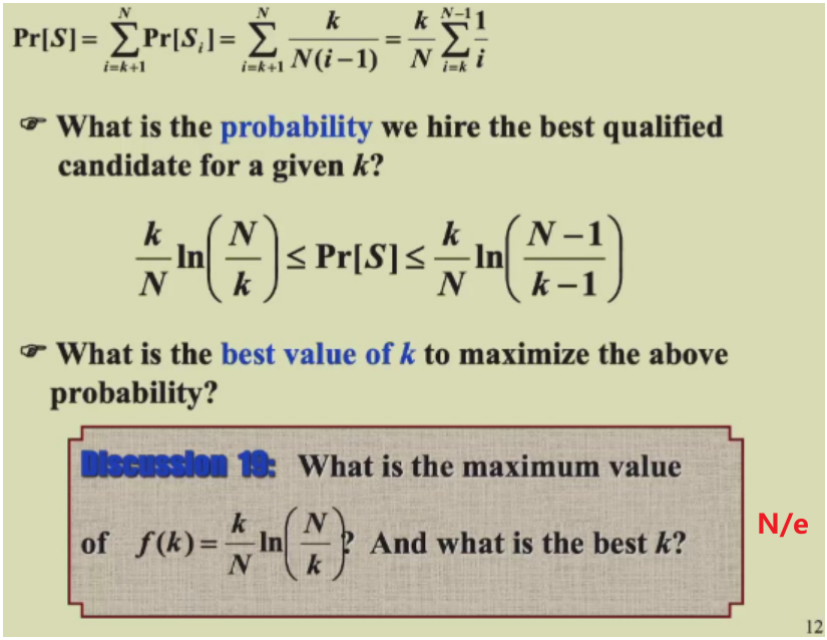
Online

The former k interviewees will not be hired, which is for finding a standard:



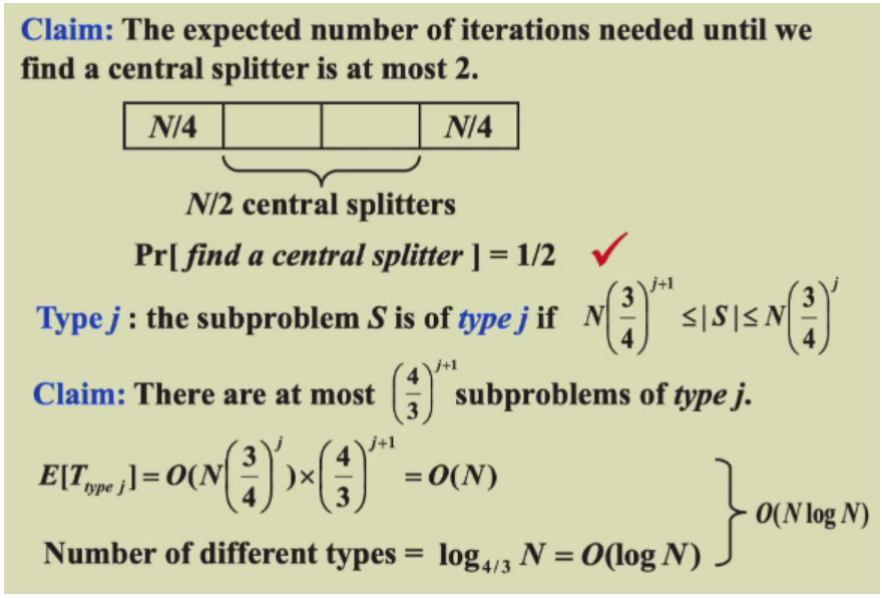
Analysis:





Quicksort

- Central splitter: the pivot that divides the set so that each side contains at least $n/4$
- Modified Quicksort: always select a central splitter before recursions



Random Number Generator

A general function:

$$x_{i+1} = Ax_i \mod M$$

- For a 31-bit prime: $M = 2^{31} - 1, A = 48,271$ is a good combination
- **Adding a constant doesn't necessarily make the function more random**

In most standard library:

$$x_{i+1} = (Ax_i + C) \mod 2^B$$

$A = 48,271,\ B =$ the number of bits in the machine's integer, C is odd

A better one **with no overflow**:

```
static unsigned long seed = 1;

#define A 48271L
#define M 2147483647L
#define Q (M/A) // Quetient
#define R (M%A) // Remainder

double random()
{
    long Tmpseed = A * ( seed % Q ) - R * ( seed / Q );
    if(Tmpseed >= 0)
        seed = Tmpseed;
    else
        seed = Tmpseed + M;
    return (double)seed/M;
}

void initialize(unsigned long InitVal)
{
    seed = Initval;
}
```

> Proof in Page 395 in *Data Structure and Algorithm Analysis in C*

Primality Test

Basic theorem, which is given by Fermat:

$$P \text{ is prime, } 0 < A < P \ \rightarrow \ A^{P-1} \equiv 1 (\text{mod } P)$$

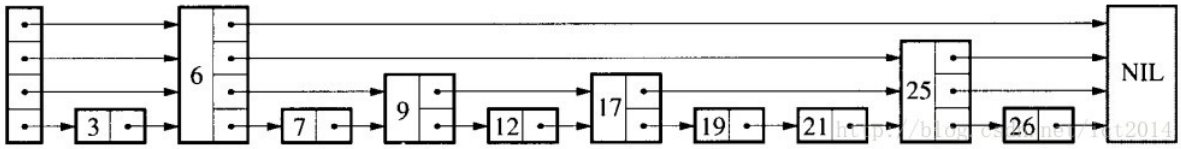
Another theorem, concerning trivial square root:

$$P \text{ is prime, } 0 < X < P, \text{ solutions to } X^2 \equiv 1 (\text{mod } P) \text{ are } X = 1, P-1$$

- If the algorithm declares the input P is not a prime, then P is **certainly not a prime**
- If the algorithm declares the input P is a prime, then P is **very probably to be a prime**
- What the algorithm do:
  - test for *Carmichael numbers*
  - randomly pick $0 < A < P$
  - confirm that $A^{P-1} \equiv 1 (\text{mod } P)$
  - check for non-trivial square root
- For a sufficiently large P, at most $\frac{P-9}{4}$ numbers fool this algorithm. That is, if the algorithm is run for 50 times, the error probability is never more than 2^{-100}

```
HugeInt Witness(HugeInt A, HugeInt i, HugeInt N)
{
    HugeInt x, y;
    if(!i)  // test for ending
        return 1;

    // compute (A^i) mod N
    x = Witness(A, i/2, N);
    if(!x)  // composite
        return 0;

    // find non-trivial square root of 1
    y = (x * x) % N;
    if(y == 1 && x != 1 && x != N-1)
        return 0;

    // if i is odd. this step conforms to computing (A^i) mod N
    if(i % 2)
        y = (A * y) % N;
    return y;
}

int isPrime(HugeInt N)
{
    return Witness(RandInt(2, N-2), N-1, N) == 1;
}
```

Skip list



- use a random number generator
- search and insert in $O(\log N)$
- probabilistic balancing rather than strictly enforced balancing
- less restrictive: the ith pointer in any level k node ($k \geq i$) points to the next node with at least i levels

Find

simply traverse

Insert

- find the position at first and then insert
- the level of the newly inserted node is **randomly generated according to the probability distribution**
  - level 1: 0.5
  - level 2: 0.25
  - ...
  - the probability distribution is decided by the restrictive one
- **the efficiency is almost the same as the restrictive one**

Delete

Find the node, link the adjacent two and free the node

1-2-3 Deterministic Skip List

> 2 elements are *linked* if there exists at least 1 pointer going from one to another
>
> The *gap size* between 2 elements linked at height *b* is equal to the number of elements of height *b-1* between them
>
> The height of a node here is no longer randomly decided but deterministic
>
> Every gap (except possibly the zero gap between the header and the tail) is of size 1, 2, or 3

Worst case: $O(\log N)$

Experimentally $1.57N$ nodes are needed

Space is somehow sacrificed

Actual implementation (everything is connected by the linked list, easy for raising and dropping):

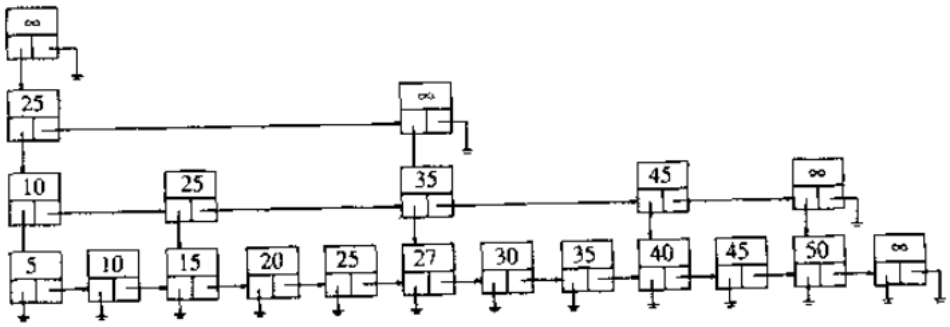


图 12-22　图 12-21 中 1-2-3 确定性跳跃表的链表实现

> A node of level h appears for h times

1 implementation

```
struct SkipNode
{
    ElementType Element;
    SkipList    Right;
    SkipList    Down;
};

static Position Bottom  = NULL; /* Needs initialization */
static Position Tail    = NULL; /* Needs initialization */

/* Initialization procedure */

SkipList
Initialize( void )
{
    SkipList L;

    if( Bottom == NULL )
    {
        Bottom = malloc( sizeof( struct SkipNode ) );
        if( Bottom == NULL )
            FatalError( "Out of space!!!" );
        Bottom->Right = Bottom->Down = Bottom;

        Tail = malloc( sizeof( struct SkipNode ) );
        if( Tail == NULL )
            FatalError( "Out of space!!!" );
        Tail->Element = Infinity;
        Tail->Right = Tail;
    }

    /* Create the header node */
    L = malloc( sizeof( struct SkipNode ) );
    if( L == NULL )
        FatalError( "Out of space!!!" );
    L->Element = Infinity;
    L->Right = Tail;
    L->Down = Bottom;

    return L;
}
```

Find

simply traverse

```
/* Return position of node containing Item, */
/* or Bottom if not found */

Position
Find( ElementType Item, SkipList L )
{
    Position Current = L;

    Bottom->Element = Item;
    while( Item != Current->Element )
        if( Item < Current->Element )
            Current = Current->Down;
        else
            Current = Current >Right;

    return Current;
}
```

- `if` drops a level, which means the node to find is in the gap

Insert

- find the position
- check the gap
- if the gap that will be spliced into is of size 3, raise the level of the middle item
- insert the new node of level 1

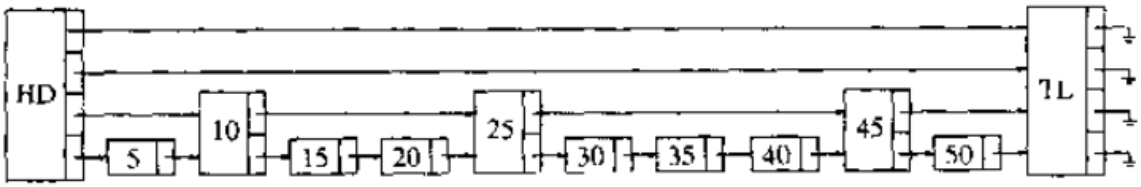


图 12-18　一个 1-2-3 确定性跳跃表

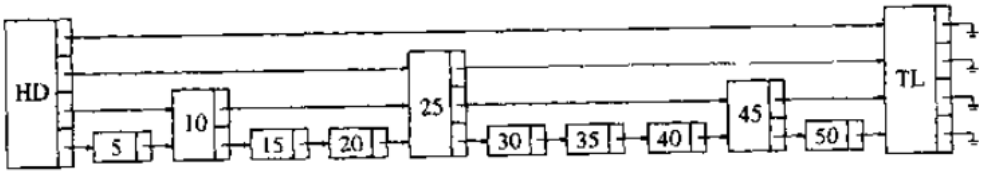


图 12-19　插入 27:首先,通过提升 25 将含 3 个高度 2 的节点的间隙分裂

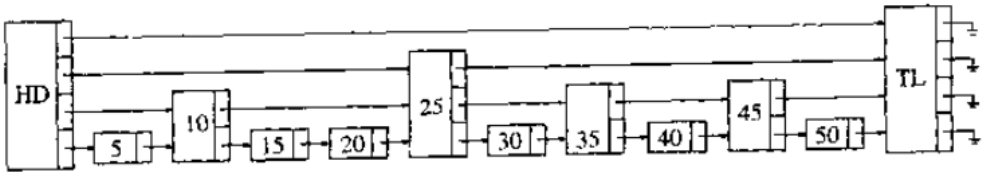


图 12-20　插入 27:其次,通过提升 35 将含 3 个高度 1 的节点的间隙分裂

```
SkipList
Insert( ElementType Item, SkipList L )
{
    Position Current = L;
    Position NewNode;

    Bottom->Element = Item;
    while( Current != Bottom )
    {
        while( Item > Current->Element )
            Current = Current->Right;

        /* If gap size is 3 or at bottom level */
        /* and must insert, then promote the middle element */
        if( Current->Element >
                Current->Down->Right->Right->Element )
        {
            NewNode = malloc( sizeof( struct SkipNode ) );
            if( NewNode == NULL )
                FatalError( "Out of space!!!" );
            NewNode->Right = Current->Right;
            NewNode->Down = Current->Down->Right->Right;
            Current->Right = NewNode;
            NewNode->Element = Current->Element;
            Current->Element = Current->Down->Right->Element;
        }
        else
            Current = Current->Down;
    }
        /* Raise height of DSL if necessary */
        if( L->Right != Tail )
        {
            NewNode = malloc( sizeof( struct SkipNode ) );
            if( NewNode == NULL )
                FatalError( "Out of space!!!" );
            NewNode->Down = L;
            NewNode->Right = Tail;
            NewNode->Element = Infinity;
            L = NewNode;
        }

        return L;
    }
```

size 2: `Current->Element == Current->Down->Right->Right->Element`

size 3: `Current->Element == Current->Down->Right->Right->Right->Element`, which is also `Current->Element > Current->Down->Right->Right->Element`

`Current->Element == Current->Down->Right->Element` in creating a new node: raise the middle node among the 3 nodes

Delete

Inverse insertion

Backtracking

- Undo is allowed
- Try and undo until a desired solution is found
- No need to traverse all the possible solutions: some undesired ones are abandoned at the beginning
- All tuples that satisfy the explicit constraints define a possible *solution space*
- Different solution spaces may have different sizes

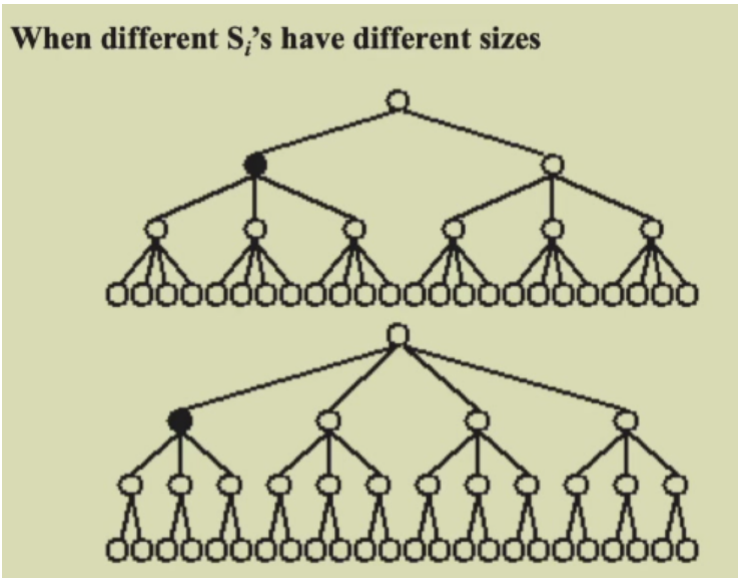
- Partial solution is part of a solution

> If there is no conflict, go ahead
>
> If there is a conflict, try to avoid the conflict
>
> If there is no way, undo
>
> If the search is done, there is a solution

A template:

```
bool Backtracking(int i)
{
    Found = false;
    if(i>N)
        return true;
    for (each xi in S)
    {
        // check constraints
        OK = Check((x1, x2, x3, ..., xi), R);
       if(OK)
       {
           Count xi in;
           Found = Backtracking(i+1);
           if(!Found)
               Undo(i);
       }
       if(Found)
           break;
    }
    return Found;
}
```

Two modes:

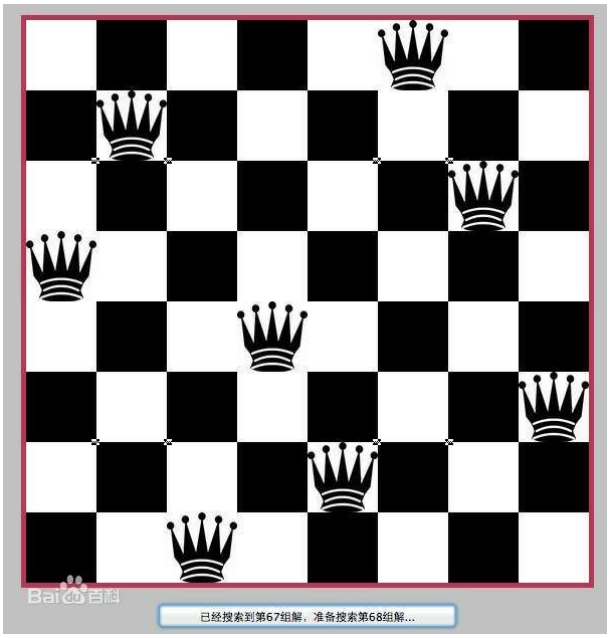


- The subtrees of the first one have higher density, which means one pruning can eliminate more undesired solutions

- The subtrees of the second one have lower density, which is **suitable to probe the constraints when the constraints are not definite**

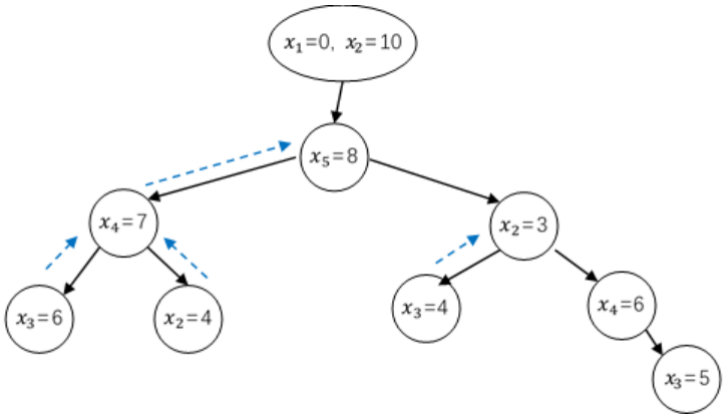
8 Queens

- place 8 queens on a 8×8 chessboard
- constraints (8 queens should not be able to attack each other):
  - $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$
  - $x_i \neq x_j, i \neq j$
  - $y_i \neq y_j, i \neq j$
  - $\frac{x_i - x_j}{y_i - y_j} \neq 1$
- steps:
  - build a solution tree (virtually)
  - perform a *DFS* on the tree and examine the constraints, i.e., try to place the next queen basing on the fore-results



Turnpike Reconstruction

- Given a distance set of $\frac{N(N-1)}{2}$ elements D
- Try to place N turnpikes on the highway satisfying the distance constraint given by set D
- **Better to set** $x_1 = 0$
- Steps:
  - calculate N and set $x_1 = 0$
  - try to set the farthest turnpike or the nearest turnpike (eliminate the largest distance) at first, which efficiently avoids repetition of shorter distances
  - eliminate exist distance from the set D

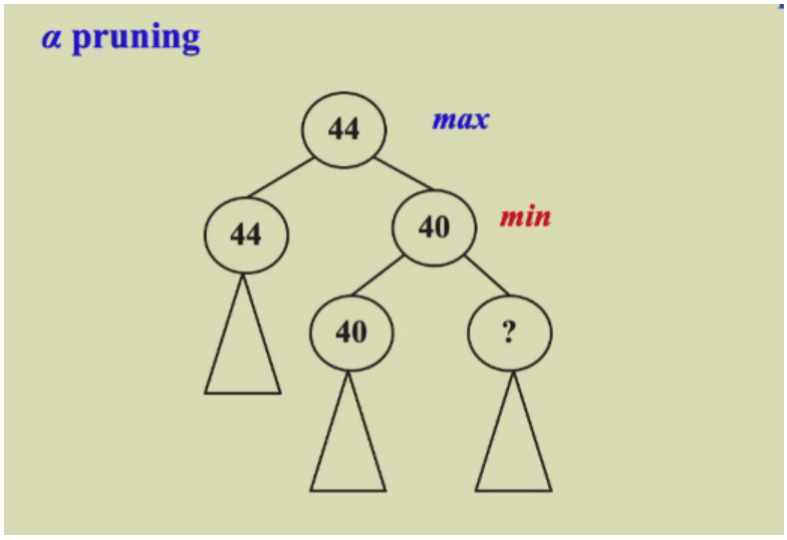


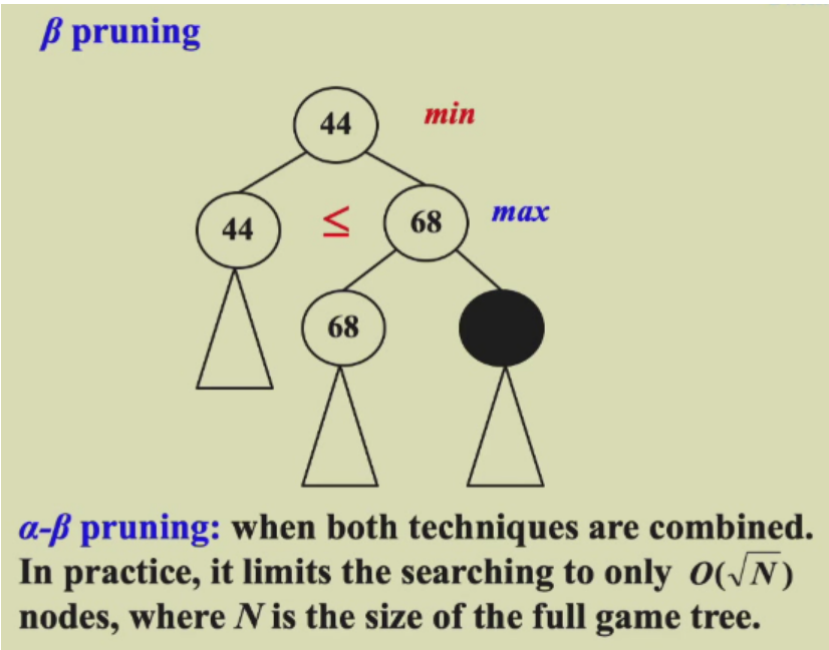
Tic-tac-toe

Minimax strategy

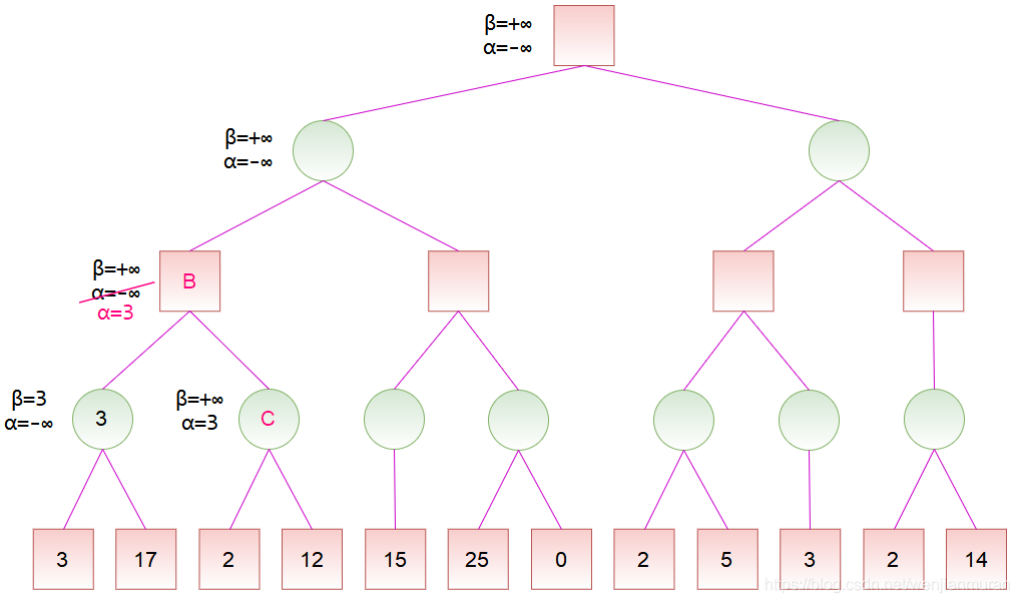
- **Pessimistic**
- Considering the characteristic of chess games (alternative hand), we can always adopt the best (maximum profits) move after the rival takes one move. Thus, in a long-term view, we can take the step with minimum maximum profits, which means **the profit of the next move is the lowest of the next-next moves** while **the profits of next-next moves are the highest of next-next-next moves**
- quantify the advantage: $f(P) = W_{computer} - W_{human}$, in which W represents the positions of potential wins
- choose the next placing according to the function, listing and comparing the predictive results
- *human* is trying to **minimize** the value
- *computer* is trying to **maximize** the value

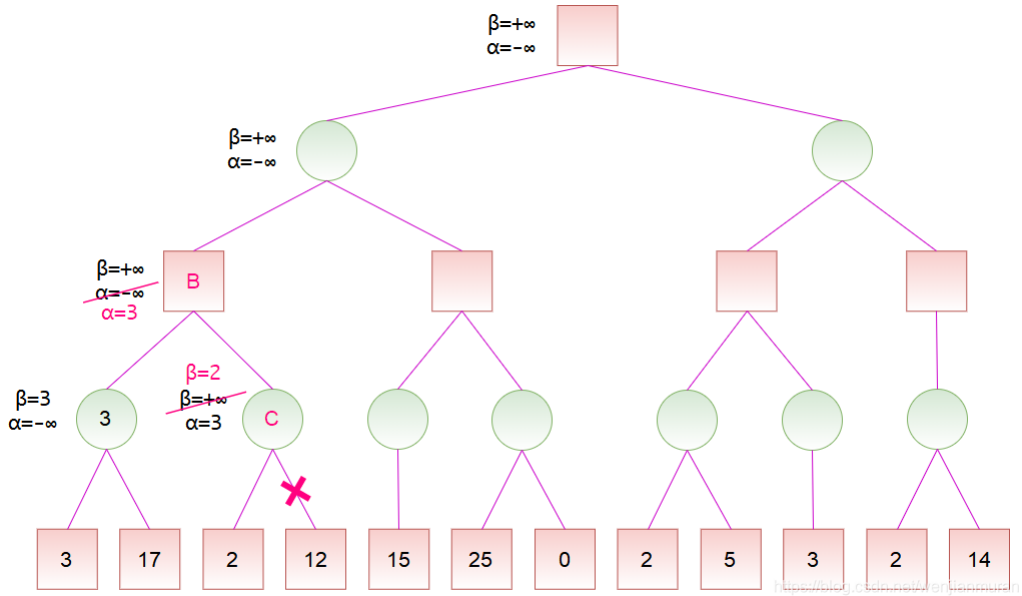
$\alpha - \beta$ Pruning





- The program builds the tree in a certain order, which makes pruning possible
- α and β in fact represent different strategies of the 2 rivals
- Combining the 2 strategies to predict the future situations or future move
- Step
  - Decide the number of the future moves to predict
  - List the possible profits and build the tree
  - From left to right calculate the profits bottom-up
  - If the current maximum profit (α) is **larger than** minimum profit (β, which can only be smaller after consequent traverse) and the subtree has not been completely traversed, then the untraversed branches can be pruned





[1 zhihu]

[2 csdn]

Bron-Kerbosch

To find the number of vertices in the **maximal clique**

[1 bokeyuan]

[2 wiki]

Divide and Conquer

- **Divide** the problem into a sub-problems of size N/b
- **Conquer** the sub-problems by solving them recursively in $T(N/b)$ time
- **Combine** the solutions in $f(N)$ time

Total time expenditure:

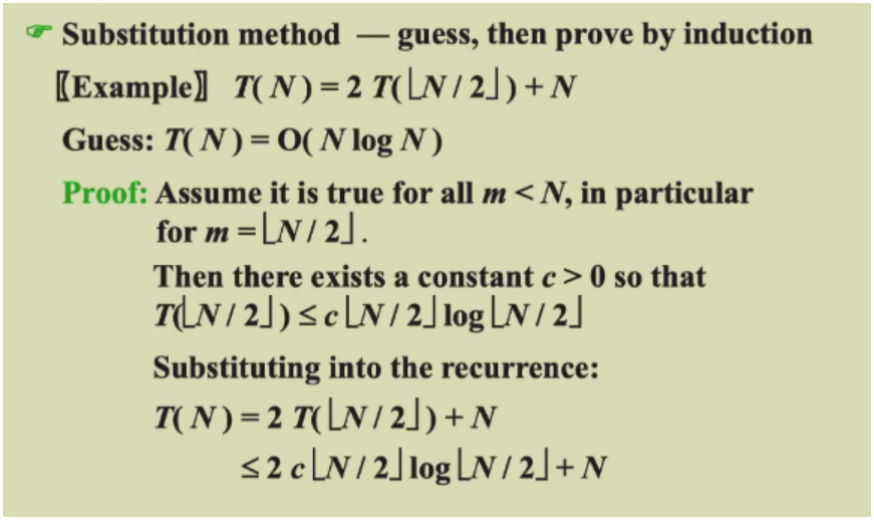
$$T(N) = aT(N/b) + f(N)$$

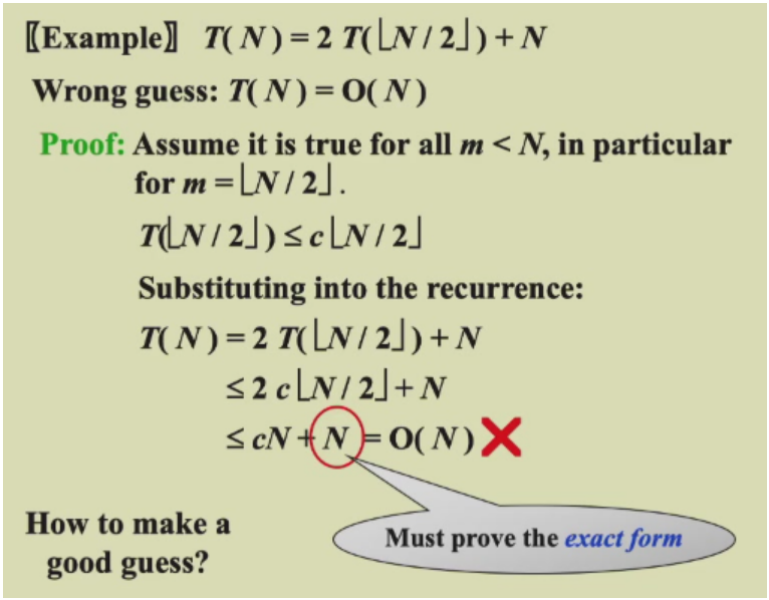
> When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions
>
> always assume $T(N) = \Theta(1)$ for small n
>
> that $T(n)$ is constant for small n. For example, we normally state recurrence (4.1) as
>
> $$T(n) = 2T(n/2) + \Theta(n)\,, \tag{4.4}$$
>
> without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

- $T(N) = aT(N/b) + f(N) \rightarrow \Theta()$
- $T(N) < aT(N/b) + f(N) \rightarrow O()$

- $T(N) > aT(N/b) + f(N) \rightarrow \Omega()$

Substitution method

- guess
- prove by induction
- 

- when proving the upper bound, we cannot prove a looser upper bound


while subtracting a lower-order term works:

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$.

We guess that the solution is $T(n) = O(n)$, and we try to show that $T(n) \le cn$ for an appropriate choice of the constant c. Substituting our guess in the recurrence, we obtain

$$\begin{aligned} T(n) &\le c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 , \end{aligned}$$

which does not imply $T(n) \le cn$ for any choice of c. We might be tempted to try a larger guess, say $T(n) = O(n^2)$. Although we can make this larger guess work, our original guess of $T(n) = O(n)$ is correct. In order to show that it is correct, however, we must make a stronger inductive hypothesis.

Intuitively, our guess is nearly right: we are off only by the constant 1, a lower-order term. Nevertheless, mathematical induction does not work unless we prove the exact form of the inductive hypothesis. We overcome our difficulty by *subtracting* a lower-order term from our previous guess. Our new guess is $T(n) \le cn - d$, where $d \ge 0$ is a constant. We now have

$$\begin{aligned} T(n) &\le (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\le cn - d , \end{aligned}$$

- the constant can be chosen to satisfy certain conditions
- the start of n, which means n_0 of $n > n_0$, can also be chosen
- changing variables helps:

  Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. As an example, consider the recurrence

  $T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \lg n$,

  which looks difficult. We can simplify this recurrence, though, with a change of variables. For convenience, we shall not worry about rounding off values, such as $\sqrt{n}$, to be integers. Renaming $m = \lg n$ yields

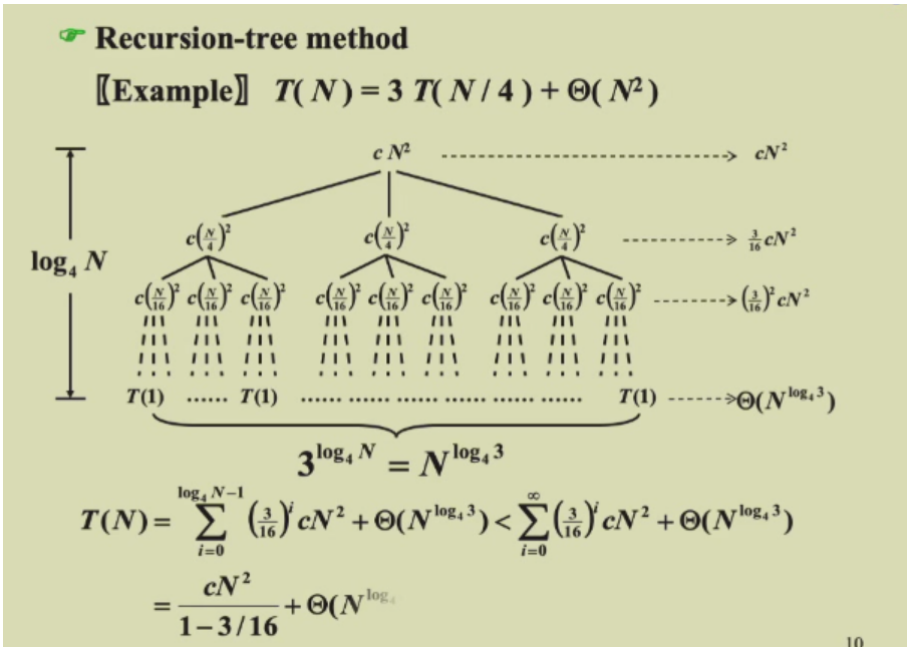
  $T(2^m) = 2T(2^{m/2}) + m$.

  We can now rename $S(m) = T(2^m)$ to produce the new recurrence

  $S(m) = 2S(m/2) + m$,

Recursion-tree method

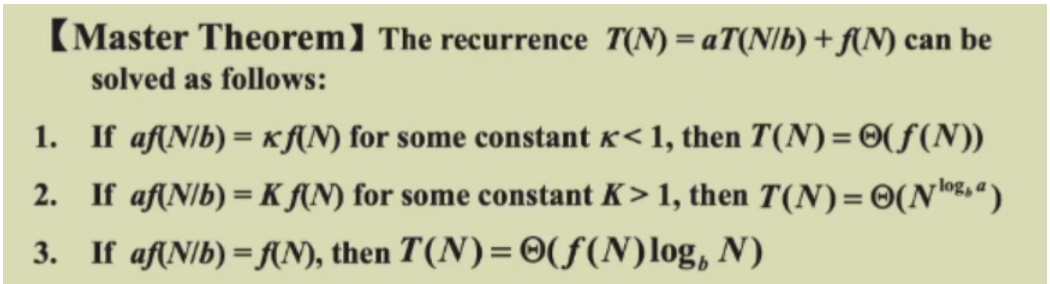
- list the tree
- sum up the cost of dividing and conquering

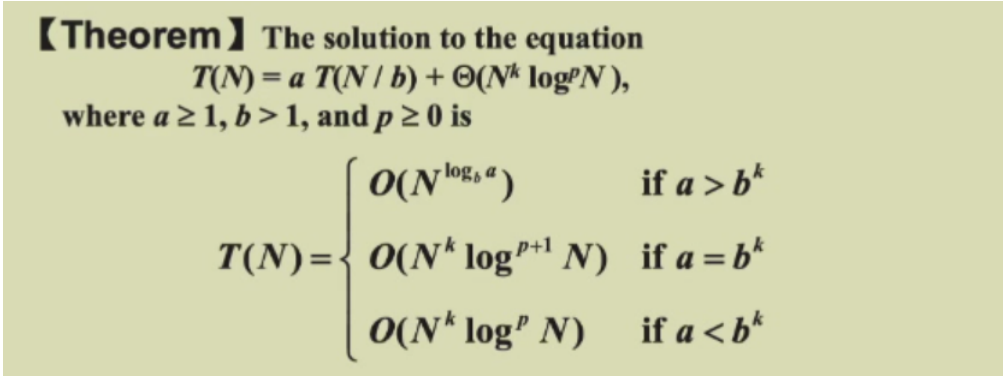
- 

- a strategy to find clues for guessing, which may not provide strict proof
- it doesn't matter that the tree is not balanced, since we are just getting a guess
- $a^{\log_b N} = a^{\log_a N / \log_a b} = N^{1/\log_a b} = N^{\log_b a}$
- In short: $T(N) = \Theta(N^{\log_b a}) + \sum_{j=0}^{\log_b N - 1} a^j f(N/b^j)$, which is the cost of conquering and the cost of dividing & recombining.

Master method

- $O(N^{\log_b a})$ is exactly the cost on the leaf nodes in *Recursion-tree*
- It does not cover every condition
- Division is used for proving polynomial greatness
- **The leaf nodes dominate**: if $f(N) = O(N^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b a})$
- **Similar domination**: if $f(N) = \Theta(N^{\log_b a})$, then $T(N) = \Theta(N^{\log_b a} \log N)$
- **The root dominates**: if $f(N) = \Omega(N^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(N/b) < cf(N)$ for some $c < 1$ and all sufficiently large N, then $T(N) = \Theta(f(N))$
- In short: $T(N) = \Theta(N^{\log_b a}) + \sum_{j=0}^{\log_b N - 1} a^j f(N/b^j)$, which is the cost of conquering and the cost of dividing & recombining. Take $g(N) = \sum_{j=0}^{\log_b N - 1} a^j f(N/b^j)$, and $g(N)$ exactly satisfies the 3 cases of master method, in which case 3 is guaranteed by the regularity condition
- The other forms cover some other different conditions:





Maximum Sum of Subsequences

- Transform a problem of finding the date pair (buy and sell) with the highest profit to finding the subsequence with the maximum net change
- Some elements are negative
- Find the one **on the right**, the one **on the left** and the one **across the midpoint**
- $T(N) = T(N/2) + \Theta(N)$

Strassen's Method

- $T(N) = 7T(N/2) + \Theta(N^2)$
- Matrix addition is a cheaper operation than multiplication, and 18 extra additions is better than 1 multiplication
- A tactful manner to deal with partitioning, which takes $\Theta(1)$ time:

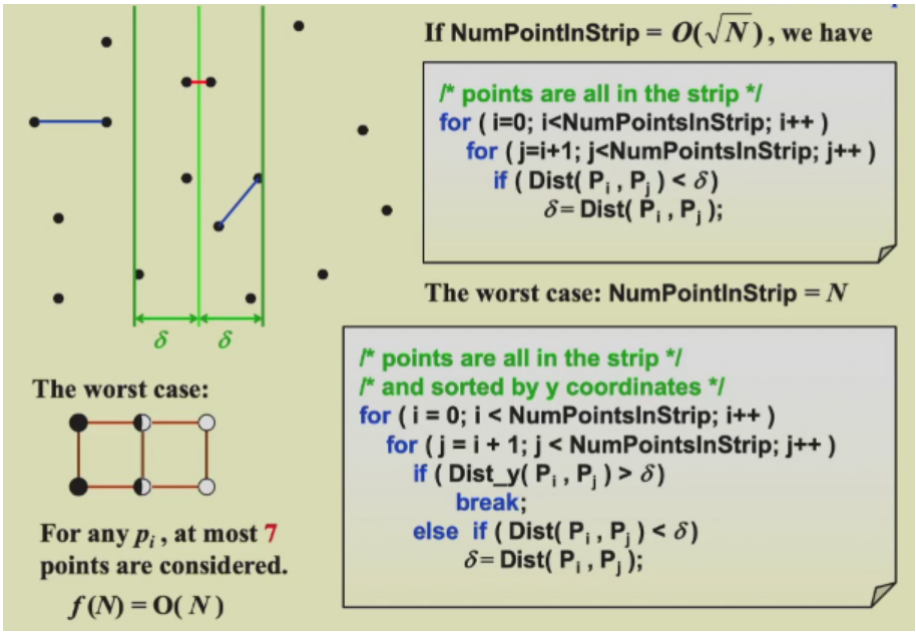
```
int offset = n / 2;
mat[i][j - offset]...
mat[i - offset][j]...
```

1 zhihu

Closest Points

- similar to maximum subsequence sum

1. Divide the points by their x-coordinates
2. Conquer the subsets recursively
3. For combining, we do not need to list all the combinations:
    - sort the points in the δ-strip according to y-coordinates (a kind of pruning)
    - fix a point to consider, then in a $\delta \times \delta$ grid, at most 7 points need to be considered



Dynamic Programming

- the compiler may not do justice to recursive algorithm
- rewrite the recursive algorithm as a non-recursive algorithm
- since we do not use recursion, we need to define a **state** to indicate the subproblem we are dealing with
    - The **state** we use should be sufficient to **uniquely define a subproblem**, e.g. we use a 2-dimensional array here but not a array in the multiplication problem
    - Add information to **state** if it is insufficient

key: State Transition

> 循环的顺序问题:
>
> 0-1背包中如果采用Matrix，则顺序逆序都可以，因为每一次循环的结果不会互相覆盖，已更新的结果不会通过状态转移方程影响未来的结果 (内循环的起始条件避免了一个物体会被取两次)
>
> 而如果采用Array存放结果，则是**滚动数组**，必须逆序，否则某轮循环的结果会覆盖之前循环的结果，导致已更新的结果通过状态转移方程影响未来的结果，违背了0-1背包中每个东西只能放一次的原则 （此时放弃了记录哪些东西构成了最优解）
>
> 压缩表的维数有时需要结合调整循环的顺序与逆序来进行
>
> sample:
>
> 假设一个物品A价值100，体积为2，正在进行第i次外层循环,假设第i-1次的F[v]全为0，按照[0 ... V]的顺序遍历，那么v = 2时，F[2] = max(F[2], F[0] + 1000) = 1000;当j = 4时,F[4] = max(F[4], F[2] + 1000) = 2000，此时max用到的F[2]是已经更新过的 （该物体被取了两次），所以正序遍历会造成错误的结果
>
> 与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的
>
> 参考题目1

> 参考题目2

method:

- Characterize an optimal solution
- Recursively define the optimal values
- Compute the values in some order
- Reconstruct the solving strategy

elements of DP:

- Optimal substructure
- Overlapping sub-problems

A. Longest-Common-Subsequence Problem

Given two sequences $X=\langle x_1, x_2, \ldots, x_n\rangle$ and $Y=\langle y_1, y_2, \ldots, y_m\rangle$, find the longest common subsequence of X and Y.

X = <A, B, C, B, D, A, B>
Y = <B, D, C, A, B, A> → <B, C, B, A>

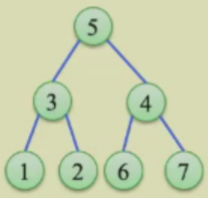
Discussion 11:
How to solve LCS problem by DP?

Comparing m-th element of X and n-th element of Y:

$$LCS[i][m] = \begin{cases} LCS[i-1][m-1] + X[m] & X[m] = Y[n] \\ LCS[i-1][m-1] & X[m] \neq Y[n] \end{cases}$$

B. Anniversary Party

There is going to be a party to celebrate the 80-th Anniversary of the Ural State University. The University has a hierarchical structure of employees. It means that the supervisor relation forms a tree rooted at the rector V. E. Tretyakov. In order to make the party funny for every one, the rector does not want both an employee and his or her immediate supervisor to be present. The personnel office has evaluated conviviality of each employee, so everyone has some number (rating) attached to him or her. Your task is to make a list of guests with the maximal possible sum of guests' conviviality ratings.



$MAXSUM[i][0]$ represents the sum if the i-th one isn't coming

$MAXSUM[i][1]$ represents the sum if the i-th one is coming

$$MAXSUM[i][0] = \sum MAXSUM[subordinate][1]$$
$$MAXSUM[i][1] = \sum MAXSUM[subordinate][0]$$
$$SUM[i] = \max\{MAXSUM[i][0], MAXSUM[i][1]\}$$

Use a Table (Memoization)

Some calls are redundant:

```
int Fib(int N){
    if(N<=1)
        return 1;
    else
        return Fib(N-1)+Fib(N-2);
}
```

The growth redundant calculation is explosive

State: N

In fact, only the last two results are necessary:

```
int Fib(int N){
    int last, nexttolast, answer;
    if(N<=1)
        return 1;

    last = nexttolast = 1;
    for(int i=2; i<=N; i++){
        answer = last + nexttolast;
        nexttolast = last;
        last = answer;
    }
    return answer;
}
```

Ordering Matrix Multiplication

- different orderings demand significantly different operations
- the number of possible orderings grows exponentially: $T(n) = \sum_{i-1}^{n-1} T(i)T(n-i), O(\frac{4^n}{n\sqrt{n}})$
- none of the **obvious greedy strategies** seems to work

solution:

- if A_i has c_i columns, then A_i has c_{i-1} rows
  c_0 is the number of rows in A_1
- State: define $m_{i,j}$ as the minimum multiplications of $A_i A_{i+1} \dots A_j$

$$m_{i,j} = \begin{cases} 0 & i = j \\ \min\limits_{i \le l \le j}\{m_{i,l} + m_{l+1,j} + c_{i-1}c_l c_j\} & i \neq j \end{cases}$$

- take $k = j - i$ and the order of the loops is determined, and the algorithm starts from $k = 1$

```
void OptMatrix( const long r[ ], int N, TwoDimArray M, TwoDimArray LastChange )
{
   int  i, j, k, L;
   long  ThisM;
   for( i = 1; i <= N; i++ )   M[ i ][ i ] = 0;
   for( k = 1; k < N; k++ ) /* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;
           M[ i ][ j ] = Infinity;
           for( L = i; L < j; L++ ) {
               ThisM = M[ i ][ L ] + M[ L + 1 ][ j ] + r[ i - 1 ] * r[ L ] * r[
j ];
               if ( ThisM < M[ i ][ j ] ){ /* Update min */
                   M[ i ][ j ] = ThisM;
                   LactChange[ i ][ j ] = L;
               }
           }  /* end for-L */
       }  /* end for-Left */
}
```

> The code here also records the optimal ordering rather than merely records the optimal number
>
> The order of traversing possibilities here is more accurate than recursion
>
> It runs in $O(N^3)$ time but is still practical comparing to the cost of matrix multiplication

Optimal Binary Search Tree

- do static binary searching
- the cost to access a node at depth d is $d+1$
- given N distinct words each with a probability p_i of being accessed
- the words are **lexicologically ordered** at first
- minimize $\sum_{i=1}^{N} p_i(d_i+1)$
    - greedy: the tree may be terribly unbalanced
    - AVL: the nodes with high probabilities may be deep

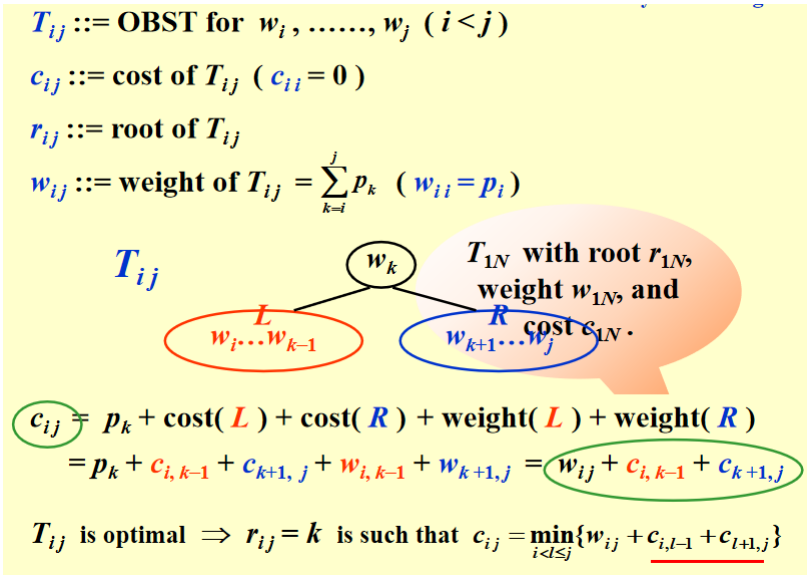
solution:

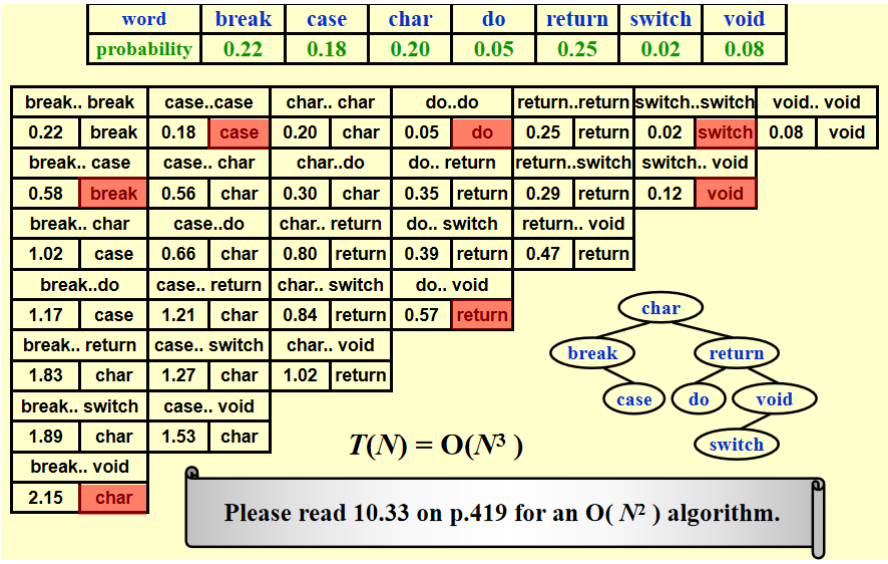
- State: r_{ij} and c_{ij}, the root of a subtree and the total cost of a subtree

$$c_{ij} = \begin{cases} 0 & i = j \\ \min\limits_{i<l\le j}\{w_{ij} + c_{l,k-1} + c_{k+1,j}\} & i \neq j \end{cases}$$

$$\text{update } r_{ij}$$

- every time the root of 2 subtrees is found, the total height is incremented and the total cost is added by w_{ij}





> The calculation goes top-down
>
> The last calculation, where $i = 1, j = N$, determines the root of the final tree. The sub-roots are determined upwards

All-Pairs Shortest Path (Floyd)

- find shortest paths between all pairs of the graph

solution:

- State: k, i, j
  - the k-th loop or the shortest path found in the k-th loop passes only the first k vertices
  - i, j are the start and the end

$$D^k[i][j] = \min\{D^{k-1}[i][j],\ D^{k-1}[i][k] + D^{k-1}[k][j]\} \quad \text{for each } i,\ j$$

-

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 Use single-source algorithm for |V| times.

$T = O(|V|^3)$ – works fast on sparse graph.

Method 2 Define

$D^k[i][j] = \min\{$ length of path $i \to \{l \le k\} \to j\}$

and $D^{-1}[i][j] = \text{Cost}[i][j]$. Then the length of the shortest path from i to j is $D^{N-1}[i][j]$.

Algorithm Start from D^{-1} and successively generate $D^0, D^1, \ldots, D^{N-1}$. If D^{k-1} is done, then either

① $k \notin$ the shortest path $i \to \{l \le k\} \to j \Rightarrow D^k = D^{k-1}$; or

② $k \in$ the shortest path $i \to \{l \le k\} \to j$

$= \{$ the S.P. from i to $k\} \cup \{$the S.P. from k to $j\}$

$\Rightarrow D^k[i][j] = D^{k-1}[i][k] + D^{k-1}[k][j]$

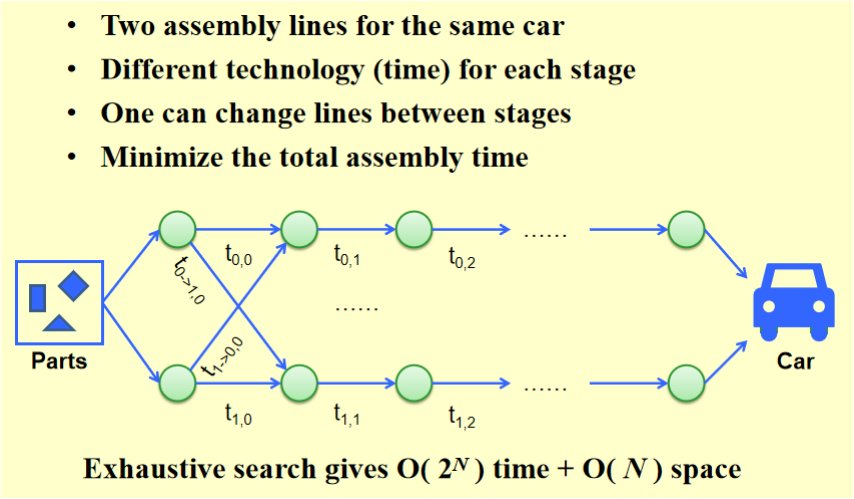
$\therefore \quad D^k[i][j] = \min\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}, k \ge 0$

- $O(N^3)$, faster in a dense graph

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
void AllPairs( TwoDimArray A, TwoDimArray D, int N ) {
    int i, j, k;
    for ( i = 0; i < N; i++ )  /* Initialize D */
         for( j = 0; j < N; j++ )
                D[ i ][ j ] = A[ i ][ j ];
    for( k = 0; k < N; k++ )  /* add one vertex k into the path */
         for( i = 0; i < N; i++ )
             for( j = 0; j < N; j++ )
                if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] ) /* Update shortest
path */
                    D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

> Works if there are negative edge costs, but no negative-cost cycles.
>
> The history information is left in the matrix

Product Assembly



- only the last stages of the two lines matter

solution:

- State: $t_{line,k}$
  - $line$: represent the 2 lines
  - k: represent the stage, or how many components have been made

$$t_{line,k} = \min\{t_{stay,k-1} + t_{line,k-1},\ t_{move,k-1} + t_{line,k-1}\}$$

```
f[0][0]=0; L[0][0]=0;
f[1][0]=0; L[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[line][stage-1]  + t_process[line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;
      L[line][stage] = line;
    }
    else {
      f[line][stage] = f_move;
      L[line][stage] = 1-line;
    }
  }
}
```

Greedy

- works only if the local optimum is equal to the global optimum
- an optimal solution is not guaranteed
- a solution close to the optimal one can be guaranteed
- proof
  - the first solution produced is included in the optimal solution

- optimal sub-structure property — the optimal solution derives from the optimal solution of the sub-problem (contradiction usually)
- *mathematical induction*
- 贪心算法一定能得到最优解的证明【活动安排问题举例】
- 漫谈算法（一）如何证明贪心算法是最优 using exchange argument

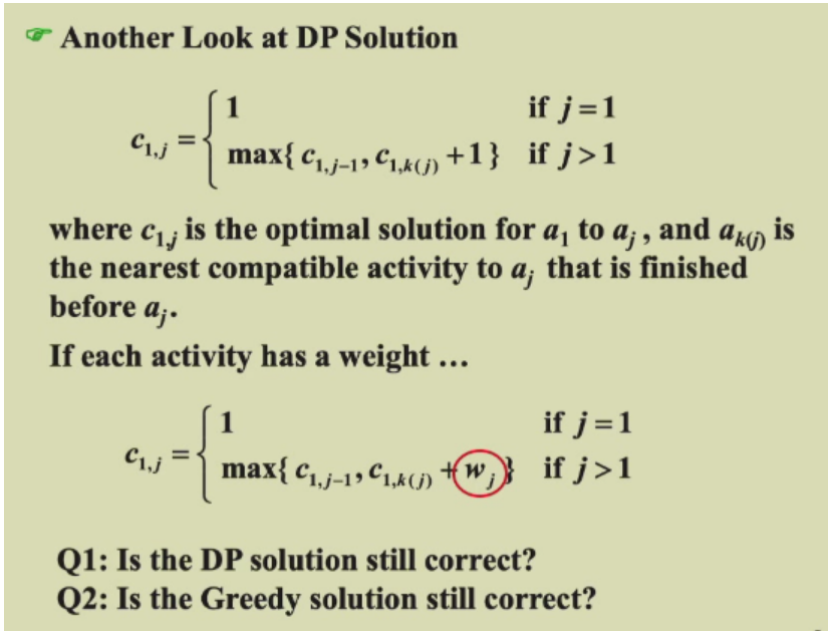
Activity Selection:

从 活动选择问题 看动态规划和贪心算法的区别与联系

Rules:

1. Select the interval which ends earliest
2. Optimal solutions

Another DP solution:

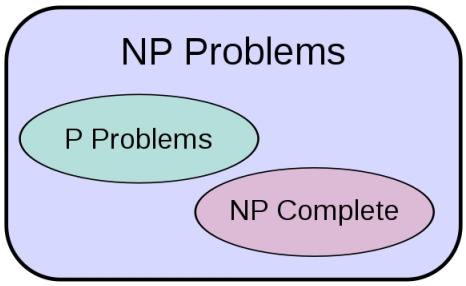


Huffman Encode:

Key: the code of a character is never the prefix of another

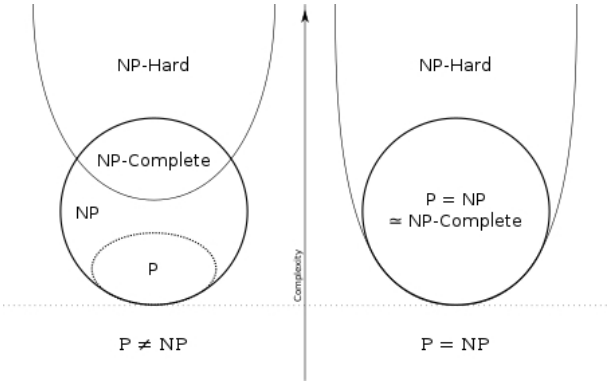
NP-Completeness

- Polynomial: $O(1)$, $O(N^k)$, $O(N \log N)$
- Super-Polynomial: $O(k^N)$, $O(N!)$
- P: some algorithm can provide an answer in polynomial time
- NP: we can prove any solution is *true* in *polynomial* time
  - worrying about **verification** but not finding a solution

- 

- NPC: any problem in NP can be *polynomially reduced* to it
  - the conversion takes polynomial time
  - polynomial reduction: transforming a problem α into a problem β takes $O(N^{k_1})$ time and solving β takes $O(N^{k_2})$ (β is usually in NPC and harder)
  - an NPC problem is always *harder* than a NP problem
- complexity class co-NP: the set of languages L such that $\bar{L} \in NP$
- NP-Hard: a problem *H* is NP-hard when every problem *L* in NP can be reduced in polynomial to *H*

> 对 NP-Hard问题和NP-Complete问题的一个直观的理解就是指那些很难（很可能是不可能）找到多项式时间算法的问题。因此一般初学算法的人都会问这样一个问题：NP-Hard和NP-Complete有什么不同？简单的回答是根据定义，如果所有NP问题都可以多项式归约到问题A，那么问题A就是 NP-Hard；如果问题A既是NP-Hard又是NP，那么它就是NP-Complete。从定义我们很容易看出，NP-Hard问题类包含了NP- Complete类。但进一步的我们会问，是否有属于NP-Hard但不属于NP-Complete的问题呢？答案是肯定的。例如停机问题，也即给出一个程序和输入，判定它的运行是否会终止。停机问题是不可判的，那它当然也不是NP问题。但对于SAT这样的NP-Complete问题，却可以多项式归约到停机问题。因为我们可以构造程序A，该程序对输入的公式穷举其变量的所有赋值，如果存在赋值使其为真，则停机，否则进入无限循环。这样，判断公式是否可满足便转化为判断以公式为输入的程序A是否停机。所以，停机问题是NP-Hard而不是NP-Complete。



[non-deterministic Turing machine]

- the next programme is arbitrarily chosen
- the machine should contain no contradictions
- the machine may never halt in some branch

Approximation

- To deal with hard problems

> 3 ways for hard problems:

> - *Exploiting special problem structure*: perhaps we do not need to solve the general case of the problem but rather a tractable special version;
> - *Heuristics*: procedures that tend to give reasonable estimates but for which no proven guarantees exist;
> - *approximation algorithms*: procedures which are proven to give solutions within a factor of optimum.

Approximation ratio:

An algorithm has an *approximation ratio* of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of a optimal solution:

$$\max(\frac{C}{C^*}, \frac{C^*}{C}) \leq \rho(n)$$

The algorithm is called $\rho(n)$-*approximation algorithm*

Approximation scheme:

An *approximation scheme* for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1+\epsilon)$-*approximation algorithm*.
We say that an approximation scheme is a *polynomial-time approximation scheme(PTAS)* if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance, like $O(n^{2/\epsilon})$
If the scheme runs in time polynomial in both n and ϵ of its input instance, it's a *fully polynomial-time approximation scheme(FPTAS)*, like $O((1/\epsilon)^2 n^3)$

> proof: $OPT \leq ALG$ is somehow self-evident

Approximate Bin Packing

P357

- in *Greedy*
- the lower bound: $\lceil \frac{\sum item}{capacity} \rceil$
- 2 types: on-line & off-line

On-line:

- `NextFit()`: consider the bin containing the last item
  - $1/2, 1/2+\epsilon, 1/2, 1/2+\epsilon \ldots$
  - 2
- `FistFit()`: find the first used bin that is able to contain the item
  - $(1/7+\epsilon)\times 6, (1/3+\epsilon)\times 6, (1/2+\epsilon)\times 6 \ldots$
  - 1.7
- `BestFit()`: consider the tightest spot
  - 1.7
- On-line algorithm is at least 5/3 and no on-line algorithm can always give an optimal solution

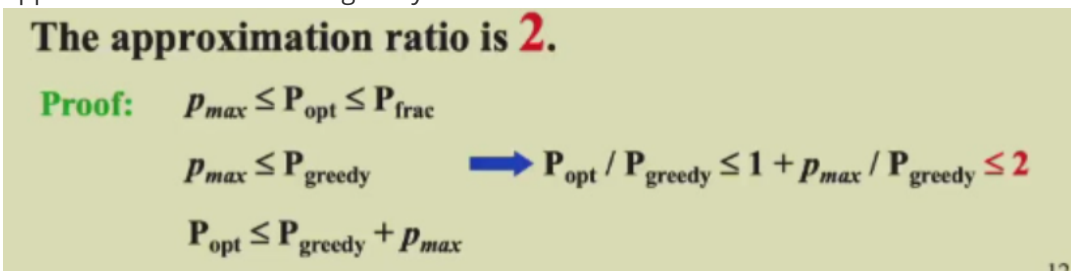
Off-line:

- sort into a non-increasing sequence

- apply the algorithm above

Knapsack Problem

- Fractional version
  - profit density
- 0-1 version
  - dynamic programming
  - approximation ratio is 2 for greedy

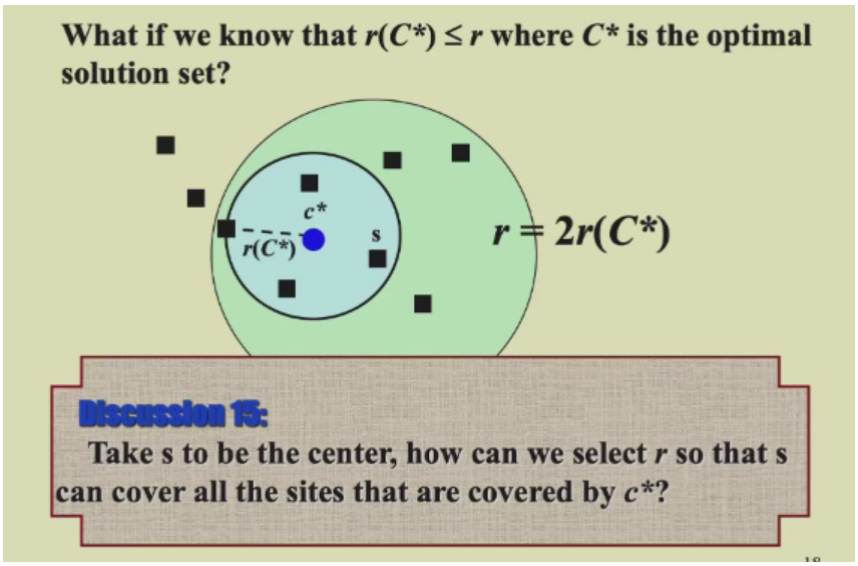


The K-center Problem

- Use K circles of radius r to cover all the sites

Greedy:

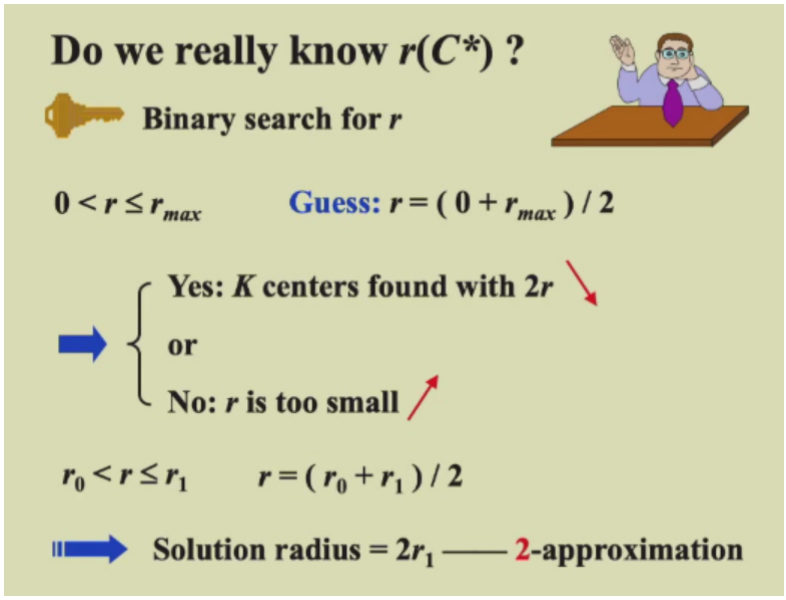
- The centers can only locate at the positions of the sites
- The radius picked in this way is no more than 2 times the optimal radius



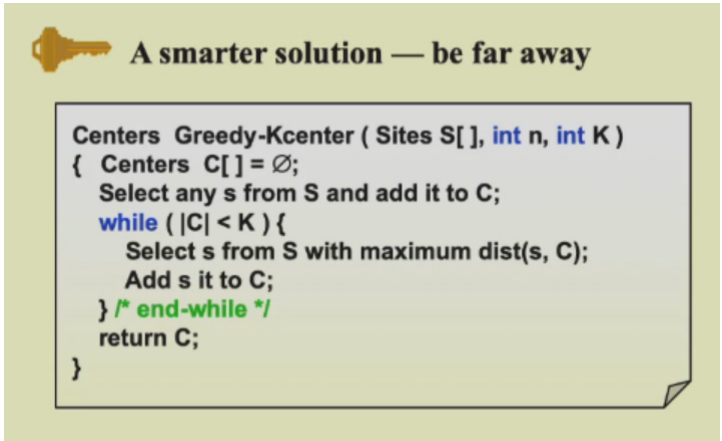
- Randomly pick a center, remove the sites covered and count the number

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r )
{  Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
   Centers  C[ ] = ∅;
   while ( S'[ ] != ∅ ) {
      Select any s from S' and add it to C;
      Delete all s' from S' that are at dist(s', s) ≤ 2r;
   } /* end-while */
   if ( |C| ≤ K ) return C;
   else ERROR(No set of K centers with covering radius at most r);
}
```

> Suppose the algorithm selects more than K centers. Then for any set C^* of size at most K, the covering radius is $r(C^*) > r$
>
> Proof: triangle; at least one circle of C^* covers at least 2 centers of those given by this algorithm



From this theorem:

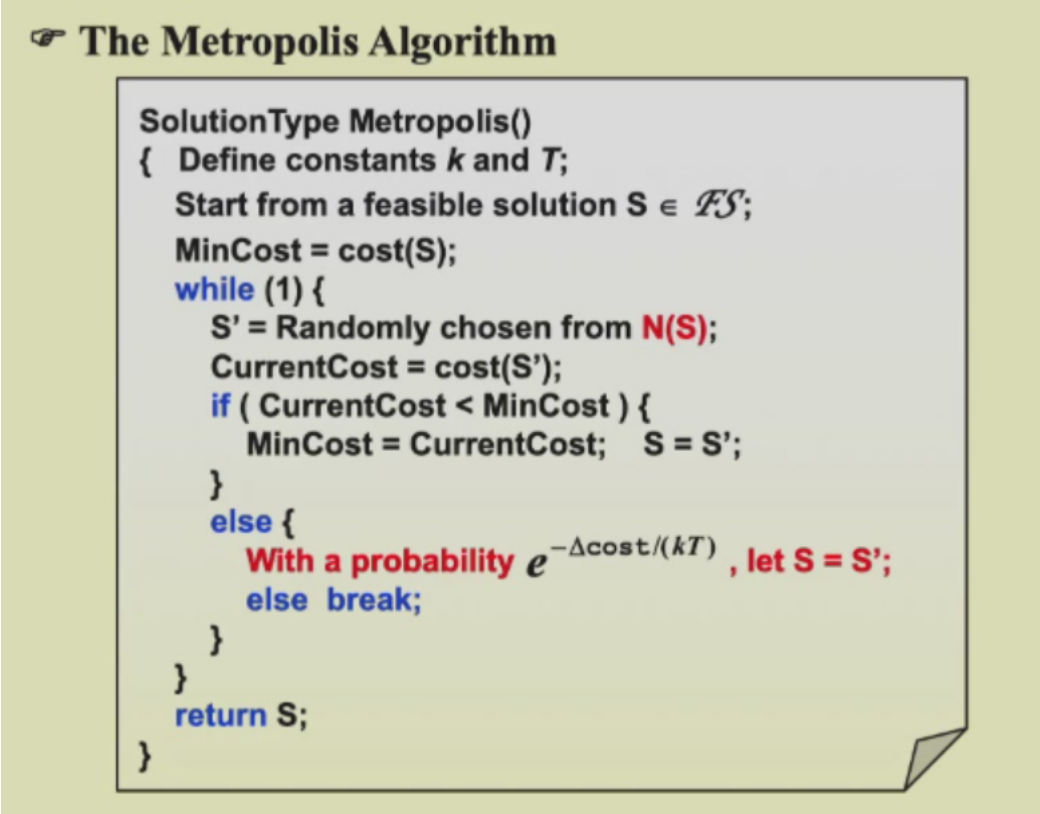


Then check the validity and r — 2-approximation problem

> This algorithm returns a set C of K centers such that $r(C) \leq 2r(C^*)$
>
> $|C| \leq K$ and so the theorem has not been violated, so according to the definition of the algorithm we can know the r we can find consequently satisfies $r(C) \leq 2r(C^*)$

Local Search

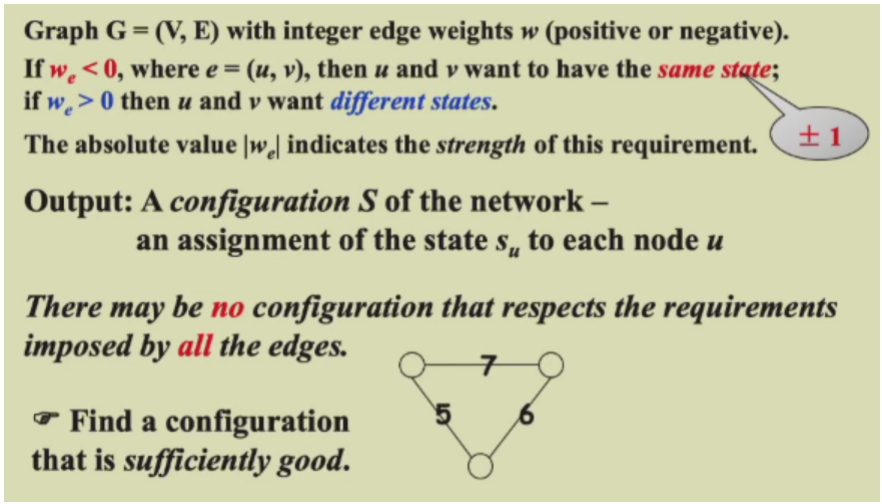
- Use local optimum to approximate global optimum
- Start with a feasible solution and search for a better one
- *Gradient Descent* — a better one: *Metropolis Algorithm*



> - Even the solution in the neighborhood is not better, there is a certain probability that it is accepted, to avoid being trapped in the local optimum
> - If the solution is close to global optimum, k, T can be set small

Hopfield Neural Networks

Definition:

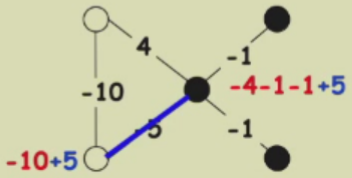


〖Definition〗 In a configuration S, edge $e = (u, v)$ is *good* if $w_e s_u s_v < 0$ ($w_e < 0$ iff $s_u = s_v$); otherwise, it is *bad*.

〖Definition〗 In a configuration S, a node u is *satisfied* if the weight of incident good edges ≥ weight of incident bad edges.

$$\sum_{v: e=(u,v)\in E} w_e s_u s_v \le 0$$

〖Definition〗 A configuration is *stable* if all nodes are satisfied.



✓ *Does a Hopfield network always have a stable configuration, and if so, how can we find one?*

- Problem: To maximize Φ
- Feasible solution set $\mathcal{FS}$: configurations
- S ~ S': S' can be obtained from S by flipping a single state

To find such a stable configuration, we use state-flipping algorithm:

```
ConfigType State_flipping()
{
   Start from an arbitrary configuration S;
   while ( ! IsStable(S) ) {
      u = GetUnsatisfied(S);
      s_u = - s_u;
   }
   return S;
}
```

Proof:

Claim: The state-flipping algorithm terminates at a stable configuration after *at most* $W = \Sigma_e |w_e|$ iterations.

Proof: Consider the measure of progress

$$\Phi(S) = \Sigma_{e \text{ is good}} |w_e|$$

When u flips state (S becomes S'):
- all good edges incident to u become bad
- all bad edges incident to u become good
- all other edges remain the same

$$\Phi(S') = \Phi(S) - \sum_{\substack{e: e=(u,v)\in E \\ e \text{ is bad}}} |w_e| + \sum_{\substack{e: e=(u,v)\in E \\ e \text{ is good}}} |w_e| \ \ge \ \Phi(S) + 1$$

Clearly $0 \le \Phi(S) \le W$ ∎

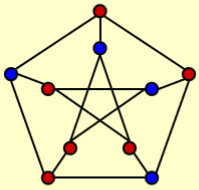
> Any local maximum in the state-flipping algorithm to maximize Φ is a stable configuration

Maximum Cut

Definition:

❖ **Maximum Cut problem:** Given an undirected graph G = (V, E) with positive integer edge weights w_e, find a node partition (A, B) such that the total weight of edges crossing the cut is maximized.

$$w(A, B) := \sum_{u \in A,\, v \in B} w_{uv}$$



- **Toy application**
  - n activities, m people.
  - Each person wants to participate in two of the activities.
  - Schedule each activity in the morning or afternoon to maximize number of people that can enjoy both activities.
- **Real applications** Circuit layout, statistical physics

- Problem: To maximize $w(A, B)$
- Feasible solution set $\mathcal{FS}$: any partition (A, B)
- S ~ S': S' can be obtained from S by moving one node from A to B, or one from B to A

Proof:

Claim: Let (A, B) be a local optimal partition and let (A^*, B^*) be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.

Proof: Since (A, B) is a local optimal partition, for any $u \in A$

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

Summing up for *all* $u \in A$

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

$$w(A^*, B^*) \leq \sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} + w(A, B) \leq 2w(A, B)$$

_

> The second inequation can be obtained in the same way as the first one
>
> A better bound can be proven

Parallel Algorithm

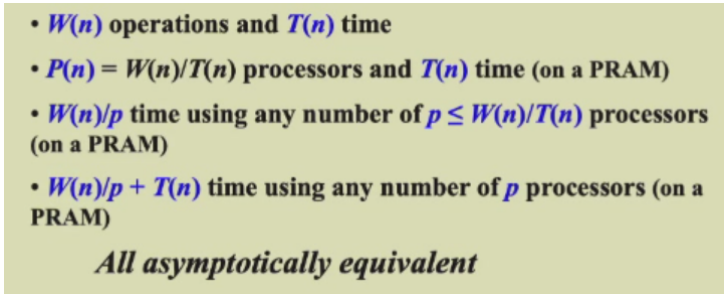
a reference

To resolve access conflicts:

- Exclusive-Read Exclusive-Write
- Concurrent-Read Exclusive-Write
- Concurrent-Read Concurrent-Write
  - Arbitrary rule
  - Priority rule
  - Common rule

Measuring the performance:

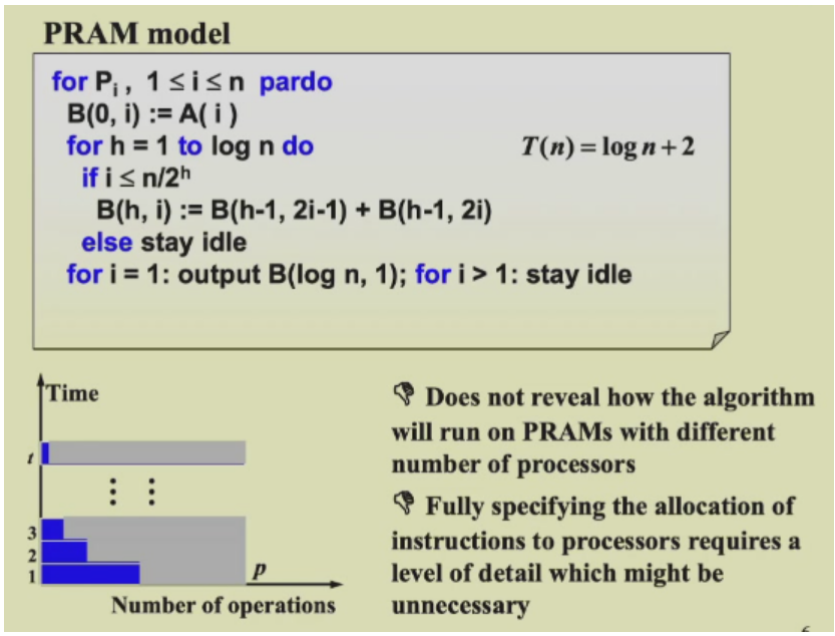
- Work load - total number of operations $W(n)$
- Worst-case running time $T(n)$



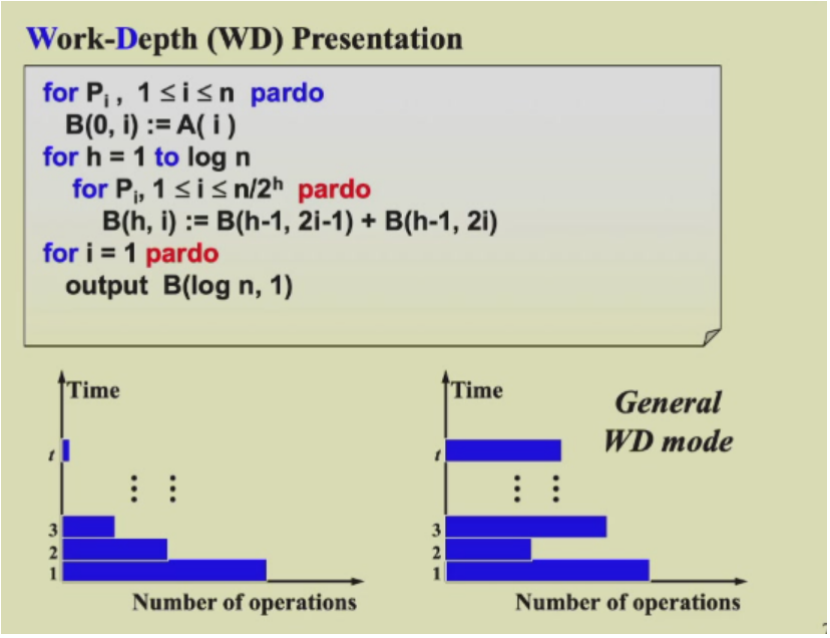
Parallel Random Access Algorithm (PRAM)

Summation

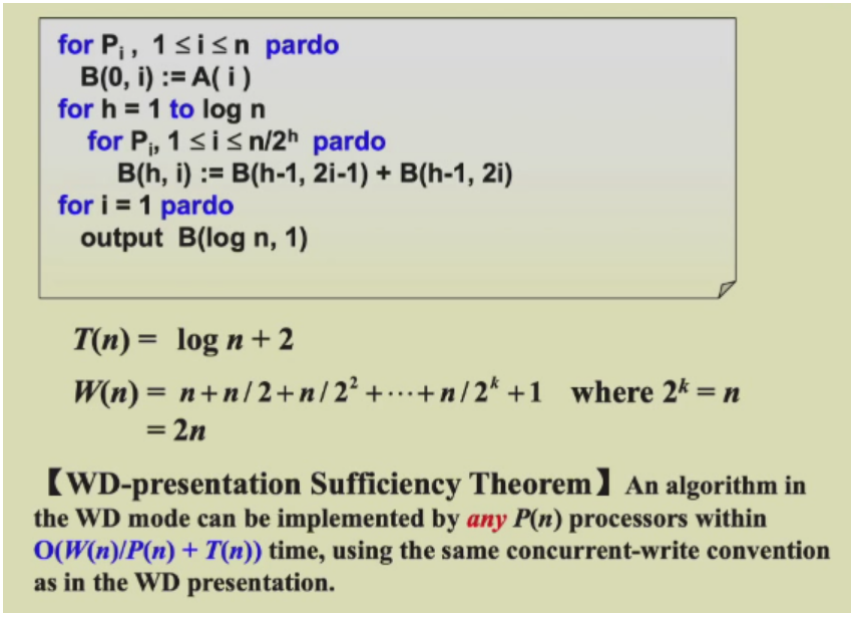
An impractical version:



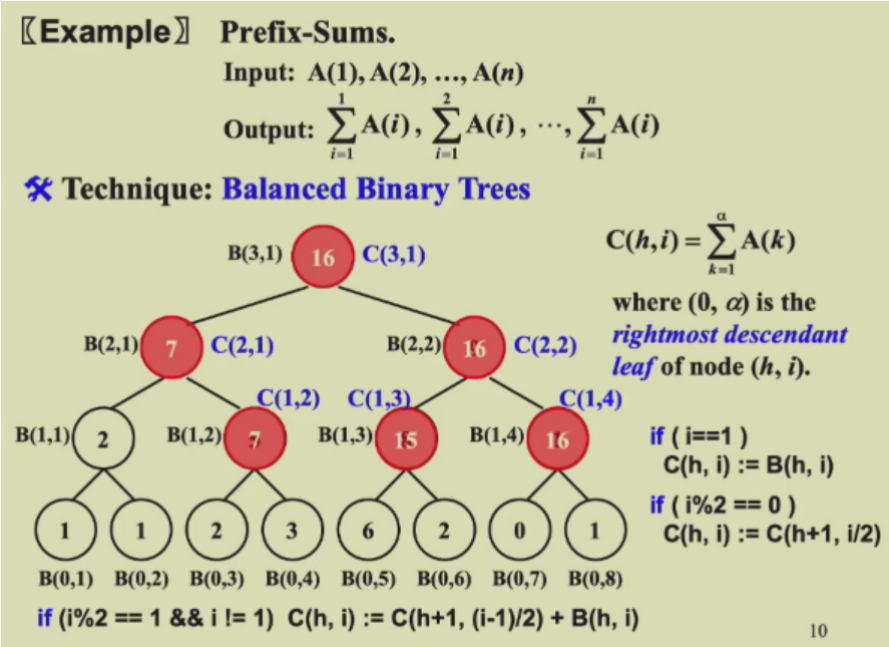
A better Work-Depth Presentation:



Performance analysis:



Prefix-Sums



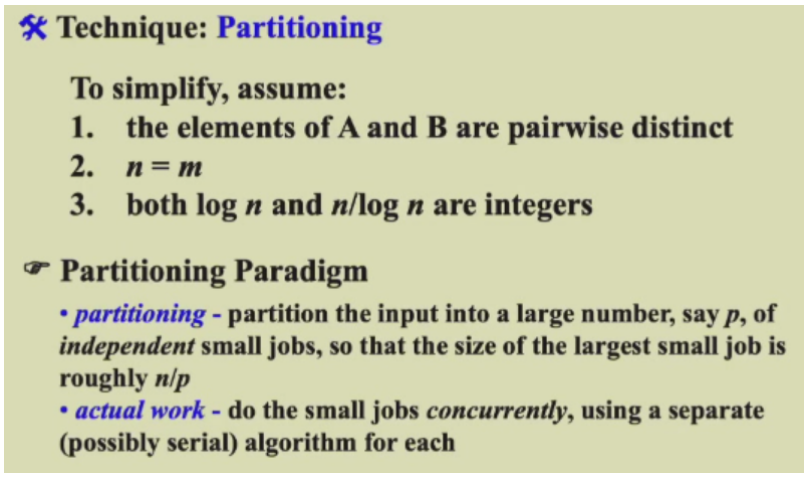
1. calculate $B(h,i)$ from bottom up
2. calculate $C(h,i)$ from top down

```
for P_i , 1 ≤ i ≤ n pardo
  B(0, i) := A(i)
for h = 1 to log n
  for i , 1 ≤ i ≤ n/2^h pardo
    B(h, i) := B(h - 1, 2i - 1) + B(h - 1, 2i)
for h = log n to 0
  for i even, 1 ≤ i ≤ n/2^h pardo
    C(h, i) := C(h + 1, i/2)
  for i = 1 pardo
    C(h, 1) := B(h, 1)
  for i odd, 3 ≤ i ≤ n/2^h pardo
    C(h, i) := C(h + 1, (i - 1)/2) + B(h, i)
for P_i , 1 ≤ i ≤ n pardo
  Output C(0, i)
```

$T(n) = O(\log n)$ $W(n) = O(n)$

Merging

> Technique: Partitioning



make use of ranking:

Merging ➡ Ranking

$RANK(j, A) = i$, if $A(i) < B(j) < A(i+1)$, for $1 \le i < n$
$RANK(j, A) = 0$, if $B(j) < A(1)$
$RANK(j, A) = n$, if $B(j) > A(n)$

The *ranking problem*, denoted RANK(A,B) is to compute:
1. RANK(i, B) for every $1 \le i \le n$, and
2. RANK(i, A) for every $1 \le i \le n$

Claim: Given a solution to the ranking problem, the merging problem can be solved in O(1) time and $O(n+m)$ work.

```
for P_i, 1 ≤ i ≤ n pardo
   C(i + RANK(i, B)) := A(i)
for P_i, 1 ≤ i ≤ n pardo
   C(i + RANK(i, A)) := B(i)
```

- use binary search to rank: demand many processors
- use serial ranking to rank: cannot be parallelized
- parallel ranking

☞ **Parallel Ranking**

Assume that $n = m$; and that $A(n+1)$ and $B(n+1)$ are each larger than both $A(n)$ and $B(n)$.

Stage 1: Partitioning $p = n / \log n$

- A_Select(i) = A(1+(i-1)log$n$) for $1 \le i \le p$ $T = O(\log n)$
- B_Select(i) = B(1+(i-1)log$n$) for $1 \le i \le p$ $W = O(p \log n) = O(n)$
- Compute RANK for each *selected* element

A	4	6	8	9	16	17	18	19	20	21	23	25	27	29	31	32
B	1	2	3	5	7	10	11	12	13	14	15	22	24	26	28	30

Stage 2: Actual Ranking

At most $2p$ smaller sized (O(log$n$)) problems.

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

$$T = O(\log n), \quad W = O(n)$$

Maximum Finding

Brute and pairwise search:

☞ **Replace "+" by "max" in the summation algorithm**

$$T(n) = O(\log n), \quad W(n) = O(n)$$

☞ **Compare all pairs**

```
for P_i, 1 ≤ i ≤ n  pardo
   B(i) := 0
for i and j, 1 ≤ i, j ≤ n  pardo
   if ( (A(i) < A(j)) || ((A(i) = A(j)) && (i < j)) )
         B(i) = 1
   else B(j) = 1
for P_i, 1 ≤ i ≤ n  pardo
   if B(i) == 0
     A(i) is a maximum in A
```

$$T(n) = O(1), \quad W(n) = O(n^2)$$

Discussion 21:
How to resolve access conflicts?

Doubly-logarithmic paradigm (partitioned by $\sqrt{n}$ recursively, master theorem):

☞ **A Doubly-logarithmic Paradigm**

Assume that $h = \log\log n$ is an integer ($n = 2^{2^h}$).

Partition by $\sqrt{n}$:

$$\begin{aligned}
\mathrm{A}_1 &= \mathrm{A}(1), & \cdots, \quad \mathrm{A}(\sqrt{n}) &\Rightarrow M_1 \sim T(\sqrt{n}), W(\sqrt{n}) \\
\mathrm{A}_2 &= \mathrm{A}(\sqrt{n}+1), & \cdots, \quad \mathrm{A}(2\sqrt{n}) &\Rightarrow M_2 \sim T(\sqrt{n}), W(\sqrt{n}) \\
&\cdots \quad \cdots & & \\
\mathrm{A}_{\sqrt{n}} &= \mathrm{A}(n-\sqrt{n}+1), & \cdots, \quad \mathrm{A}(n) &\Rightarrow M_{\sqrt{n}} \sim T(\sqrt{n}), W(\sqrt{n})
\end{aligned}$$

$$M_1, M_2, \cdots, M_{\sqrt{n}} \Rightarrow \mathrm{A}_{\max} \sim T = O(1),\ W = O(\sqrt{n}^2) = O(n)$$

$$T(n) \le T(\sqrt{n}) + c_1, \quad W(n) \le \sqrt{n}\, W(\sqrt{n}) + c_2 n$$

$$\Rightarrow T(n) = O(\log\log n), \quad W(n) = O(n \log\log n)$$

Doubly-logarithmic paradigm (partitioned by $h = \log\log n$ recursively, make use of the result of the above partition):

☞ **A Doubly-logarithmic Paradigm**

Partition by $h = \log\log n$:

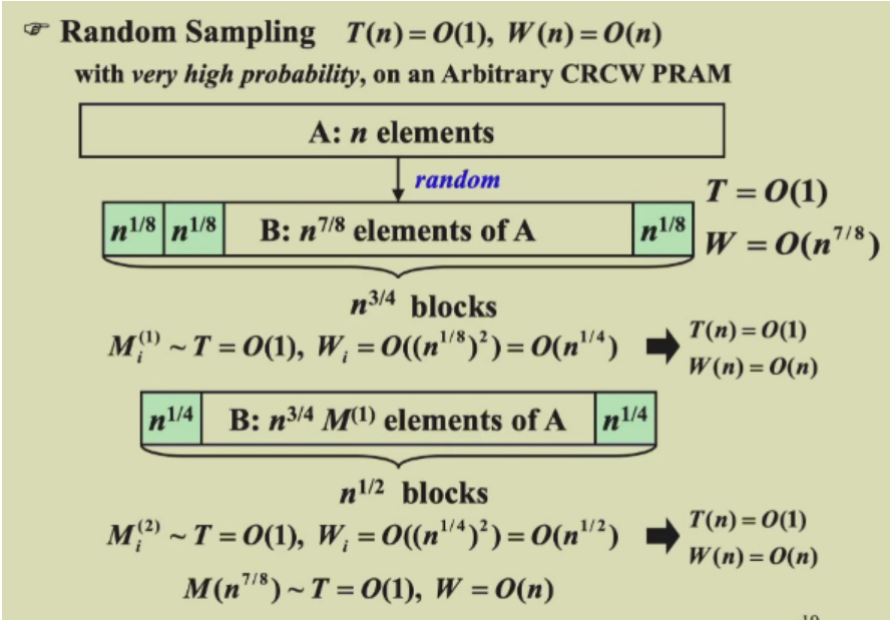
$$\begin{aligned}
\mathrm{A}_1 &= \mathrm{A}(1), & \cdots, \quad \mathrm{A}(h) &\Rightarrow M_1 \sim O(h) \\
\mathrm{A}_2 &= \mathrm{A}(h+1), & \cdots, \quad \mathrm{A}(2h) &\Rightarrow M_2 \sim O(h) \\
&\cdots \quad \cdots & & \\
\mathrm{A}_{n/h} &= \mathrm{A}(n-h+1), & \cdots, \quad \mathrm{A}(n) &\Rightarrow M_{n/h} \sim O(h)
\end{aligned}$$

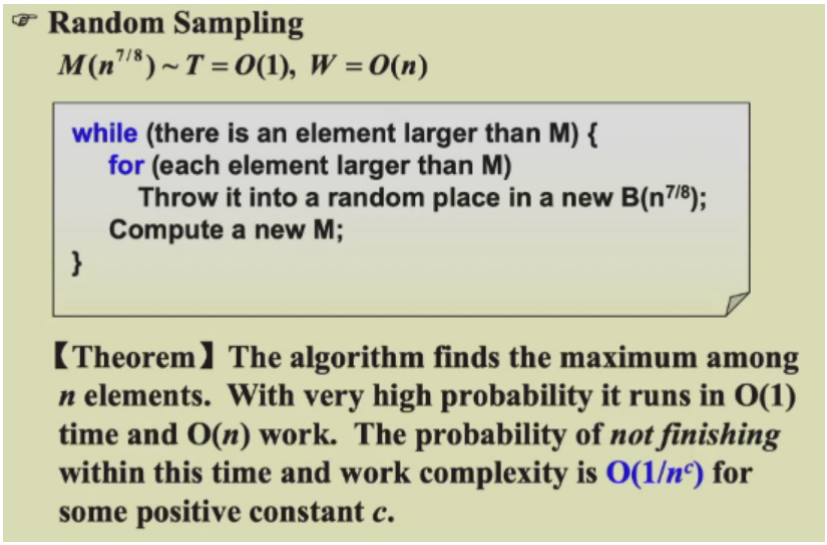
$$M_1, M_2, \cdots, M_{n/h} \Rightarrow \mathrm{A}_{\max}$$

$$T(n) = O(h + \log\log(n/h)) = O(\log\log n)$$

$$W(n) = O(h \times (n/h) + (n/h)\log\log(n/h)) = O(n)$$

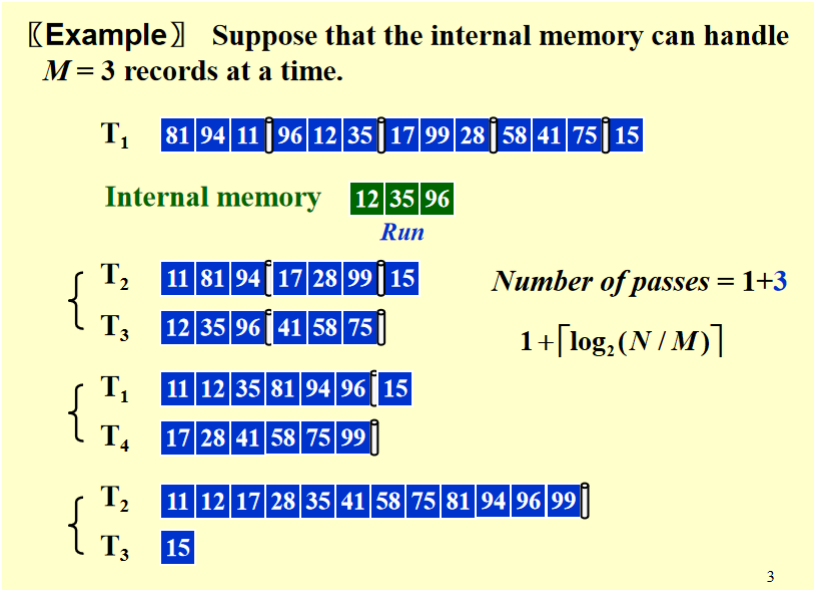
Random Sampling





External Sort

- make use of mergesort



How to reduce the number of passes?

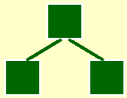
Use a *k-way* merge!

T_1 81 94 11 | 96 12 35 | 17 99 28 | 58 41 75 | 15

T_2 11 81 94 | 41 58 75

T_3 12 35 96 | 15

T_4 17 28 99



T_1 11 12 17 28 35 81 94 96 99

T_5 15 41 58 75

T_6

$$\textit{Number of passes} = 1 + \lceil \log_k (N / M) \rceil$$

Require $2k$ tapes!

Discussion 20:
What will happen if 22 runs are placed on T_2, with 12 on T_3 ?

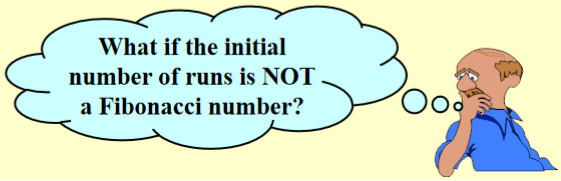
Claim: If the number of runs is a Fibonacci number F_N, then the best way to distribute them is to split them into F_{N-1} and F_{N-2} .

Claim: For a k-way merge, $F_N^{(k)} = F_{N-1}^{(k)} + \Lambda + F_{N-k}^{(k)}$

where $F_N^{(k)} = 0 \ \ (0 \le N \le k-2), \ F_{k-1}^{(k)} = 1$

Polyphase Merge

$k+1$ tapes only



> use some vacuum runs to make a *Fibonacci* number

2-way merge

input buffers output buffer

In general, for a k-way merge we need $2k$ input buffers and 2 output buffers for parallel operations.

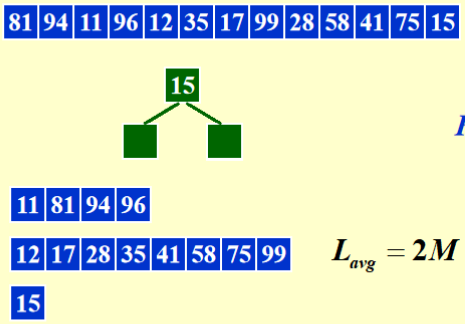
k ↑ → # of input buffers ↑ → buffer size ↓

↑ seek time ← ↓ block size on disk ↓

Beyond a certain k value, the I\O time would actually increase despite the decrease in the number of passes being made. The optimal value for k clearly depends on disk parameters and the amount of internal memory available for buffers.

> the increase of k will increase the time for hardware and reduce the passes

Can we generate a longer run?

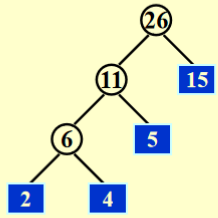


Powerful when input is often *nearly sorted* for external sorting.

How to minimize the merge time?

〖Example〗 Suppose we have 4 runs of length 2, 4, 5, and 15, respectively. How can we arrange the merging to obtain minimum merge times?

Huffman Tree!



$2 \times 3 + 4 \times 3 + 5 \times 2 + 15 \times 1 = 43$

Total merge time = O (*the weighted external path length*)