数据结构与算法分析

华中科技大学软件学院

2014年秋

大纲

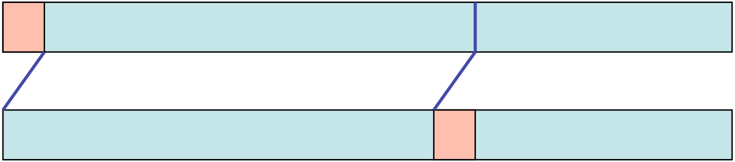
实验五

- First need to locate the root node: first element in the pre-order traversal sequence
- Root partitions the in-order sequence into three parts: {left sub-tree sequence}root{right sub-tree sequence}
- The above gives a partition in the pre-order sequence respectively
- Use the left sub-tree and right sub-tree traversal results to construct sub-trees recursively

定位根结点

```
static int find_root (char a[], int n, char name)
{
    int i;

    for (i = 0; i < n; i++)
    {
        if (a[i] == name)
            return (i);
    }

    /* not found */
    return (-1);
}
```

树的划分



构建二叉树

```
static NODE *construct (char a[], char b[], int n)
{
    NODE *root;
    int k, l;

    /* trivial case: tree is empty */
    if (n == 0)
        return (NULL);

    root = (NODE *) malloc (sizeof (NODE));
    if (root == NULL)
        return (root);

    root->name = a[0];
    k = find_root (b, n, a[0]);
    if (k < 0)
    {
        printf ("node␣%c␣not␣found␣in␣in-order␣traversal\n", a[0]);
        return (NULL);
    }

    root->left = construct (&a[1], b, k);
    root->right = construct (&a[k + 1], &b[k + 1], n - k - 1);

    return (root);
}
```

复杂度分析

- Trivial case: T(0) = 1
- Recursive step: T(n) = T(k) + T(n - k-1) + non-recursive operations
  - Best case: only right sub-trees, T(n) = T(n-1) + O(1), O(n)
  - Worst case: only left sub-trees, T(n) = T(n-1) + O(n), $O(n^2)$
  - Balanced case: T(n) = 2T(n/2) + O(n), $O(n\log n)$

课程计划

- 已经学习了
  - 树的表示
  - 二叉树与遍历
  - BST的操作
  - AVL树

课程计划

- 已经学习了
  - 树的表示
  - 二叉树与遍历
  - BST的操作
  - AVL树
- 即将学习
  - 散列
  - 冲突的处理
  - 再散列

Roadmap

为什么需要散列

- Binary search on sorted list takes time log n
- But consider array access: A[i]
  - Given i, compute address, say Say, &A[0]+i*4 for integers
  - Then access value at &A[0]+i*4
- Constant + constant = constant time
- If keys to elements are integers, then store each element in position A[key]
- Can search by key in constant time

折半查找与数组

- 数组访问比折半查找还快
- 折半查找假定所有元素已被排序
- 如果所有的关键字都在范围之内，为什么不对应地存入数组？这样就可以在常数时间里访问
- 问题在于：
  - 通常，关键字可能超出存储空间的范围
  - 有时关键字也不是数字类型，例如，可能是字符串
- 引入Hashing解决上述问题

Hashing

- Another data structure for dictionary probing: given set of items with number keys, support fast insert, delete, search
- Not supported:
  - Fast findMin, findMax, print-in-sorted-order
- Hash table: array[0..n−1] + hash function
  - Function maps item x to number in [0..n−1]
  - Insert: store x in pos h(x)
  - Search: seek x in pos h(x)

Hashing的效果

- Array access is constant time
- If array size >= key count, then room for all
- Access algorithm:
  - Compute h(x) - const time
  - Compute address h(x) - const time
  - Access memory element A[h(x)] - const time
- Array access is O(1)
  - Very fast
  - Better than log n

选择哈希函数

- Goal: use all array entries equally
  - Hash function spreads outputs around
  - Input is “hashed” – scrambled up
- Inputs are unlimited, keys are unlimited
- Array is of some size
- Multiple items per array cell
  - Try to divide up evenly
  - Each cell = linked list
  - Access time= $O(\text{keycount}/\text{arraysize}) = O(1)$
- Now: choose hash function

哈希函数

- Map integer keys to [0..size−1]
- Obvious idea: modulus, h(x) = x % size
- Eg: hash prices in cents to size 1000 table
  19.95 → 995, 29.95 → 995, 39.95 → 995
- If, say, keys are multiples of size, then all mapped to A[0]
- Problem: structure in keys remains in hashes
- Could call random number generator for each h(x), but h(x) must be same later
- Better (easy) solution: choose size = large prime, say 1007
  19.95 → 998, 29.95 →981, 39.95→ 974

设计哈希函数

- Features a good hash function should have
  - Low cost
  - Uniformity
  - Deterministic, continuity/discontinuity
- Other applications than table lookup
  - Cryptography
    - Collision free
    - Easy to compute
    - Hard to invert
  - Data integrity
    - Message digest
    - Checksum
  - Digital signature

哈希函数的特性

- Compression – h maps an input x of arbitrary bit length into a fixed number of bits h(x)
- Ease of computation – given h and x, it is easy to compute h(x)
- Pre-image resistance – given y, it is computationally infeasible to find x such that h(x) = y
- Collision resistance – it is computationally infeasible to find two distinct x and x' such that h(x) = h(x')

一些哈希函数

Name	Bitlength	Rounds × Steps per round	Relative speed
MD4	128	3 × 16	1.00
MD5	128	4 × 16	0.68
RIPEMD-128	128	4 × 16 twice (in parallell)	0.39
SHA-1	160	4 × 20	0.28
RIPEMD-160	160	5 × 16 twice (in parallel)	0.24

冲突的频率

- How many collisions?
  - Try to keep few
  - very hard to avoid altogether
- Birthday problem: what's the probability that >=2 among n share birthday?
- Compute probability that all n are different and subtract

$$\frac{364}{365} \times \frac{363}{365} \times \ldots \times \frac{(366-n)}{365} = \frac{365!/(366-n)!}{365^n}$$

- Probability = 1 - $365!/(366-n)!/365^n$

Roadmap

哈希表中的冲突

- Assume hash maps item to cells evenly
- Each of m cells gets 1/m of items
  - Insert the first item x1 at h(x1), now get second
  - Probability of the different place = (m−1)/m
  - Probability of a different cell for the third: (m−2)/m
- Probability of all different: $(m-1)/m*(m-2)/m*\ldots*(m+1-n)/m = (m!/(m-n!))/m^n$
- Can show: $n = 1.177 * \sqrt{m} \rightarrow Pr(collision) > \frac{1}{2}$
- m = 1,000,000, n = 1177 $\rightarrow$ Pr(collision) > 0.5

分离链路法

- For all hash functions, choose one that minimizes collisions
- Other issue: how to handle collisions
- Simple idea: each element is a linked list
    - Insert: insert into cell's linked list
    - Search: find cell, walk through its list
- Large table, good function $\rightarrow$ short lists

哈希表

- H(x) = x % 7, insert 11, 12, 13, 14, 15, 16, 17, 18 , 19, 20

哈希表

- H(x) = x % 7, insert 11, 12, 13, 14, 15, 16, 17, 18 , 19, 20

0	->14	-> ‖	
1	->15	-> ‖	
2	->16	-> ‖	
3	->17	-> ‖	
4	->11	->18	-> ‖
5	->12	->19	-> ‖
6	->13	->20	-> ‖

装填因子

- Load factor λ= ratio (element count) / (table size) = average list length
- Unsuccessful search: λ comparisons
- Expected run time for a successful query: 1 + $\lambda/2$
  - 1 for each match
  - Average scan of the list before reaching the right position: $\lambda/2$

开放地址法

- No separate data structure
  - No creation of new nodes
  - On the other hand: need smaller λ <0.5
- Using a secondary function: if h(x) is full, try h1(x) = h(x)+f(1) % table size, h2(x)=h(x)+f(2) % table size, etc., until a free cell is found
- Has to use equal to check for the right one
  - f = "collision strategy"
  - Simplest strategy, linear probing: f(i) = i
- Eg: insert 89,18,49,58,69 into size-10 table

线性探测

- $h_i(x) = h(x) + i$, where h(x) = x % 10
- Insert 89, 18, 49, 58, 69

线性探测

- $h_i(x) = h(x) + i$, where h(x) = x % 10
- Insert 89, 18, 49, 58, 69

49
58
69
18
89

平方探测

- We can show: expected time is less for quadratic probing, $f(i) = i^2$
- Insert 89, 18, 49, 58, 96 this way
- Quadratic probing may fail to insert a key even if a table has not been filled
- Example, in a size-7 table, insert 0, 7, 14, 21 and 28, we will get repeated sequence of positions to query
- For n = 7*q + r (r = 0..6), there are only 4 possible values for n^2 % 7: 0, 1, 2, 4

平方探测定理

Theorem

As long as table size is prime and table at least half empty ($\lambda < 0.5$), never fails

Proof: for all $0 < i, j < 1/2$ floor(table size), $H(x) + i^2 = H(x) + j^2$(% TableSize) to $i^2 - j^2 = 0$ (% TableSize) $\rightarrow$ (i - j)*(i + j) = 0(% TableSize) $\rightarrow$ i = j, given TableSize is prime and i+j < TableSize. That is to say each x may be placed in 1/2 ceil(table size) positions (i = 0, 1, ⋯, 1/2 floor(table size) all lead to different probing positions), which cannot be all filled in a table with $\lambda < 0.5$.

平方探测中的删除

- Delete problem: if remove interim number, might lose (collided) number afterward, lazy deletion
- Deletion is always harder

Roadmap

再散列

- Access (all methods) takes longer as table becomes fuller
- Eventually: better off creating bigger table
- When too slow or insert fails
  - Do as with vector: create double-size table,
  - Scan through original table, compute (new) hashes, copy to right place in new table
- Amortized: + constant to each operation
- Amortization analysis: split cost over time
- Start with 16, 13, 15, 2, 27, 34, 66 in 0..9, rehash into 0..19

字符串作为关键字

- Strings as key is a common case
- Hash functions defined for integers, so must convert string $\rightarrow$ integer
- One simple idea: sum up ASCII values of chars in a string

哈希函数代码

```
int hash (char key[], int n, int size)
{
    int hash = 0;

    for (int i = 0; i < n; i++)
        hash += key[i];

    return (hash % size);
}
```

10 chars, about 128^{10} strings, but hash range: 0..1280 $\rightarrow$ < 1300 keys

改进的函数代码

$\sum_{i=0}^{n} s[n-i-1] * 31^{i}$, recall:
$k0 + 31 * k1 + 31^2 * k2 = ((k2) * 31 + k1) * 31 + k0$

```
int hash (char key[], int n, int size)
{
    int hash = 0;

    for (int i = 0; i < n; i++)
        hash = 31*hash + key[i];
    hash %= size;
    if (hash < 0)
        hash += size;

    return (hash);
}
```

使用随机数

```
unsigned char Rand8[256];     // This array contains a random
                                 permutation from 0..255 to 0..255
int Hash(char *x) {           // x is a pointer to the first char;
     int h;                   //     *x is the first character
     unsigned char h1, h2;

     if (*x == 0) return 0;   // Special handling of empty string
     h1 = *x; h2 = *x + 1;    // Initialize two hashes
     x++;                     // Proceed to the next character
     while (*x) {
         h1 = Rand8[h1 ^ *x]; // Exclusive-or with the two hashes
         h2 = Rand8[h2 ^ *x]; //     and put through the randomizer
         x++;
     }                        // End of string is reached when *x=0
     h = ((int)(h1)<<8) |     // Shift h1 left 8 bits and add h2
          (int) h2 ;
     return h ;               // Hash is concatenation of h1 and h2
}
```

Roadmap

伪随机数发生器

- 线性同余算法

$$X_t = (aX_{t-1} + c) \mod m, t = 1, 2, \ldots$$

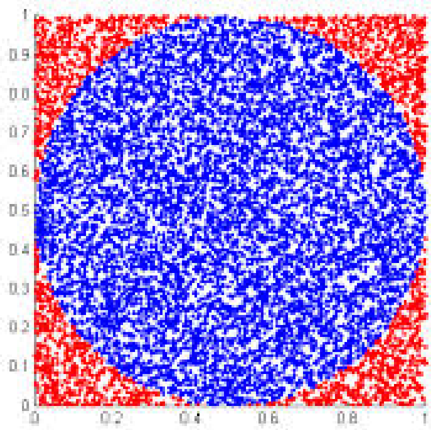
$$U_t = \frac{X_t}{m}$$

- 常数的选择:
$a = 7^5 = 16807, c = 0, m = 2^{31} - 1 = 2147483647$

Monte Carlo方法

- Monte Carlo methods follow a typical process:
  - Define a domain of possible inputs.
  - Generate inputs randomly from a probability distribution over the domain.
  - Perform a deterministic computation on the inputs.
  - Aggregate the results

圆周率计算



字符串匹配

- Given string text, search for occurrences of string pattern in it, pattern could be a regular expression, but assume just some other string
- Examples:
  - find "to be or not to be" string that's complete works of Shakespeare
  - find words on webpage/Word document
  - find DNA snippet in genome
- Different from dictionary problem – linked list/array/BST
  - string isn't discrete set of times 1 2 3
  - pattern appears over *several* chars in string

匹配算法

- Naive algorithm:
```
for i = 0 to n-1
  for j = 0 to len(pattern)-1
    look at t[i], p[j]
time: n*m
```
- Rabin-Karp algorithm:
```
for i = 0 to n-1-len(p)
  look at hash(t[i..i+len(p)-1]), hash(p)
    if match, look at strings
```

Rabin-Karp

- Seems better, but how to get hash(t[i..i+len(p)-1])s? naive way: walk through, copy out chars, send to has - time len(p) each time, total: n*m!
- Better way: choose hash so that can be computed incrementally. If sum of chars, then to go from one to next: subtract first, add next
- If sum of powers, make powers decreasing, subtract first, multiply total by prime ("left shift"), add next
- Either way, each transition is O(1)
- Total now: O(n)

Rabin-Karp

- Find "not to" in "to be or not to be"
  - Text: 116 111 32 98 101 32 111 114 32 110 111 116 32 116 111 32 98 101
  - Pattern: 110 111 116 32 116 111
  - Pattern sum: 596

```
116 111 32 98 101 32 = 490
111 32 98 101 32 111 = 490 - 116 + 111 = 485
32 98 101 32 111 114 = 485 - 111 + 114 = 488
98 101 32 111 114 32 = 488 - 32 + 32 = 488
101 32 111 114 32 110 = 488 - 98 + 110 = 500
32 111 114 32 110 111 = 500 - 101 + 111 = 510
111 114 32 110 111 116 = 510 - 32 + 116 = 594
114 32 110 111 116 32 = 594 - 111 + 32 = 515
32 110 111 116 32 116 = 515 - 114 + 116 = 517


110 111 116 32 116 111 = 515 - 32 + 111 = 596
```

小结

- Array access is $O(1)$
- Many possible keys $\rightarrow$ map to array position with hash function, store x in A[h(x)]
- Collisions are likely $\rightarrow$ must be dealt with using certain strategy
  - Separate chaining
  - Open addressing
- Non-integer keys must be mapped to integers

装填因子

- Load factor λ = ratio:
  - (element count) / (table size)
  - = average list length
- Expected total access time: 1 + $\lambda/2$
- For fixed element count, 1 + element-count/ 2*size = O(1)
- For chaining: keep λ <= 1
- For quadratic probing: keep λ <= 1/2, may fail if λ > 1/2

散列操作的复杂度

- Search/insert/delete
  - Average: O(1)
  - Worst: O(n)
- With good hash function, worst case is very unlikely
- In average case, get O(1) with very small constant

Roadmap

优先队列

- We would like to assign priorities to tasks in a queue to achieve a single objective: to minimize the total cost
- Think of you are standing in a line in the cafeteria, each one spends different time making orders
- Intuitively, priority corresponds to the inverse of the time to be spent

优先队列的操作

- Two typical operations: insert, deleteMin (the one has the highest priority)
- Straightforward implementation: use a linked list to keep data, findMin and delete it
- That would take O(n), longer time than necessary
- What about using a BST, findMin takes only log(n)

二叉堆

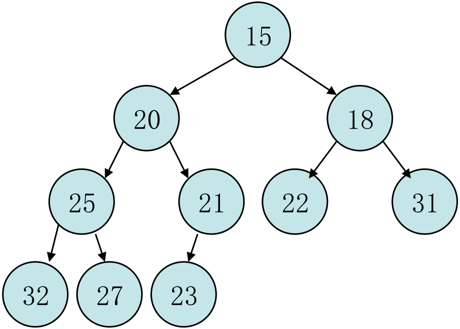
- We want $O(1)$ insertion and $O(\log n)$ deletion operations
- Two properties in mind: a complete binary tree, and heap order
- Complete binary tree: a full tree with the only exceptions on the bottom
- Heap order: the minimum element is always on the root
- All operations should not violate the above properties!

插入

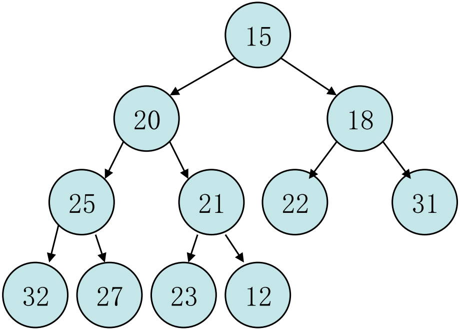
- To keep a tree complete:
  - Create a new node at the bottom
  - Set the new element to this node
- But heap order might have been broken
- Remedy: repeatedly exchange the inserted element with its parent if the latter is smaller
- A new element will be pushed to the top of a sub-tree in which it is the minimum

上滤

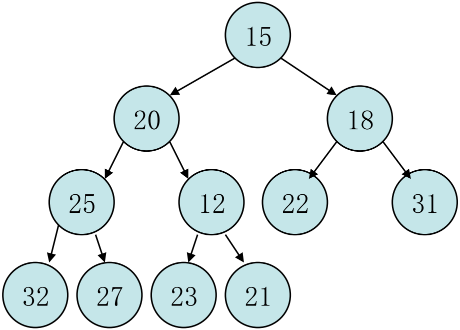
在堆中插入12



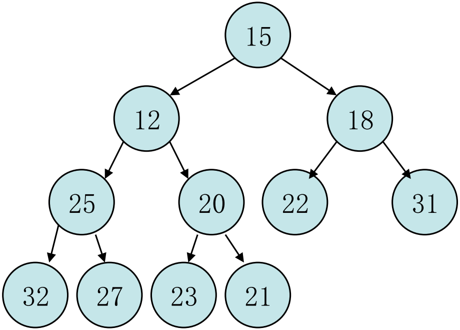
上滤



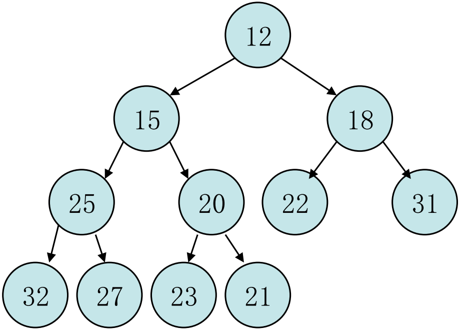
上滤



上滤



上滤

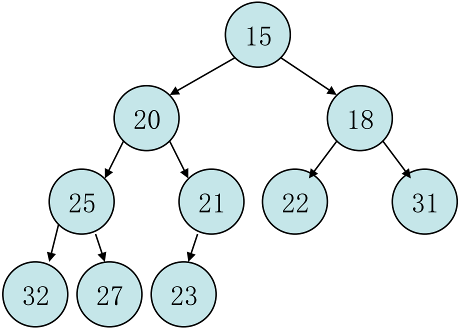


deleteMin

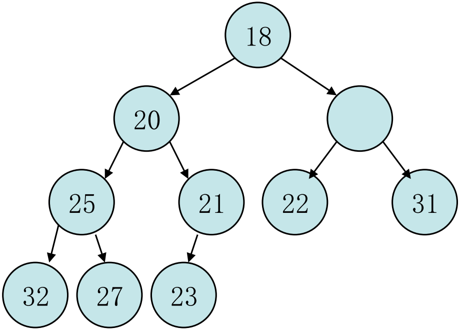
- Remember min is always on the root
- But deleting the root makes two sub-trees: another node has to fill in
- Which one? The smaller of the root’s two children
- Repeat the procedure until the bottom is reached
- Fill in the hole with the rightmost one on the bottom if needed, because the tree must be complete

下滤

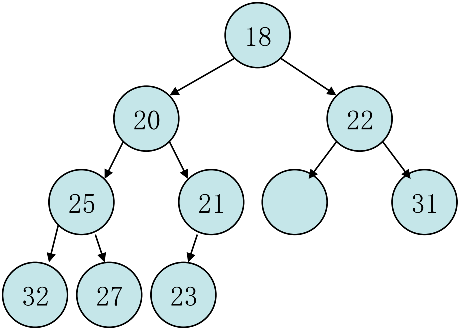
在堆中删除15



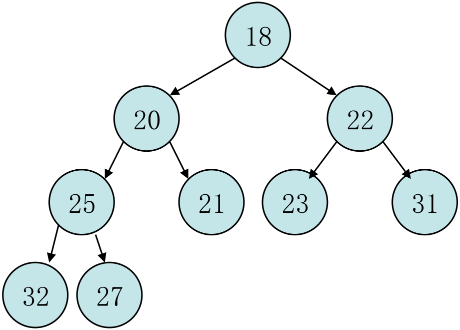
下滤



下滤



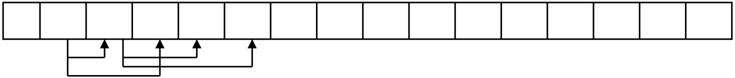
下滤



复杂度

- Worst cases:
  - Insertion, O(log n)
  - deleteMin, O(log n)
- Average cases:
  - Insertion, O(1), actually 2.607 comparisons, 1.607 moves
  - deleteMin, O(log n)
- Max heap can be similarly defined

数组实现

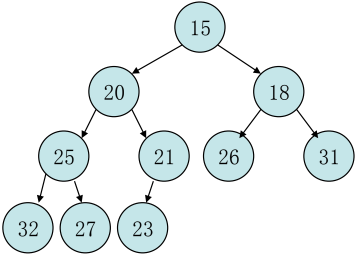


Parent to children links become default, determined by index in the array, left child 2i, right child 2i + 1, parent on i/2 , round down

```
typedef struct hEAP
{
    int capacity;
    int size;
    int *data;
} HEAP;
```

举例

	15	20	18	25	21	26	31	32	27	23					



Insert代码

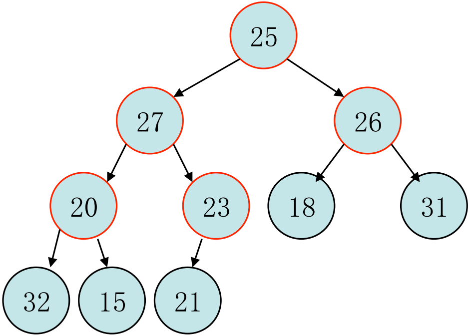
```
void insert (HEAP *h, int x)
{
    int i;

    if (h->size == h->capacity)     /* heap full */
        return;

    /* last element: h->elements[h->size - 1]
        parent of element i: i/2 */
    for (i = ++h->size; i > 1
         && h->elements[i/2] > x; i /= 2)
        h->elements[i] = h->elements[i/2];

    h->elements[i] = x;
}
```

建堆

- Where is the max element? On the bottom, may have as many leaves as a half of all nodes
- How to build a heap given N elements?
- Insert one element at a time, or
- Start from a complete binary tree with elements arbitrarily located
- Then adjust the of internal nodes' positions to meet heap property
- Follow a bottom-up fashion until root is adjusted

Build a Heap

	25	27	26	20	23	18	31	32	15	21



Build a Heap

	25	27	26	20	23	18	31	32	15	21

Build a Heap

	25	27	26	20	23	18	31	32	15	21

	25	27	26	20	21	18	31	32	15	23

Build a Heap

	25	27	26	20	23	18	31	32	15	21

	25	27	26	20	21	18	31	32	15	23

	25	27	26	15	21	18	31	32	20	23

Build a Heap

	25	27	26	20	23	18	31	32	15	21

	25	27	26	20	21	18	31	32	15	23

	25	27	26	15	21	18	31	32	20	23

	25	27	18	15	21	26	31	32	20	23

Build a Heap

	25	27	26	20	23	18	31	32	15	21
	25	27	26	20	21	18	31	32	15	23
	25	27	26	15	21	18	31	32	20	23
	25	27	18	15	21	26	31	32	20	23
	25	15	18	20	21	26	31	32	27	23

Build a Heap

	25	27	26	20	23	18	31	32	15	21

	25	27	26	20	21	18	31	32	15	23

	25	27	26	15	21	18	31	32	20	23

	25	27	18	15	21	26	31	32	20	23

	25	15	18	20	21	26	31	32	27	23

	15	20	18	25	21	26	31	32	27	23

Percolate Down代码

```
void percolate_down (HEAP *h, int i)
{
    int j, tmp, k;

    for (j = i, tmp = h->elements[j]; j * 2 < h->size; j = k)
    {
        /* find the smaller child */
        k = h->elements[2*j] < h->elements[2*j + 1] ? 2*j : 2*j + 1;
        /* if the root is bigger, move child one layer up */
        if (tmp > h->elements[k])
            h->elements[j] = h->elements[k];
        else
            break;
    }
    h->elements[j] = tmp;
}
```

建堆

- N: number of keys in a heap, implemented with an array
- Complexity: O(N), N inserts, remember average of an insert is O(1)
- Worst case: each insert needs # of moves = height of the inserted node, total # of moves is O(N)

```
for (i = h->size; i > 0; i--)
{
    percolate_down (i);
}
```

建堆的复杂度

Theorem

the complexity of buildHeap is O(n)

Proof idea: 2 comparisons for an internal node, 1 for finding a smaller child, 1 with that child. The number of moves in percolate down = height of the node. Then we need to know the sum of heights of all internal nodes in a heap. This is between the value for a full (perfect) binary tree with height h-1 and the value for a full binary tree with height h.

复杂度

$$S = \sum_{i=0}^{h-1} 2^i * (h - i) = h + 2(h - 1) + ... + 2^{h-1}$$

$$2S = 2h + 2 * 2(h - 1) + ... + 2 * 2^{h-1}$$

-S+2S， we have

$S = -h + 1 + 2 + ... + 2^h = (2^{h+1} - 1) - h = n - h$

堆排序

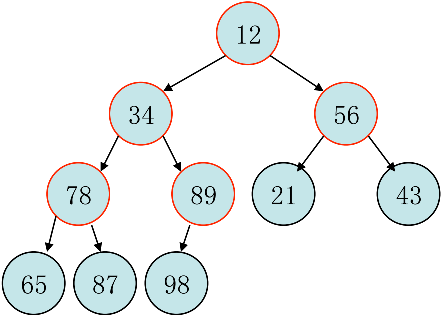
- Always delete/return the root of a heap, and copy the results to a new array
- The array is sorted – by deleteMin
- Waste of space, need an extra array
- Since deleteMin makes the heap shrunk, reuse its space, we have a decreasing order
- Max heap + deleteMax
- Complexity = build a heap + deleteMax n times, $O(n \log n)$

堆排序

- Given an array: 12, 34, 56, 78, 89, 21, 43, 65, 87, 98
- First build a max heap
- Then repeatedly deleteMax, and put the deleted element into the last cell of the heap

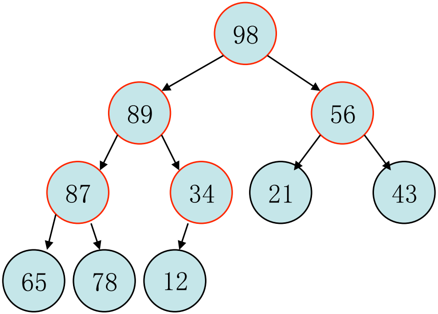
Original Array

	12	34	56	78	89	21	43	65	87	98



Max Heap

	98	89	56	87	34	21	43	65	78	12



	89	87	56	78	34	21	43	65	12	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12

Heap Sort

	98	89	56	87	34	21	43	65	78	12

	89	87	56	78	34	21	43	65	12	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12

	89	87	56	78	34	21	43	65	12	98

	87	78	56	65	34	21	43	12	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12

	89	87	56	78	34	21	43	65	12	98

	87	78	56	65	34	21	43	12	89	98

	78	65	56	12	34	21	43	87	89	98

	65	43	56	12	34	21	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98
	43	34	21	12	56	65	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98
	43	34	21	12	56	65	78	87	89	98
	34	12	21	43	56	65	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98
	43	34	21	12	56	65	78	87	89	98
	34	12	21	43	56	65	78	87	89	98
	21	12	34	43	56	65	78	87	89	98

Heap Sort

	98	89	56	87	34	21	43	65	78	12
	89	87	56	78	34	21	43	65	12	98
	87	78	56	65	34	21	43	12	89	98
	78	65	56	12	34	21	43	87	89	98
	65	43	56	12	34	21	78	87	89	98
	56	43	21	12	34	65	78	87	89	98
	43	34	21	12	56	65	78	87	89	98
	34	12	21	43	56	65	78	87	89	98
	21	12	34	43	56	65	78	87	89	98
	12	21	34	43	56	65	78	87	89	98

小结

- 二叉堆：优先队列，非线性结构
  - 结构：完全二叉树，可以用数组实现
  - 堆序：任意子树，根结点为最小（大）值
- 快速操作：insert $O(1)$，deleteMin(Max)，$O(\log n)$
- 建堆：对所有非树叶节点依次进行下滤，$O(n)$
- 堆排序：先构建最大堆，再进行多次deleteMax，替代最后一个树叶

实验7

- 1, 使用分离链路法处理冲突
  - Hash表的大小为2^k-1，初始可以为15
  - 表中的装填因子达到3/4时，增加表的大小至$2^{k+1}-1$，完成再哈希
  - 实现插入，删除和查找操作
  - 设计两个针对可变长度字符串的hash函数，并设计数据评价其性能
- 2, 书上5-7，多项式乘法的改进