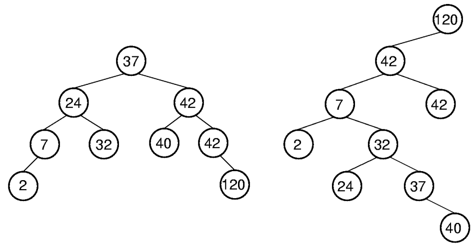
5.5 Binary Search Trees
二叉搜索树

54

54

Defination of Binary Search Trees

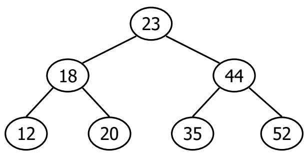
- ➢ All items in the left subtree < the root.
- ➢ All items in the right subtree >= the root.
- ➢ Each subtree is itself a binary search tree.



55

55

Binary Search Tree Traversals



preorder

23 18 12 20 44 35 52

postorder

12 20 18 35 52 44 23

inorder

12 18 20 23 35 44 52

56

56

Binary Search Tree

- 搜索二叉树所涉及的基本操作
  - Search
  - Insert
  - remove
  - Deletemin
  - Traversal---print
    - inorder
- 1个指针 +1个整型变量就可描述一棵搜索二叉树
  - 1个根指针root指向BST的根结点
  - 1个整型变量nodecount存放BST中的结点数

57

57

BST Class(1)

```
template <class E>
class BST {
private:
  BSTNode<E>* root;   // Root of the BST
  int nodecount;       // Number of nodes
  BSTNode<E>* findhelp( BSTNode<E>*, const E&) const;
  BSTNode<E>* inserthelp(BSTNode<E>*, const E&);
  BSTNode<E>* removehelp(BSTNode<E>*,
                     const E&,  BSTNode<E>*&);
  BSTNode<E>* deletemin(BSTNode<E>* , BSTNode<E>*&);
  void clearhelp(BSTNode<E>*);
  void printhelp(BSTNode<E>*, int) const;
```

58

58

BST Class (2)

```
public:
   BST() { root = NULL; nodecount = 0; }

  ~BST() { clearhelp(root); }

  void clear() { clearhelp(root); root = NULL;
                 nodecount = 0; }

  bool remove(E& e) {
    BSTNode<E>* t = NULL;
    root = removehelp(root, e, t);
    if (t == NULL) return false;
    nodecount--;
    delete t;
    return true; }
```

59

59

BST Class (3)

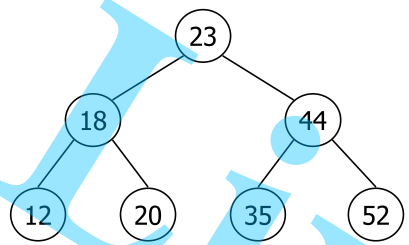
```
BSTNode<E>* find(const E& e) const
   { return findhelp(root, e); }

bool insert(const E& e) {
    root = inserthelp(root, e);
    nodecount++;
    return true; }

int size() { return nodecount; }

void print() const {      //相当于中序遍历
   if (root == NULL)
     cout << "The BST is empty.\n";
   else printhelp(root, 0);
}
};
```

60

60

BST Search/find

步骤
1. 与根相比
2. 等于返回
3. 小于往左子树寻找
4. 大于往右子树寻找



1. Search 20
2. Search 45

61

61

BST Class (4)-- findhelp

```
template <class E>  //返回指向找到结点的指针
BSTNode<E>* BST<E>:: findhelp( BSTNode<E>*
  subroot, const E& e) const {
  if (subroot == NULL) return NULL;
  if (e < subroot->element())
    return findhelp(subroot->left(), e);
  else if (e > subroot->element())
    return findhelp(subroot->right(), e);
  else { return subroot; }
}
```

O(d)

62

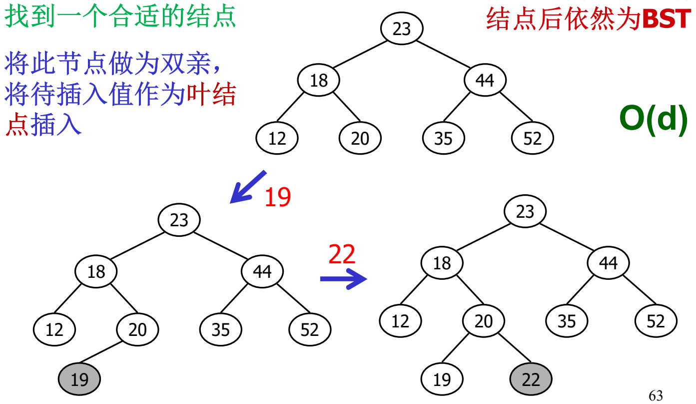
62

BST Insertion

步骤：

1. 找到一个合适的结点
2. 将此节点做为双亲，将待插入值作为叶结点插入

插入原则：插入新结点后依然为**BST**



O(d)

63

63

Iterative (non-Recursive非递归) BST Insertion Algorithm

```
Algorithm    insertBST (ref root <pointer>, val new <pointer>)
//       Pre    root is address of the root;  new is address of the new node
//       Post   new node inserted into the tree
    if (root = null)      root = new
    else
        pWalk = root
        loop (pWalk not null)       // Location found for the new node
            parent = pWalk
            if (new –> data < pWalk –> data)
                pWalk = pWalk –> left
            else
                pWalk = pWalk –> right
        if (new –> data < parent –> data)
            parent –> left = new
        else
            parent –> right = new
    return
End insertBST
```

Step1：找到一个合适的结点

Step2：将此节点做为双亲，将待插入值做为叶结点插入

64

64

BST Class (5-1)--BST Inserthelp (non-Recursive非递归)

```
template <class E>  BSTNode<E>*  BST<E>::
inserthelp( BSTNode<Elem>* subroot,  const E& val) {
  if (subroot == NULL) // Empty: create node
        return  new BSTNode<E>(val, NULL,NULL);
  BSTNode<E>  * temp = subroot;
  BSTNode<E>  *parent;
  while (temp!=NULL) {
        parent=temp;
        if (val < temp->element())   temp = temp->left();
        else   temp=temp->>right();  }
  temp=new BSTNode<Elem>(val, NULL,NULL);
   if (val < parent->element())     parent->setLeft(temp);
  else   parent->setRight(temp);
  return subroot;   // Return tree with node inserted
}
```

O(d)

Step1

Step2

65

65

Recursive(递归) BST Insertion Algorithm

```
Algorithm   insertBST (ref root <pointer>, val new<pointer> )
// Inserts a new node into BST using recursion
//      Pre    root is address of the root
//             new is address of the new node
//      Post   new node inserted into the tree
        if (root == null)
            root = new
      else
           if (new –> data < root –> data)
                insertBST (root –> left, new)
           else
                insertBST (root –> right, new)
      return
End   addBST
```

比较待插入值与根节点的大小

1）如果小于，将待插入值插入左子树

2）否则，将待插入值插入右子树

66

66

BST Class (5-2)-- BST Insert (Recursive 递归)

比较待插入值与根节点的大小

1）如果小于，将待插入值插入左子树

2）否则，将待插入值插入右子树

```
template <class E,>  BSTNode<E>* BST<E>::
 inserthelp (BSTNode<E>* subroot, const E& val) {
   if (subroot == NULL) // Empty: create node
         return new BSTNode<Elem>(val,NULL,NULL);
   if (val < subroot->element()))
         subroot->setLeft( inserthelp(subroot->left(), val));
   else
         subroot->setRight( inserthelp(subroot->right(), val));

   return subroot;  // Return subtree with node inserted
}
```

O(d)

67

67

BST Deletion

删除原则：删除结点后依然为BST

定义：从BST树中删除给定值

Step1：找到值等于待删除值的结点

Step2：根据该结点的特点采取不同的删除策略

- Leaf node: set the deleted node's parent link to null. Simplest
- Node having only left subtree: attach the left subtree to the deleted node's parent. simple
- Node having only right subtree: attach the right subtree to the deleted node's parent. simple
- Node having both subtrees   difficult

68

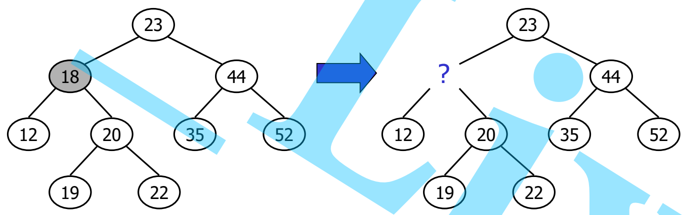
68

BST Deletion

---Node having both subtrees (1)

删除原则：删除结点后依然为BST

Example1：删除值为18的结点



从其子树中找一个值代替当前结点的值

69

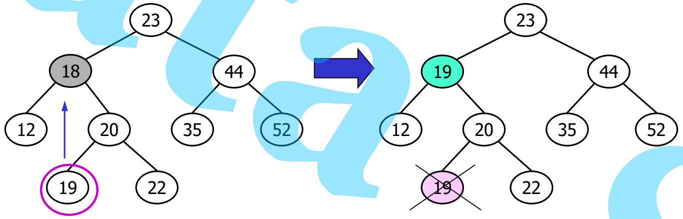
69

BST Deletion
---Node having both subtrees (2)

删除原则：删除结点后依然为BST

Using smallest value in the right subtree

Example1：删除值为**18**的结点



S1：找到右子树中最小值对应的结点
S2：用其值代替的当前结点的值
S3：删除该结点

叶子

70

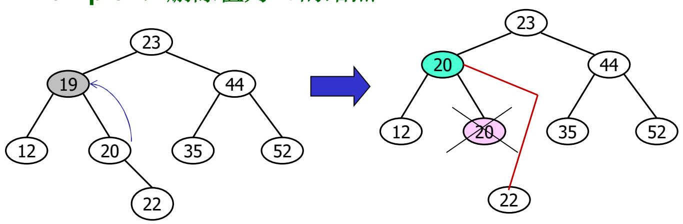
70

BST Deletion
---Node having both subtrees (3)

删除原则：删除结点后依然为BST

Using smallest node in the right subtree

Example2：删除值为**19**的结点



S1：找到右子树中最小值对应的结点
S2：用其值代替的当前结点的值
S3：删除该结点

只有右子树

71

71

BST Deletion
---Node having both subtrees (4)

删除原则：删除结点后依然为BST

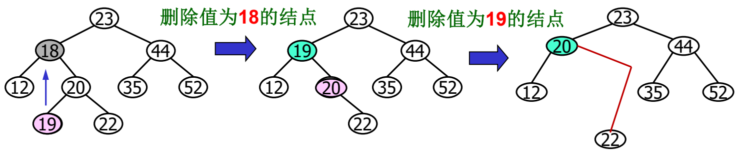
Using smallest node in the right subtree

① 找到右子树中的具有最小值的结点

✓ 要么为叶子结点，要么只有右子树

② 用该最小值替换待删结点的值

③ 删除具有最小值的结点（**simple**）



72

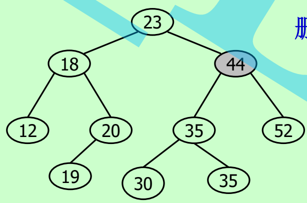
72

BST Delete

- Node having both subtrees
  - Using largest node in its left subtree

结合下图思考：*Why* 不用此策略？

删除值为**44**的结点



73

73

BST Deletion

---算法步骤

Leaf Node or Node having only right subtree

Node having only left subtree

Step1： 找到根结点值等于待删除值的子树，

Step2： 如果该子树无左树，用该子树的右子树代替该子树。

否则如果该子树无右树，用该子树的左子树代替该子树

否则

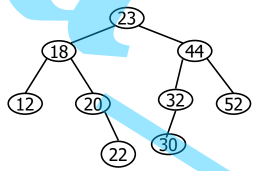
Node having both subtrees

① 找到右子树中的具有最小值的结点

✓ 要么为叶子结点，要么只有右子树

② 用该最小值替换待删结点的值

③ 删除具有最小值的结点



1 Delete 12

2 Delete 20

3 Delete 32

4 Delete 44

74

BST Class (6)-- Removehelp

```
template <class E> BSTNode<E>*  BST<E>::
removehelp(BSTNode<E>* subroot, const E& e, BSTNode<E>* & t)  {
    if (subroot == NULL) return NULL;
    if (e < subroot->element( ) )
          subroot->setLeft(removehelp(subroot->left(), e, t));
    else if (e > subroot->element( )  )
          subroot->setRight(removehelp(subroot->right(), e, t));
    else {   // Found it
          BSTNode<E>* temp;    t=subroot;
           if (subroot->left() == NULL)     //没有左树
                 subroot = subroot->right();
          else if (subroot->right() == NULL)   //没有右树
                 subroot = subroot->left();
          else {    // Both children are non-empty
                 subroot->setRight(deletemin(subroot->right(), temp));
                 subroot->setElement(temp->element));  }  }
    return subroot;
}
```

O(d)

75

75

BST Class (7)-- delete Minimum Value

带回两个指针

1. 指向已删除最小值的BST，返回值
2. 指向被删除的最小值结点，min

```
template <class E>
BSTNode<E>* BST<E>::
deletemin(BSTNode<E>* subroot,  BSTNode<E>* & min) {
  if (subroot->left() == NULL) {
    min = subroot;
    return subroot->right();
  }
  else {    // Continue left
     subroot->setLeft(deletemin(subroot->left(), min));
     return subroot;
  }
}
```

判断根节点是否有左子树

1）如果没有，根结点为最小值，返回其右子树

2）否则，从左子树中删除最小值（递归调用）

76

76

BST Class (8)-- Printhelp (Inorder traversal)

```
template <class E>   void BST<E>::
printhelp(BSTNode<E>* subroot, int level) const
{
        if (subroot==NULL)  return;
         printhelp(subroot->left(),level+1);
         for(int i=0; i<level; i++)
                 cout << "*";
         cout<<subroot->element()<<endl;
         printhelp(subroot->right(),level+1);
 }   //中序遍历
```

O(n)

77

77

BST Class (9)-- clearhelp ((postorder traversal)

```
template <class E>  void  BST<E>::
clearhelp(BSTNode<E>* subroot)
{
        if (subroot==NULL) return;
        clearhelp(subroot->left());
        clearhelp(subroot->right());
        delete subroot;
}
```

O(n)

后序遍历

78

78

Cost of BST operations

search:

Insert:

Delete:

O(d)

Worst case: O(n) 单边树

Best case: O(log(n)) CBT

希望**BST**尽可能左右平衡

79

79

An application example of BST

- 写一个程序，输入下列序列 构建**BST**，并测试插入，删除，查找, 打印等功能
- **37,24,42,7,2,40,42,32,120**
- **120,42,42,7,2,32,37,24,40**

输入序列顺序不同，构建的BST可能不同；
但是，中序遍历的结果却是绝对相同的。

80

80

searchBT.h

课件 **P25-26, 61-63,65, 68(0r 70), 78-81**

结点class

BST class

81

81

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "searchBT.h"
using namespace std;
void main() {
  BST<int> b1;
  int temp, i;
  cout<<" BST construction function test......"<<endl;
  cout<<"please input 9 int:";
  for(i=0;i<9;i++) {
        cin>>temp;   b1.insert(temp);   }
   cout<<" BST (inorder)"<<endl;    b1.print();
   cout<<" delete function test......"<<endl;
   cout<<"please input the data you want remove:";
   cin>>temp;
   b1.remove(temp);
   cout<<"after remove "<<temp<<"BST(inorder) is "<<endl;
   b1.print();
}
```

82

82

5.6 Heaps(堆)

Also called Priority Queue
（优先队列）

83

83

Priority Queue/优先队列

优先级高的先出队

When a collection of objects is organized by importance or priority, we call this a **Priority Queue**

基本操作：
Insert（Enqueue），插入一个新任务后依然需保持优先队列的特点
removeFirst（Dequeue)，　完成（删除）优先级最高任务后依然需保持优先队列的特点

实现：
一些简单的实现： list，BST
Heap（堆）： 普遍应用，和优先队列几乎被认为是同一个概念

84

84

Defination of Heaps

①
Complete binary tree whose node with property:

1) value of any node **less than or equal** to that of its children (Min-heap, 小堆) or ②

2) value of any node **larger than or equal to** that of its children (Max-heap 大堆)

逻辑定义

85

85

Defination of Heap

因为堆是CBT，所以堆通常用基于数组的方式来实现，即将BT中的结点按层由低到高，层内由左到右进行编号并存放于1维数组中，其结点下标满足下列关系式：

- PARENT(i) = (i-1)/2; /*父结点*/
- LEFT(i) = 2i+1; /* 左子结点 */
- RIGHT(i) = 2i+2; /* 右子结点 */
- n_0 =(int) ((n+1)/2) ; /* 叶子结点的个数 */

物理定义

Defination：n个元素组成的序列$\{k_0,k_1,k_2,\ldots,k_{n-1}\}$,当且仅当满足下列关系之一时，称之为堆

1） $k_i \le k_{2i+1}$,且 $k_i \le k_{2i+2}$, i =0,1,…, n/2-1 小堆

2) $k_i \ge k_{2i+1}$,且 $k_i \ge k_{2i+2}$, i =0,1,…, n/2-1 大堆

86

Heap 与BST的区别

- BST ：
  - 左与右的关系
  - 不一定是CBT
  - 一般用基于指针的方式存储/实现
- heap:
  - 前辈与后辈的关系
  - 一定是CBT
  - 一般用基于数组的方式存储/实现

87

87

Heap

- 1个数组 +2个整型变量就可描述一个堆
  - 1个数组存放heap中各结点的值
  - 1个整型变量maxSize存放数组的尺寸
  - 1个整型变量size存放 堆中的结点数
- heap所涉及的基本操作
  - Insert
  - remove
  - removeFirst
  - buildHeap

88

88

maxHeap class(1)---Array based implement

```
template<class Elem> class maxHeap {
private:
  Elem* Heap;   // Pointer to the heap array
  int maxSize;    // Maximum size of the heap
  int size;       // Number of elems now in heap
  void siftDown(int); // Put element in place
public:
  maxHeap(Elem* h, int num, int max) {
    size=num;   maxSize=max;   Heap = new Elem[max];  }
  int heapSize() const  { return size  ; }
  bool isLeaf(int pos) const {
      return (pos >= size/2) && (pos < size);      }
  int leftChild(int pos) const  { return  2*pos+1; }
  int rightChild(int pos) const { return  2*pos+2;  }
  int parent(int pos) const   { return  (pos-1)/2; }
  void print( ) const   { …   }
  void clear( )   { …  }
  int find (const Elem&) { … }
```

89

89

maxHeap class(2)---Array based implement

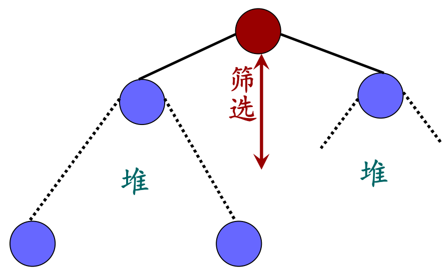
```
void buildHeap();
void insert(const Elem&);
Elem removeFirst( );
Elem remove(int);
};
```

90

90

Siftdown /筛选

所谓“siftdown”指的是，对一棵左/右子树均为堆的完全二叉树，“调整”根结点使整个二叉树也成为一个堆。



91

91

SiftDown

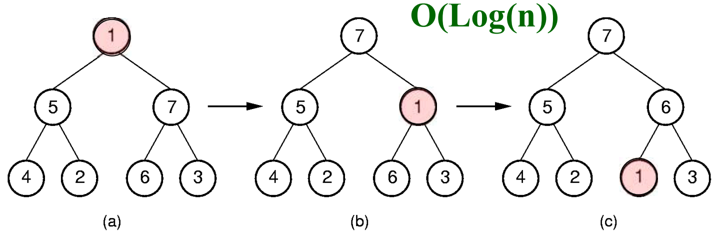
Siftdown/筛选的方法

S1： 将根结点作为当前结点

S2： 将当前结点值与其左、右子树的根结点值比较，并与三者中最大者进行交换；更新当前结点

S3：重复S2，直至叶子结点或无交换发生，所得结果即为堆。

O(Log(n))



92

maxHeap class(3)--- SiftDown

```
template <class Elem>
void maxHeap<Elem>::siftDown(int pos) {
  while (!isLeaf(pos)) {
    int j = leftChild(pos);
    int rc = rightChild(pos);
    if ((rc<size) && (Heap[j]< Heap[rc]))
      j = rc;
    if ( Heap[pos] >= Heap[j] ) return;
    swap(Heap, pos, j); // 请自行写出该函数的代码
    pos = j;
  }
}
```

O(Log(n))

93

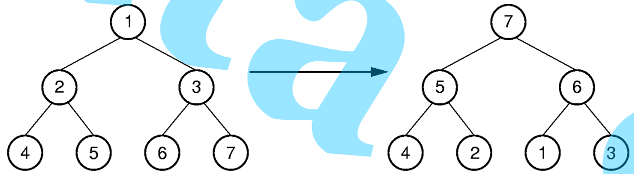
93

Building the MaxHeap

For fast heap construction:

•**Call siftdown for each item from high end(尾端) of array to low end（前端）** 从下往上/从后往前

•**Don't need to call siftDown on leaf nodes.(why?)**

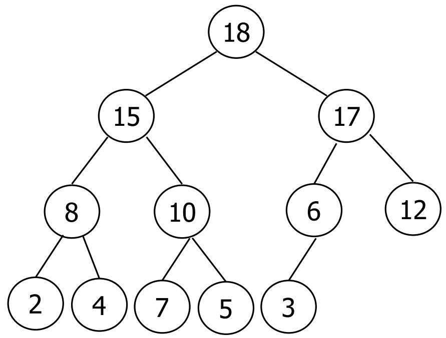


(7-3), (5-2), (7-1), (6-1) **共交换4次**

94

下列数组/**CBT**是大堆吗？若不是，请构建(要求写出具体过程)

5, 10, 12, 8, 15, 6, 17 , 2, 4, 7, 18, 3



95

maxHeap class(4)--- BuildingHeap

```
template <class Elem>
void maxHeap<Elem>:: buildHeap( ) {
   for(i = size/2-1; i >= 0; i--)
        siftDown(i);
}
```

O(n)

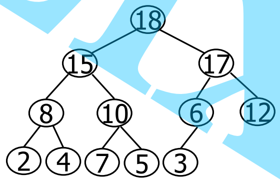
f(n)的具体计算公式见课本p184

96

Insert a value in the MaxHeap

思路：

- 在堆末尾添加一取值为待插入值的叶子结点，作为当前结点，并size加1
- 将当前结点值与其双亲结点值比较，若大于则进行交换，并将其双亲作为当前节点；
- 重复上述操作，直至当前结点值小于等于其双亲结点值 或到达根结点。



97

maxHeap class(5)--- Insert

```
template <class Elem>
void maxHeap<Elem>:: insert(const Elem& e)
{
    Assert( size < maxSize, "Heap is full");
    int curr = size;
    Heap[curr] = e;  size++;
    while(curr!=0 && Heap[curr]>Heap[parent(curr)])  {
          swap(Heap, curr,parent(curr)); curr=parent(curr);
    }
    return true;
}
```

O(Log(n))

98

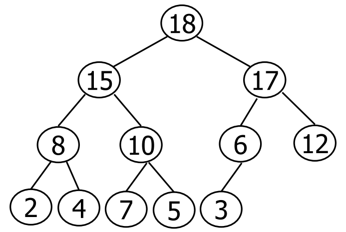
98

Remove First value in the Maxheap

思路：

作用？

- 将根结点值与最末叶子结点值进行交换，并size减1
- 对根结点 做 siftDown 操作



99

99

maxHeap class (6)--Remove First Value

O(Log(n))

```
template <class Elem>
Elem maxHeap<Elem>::
removeFirst() {
  Assert ( size > 0, "Heap is empty");
  swap(Heap, 0, --size);   // Swap First with end
  if (size != 0) siftDown(0);
  return Heap[size];  // Return First value
}
```

100

100

Remove 给定下标位置的值从maxHeap

思路：

作用？

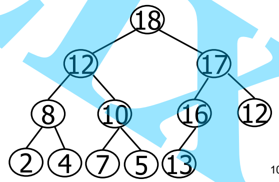
①将待删除结点作为当前结点

②将当前结点与最末叶子结点进行值交换，并size减1。

③将当前结点值与其双亲结点值比较，若大于则进行交换，同时将其双亲作为当前节点；

④重复步骤3，直至当前结点值小于其双亲结点值 或 到达根结点。

⑤对当前结点调用 siftDown



101

101

maxHeap class(7) --Remove

```
template <class Elem>
Elem maxHeap<Elem>::remove(int pos) {
  Assert((pos>=0)&&(pos<size),"Bad position");
  if ( pos == size-1 ) size--;
  else {
    swap(Heap, pos, --size);
    while ((pos != 0) &&
           (Heap[pos] > Heap[parent(pos)])){
      swap(Heap, pos, parent(pos));
      pos = parent(pos);
    }
    if (size != 0) siftDown(pos);   }
  return Heap[size];
}
```

O(Log(n))

102

102

An application example of heap

- 写一个程序，输入下列序列构建**maxHeap**，并测试插入，删除，查找，清空，打印等功能
- 1 2 3 4 5 6 7 8 9
- 5 2 9 3 7 6 8 4 1

输入序列顺序不同，构建的heap可能不同；但是，重复removeFirst直到堆为空得到的结果却是绝对相同的。

103

103

```
……
#include "heap.h"   // maxHead class--课件中PP91,92,95,97,101,103
using namespace std;
void main() {
    int i;   double a[100], temp;
    cout<<"please input 7 data:"<<endl;
    for(i=0;i<7;i++)   cin>>a[i];
    maxHeap<double>  h1(a,7,100);
    cout<<"after buildHeap the heap is:"<<endl;
    h1.buildHeap();   h1.print();     cout<<endl;
    cout<<"insert function test........"<<endl;
    cout<<"please input the insert data:";
    cin>>temp;    h1.insert(temp);
    cout<<"after insert "<<temp<<" the heap is:"<<endl;
    h1.print();  cout<<endl;
    cout<<"removeFirst function test........"<<endl;
    while(h1.heapSize()) {
      temp=h1.removeFirst( );  cout<<temp<<"  ";    }
    cout<<endl;
    ......
}
```

104

104

本章作业二

- 5.9
- 5.13 (b)
- 5.16

105