COMP2012H
Spring 2014
Dekai Wu

Sorting Algorithms

- Insertion sort
- Mergesort
- Quicksort
  - Selection
- Heap and heapsort

Insertion Sort

The problem is to sort an unordered array `A[0..n-1]` in non-decreasing order.

Given `A[0..n-1]`,

- Sorting a handful of cards
- Move an integer p as a "pointer" from 1 to $n-1$.
- Maintain the invariant that the prefix `A[0..p]` becomes sorted before p is advanced.
- Let x be the content of `A[p]`. The invariant is maintained by inserting x at the proper location, say `A[j]`, among `A[0..p-1]`. `A[j]` is vacated for x by shifting the entries `A[j..p-1]` one position to the right.

```
void insertionSort(vector<int> & A)
{
   int j;

   for (int p = 1; p < A.size(); p++) {
        int tmp = A[p];
        for (j = p; j > 0 && tmp < A[j-1]; j--)
             A[j] = A[j-1];
        A[j] = tmp;
   }
}
```

Insertion Sort: An Example

5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

What is the best case input and running time? (Sorted sequence, n)
What is the worst case input and running time? (Reverse sequence, n^2)

This is a n^2 algorithms, as the worst-case running time trends like n^2 with the input size.

Divide and Conquer: Mergesort

We are to sort a sequence of n numbers. If $n = 1$, then we are done. Otherwise,

1. Divide the sequence into two subsequences, each having at most $\lceil n/2 \rceil$ numbers.
2. *Sort each subsequence recursively.*
3. *Merge* the two sorted subsequences to produce one sorted sequence.

- How to divide the sequence into two subsequences? How to divide quickly?
- What does it mean to sort recursively?
- How to merge? How quick is it?

Mergesort Operation

- If the input sequence is given as a linked list, then we can scan the linked list from left to right. We stop at the $\lfloor n/2 \rfloor$th entry and cut the link to obtain two sublists. Overall, this approach is of the same time complexity as the array case.
- If the input sequence is given as an array `A[0..n-1]`, then we can avoid the linear-time scan. We can represent any subsequence of `A[0..n-1]` by two integers `left` and `right` which index the two entries delimiting the subsequence. For example, to divide `A[left .. right]`, we compute
  `center = (left + right)/2`
  and the two subsequences are `A[left .. center]` and `A[center+1 .. right]`. This takes $O(1)$ time.
- Recall that we are designing an algorithm `mergesort` that sorts n integers. And n can be any positive number. So *sorting each subsequence recursively* means that we invoke `mergesort` two times, once to sort `A[left .. center]` and once to sort `A[center+1 .. right]`.

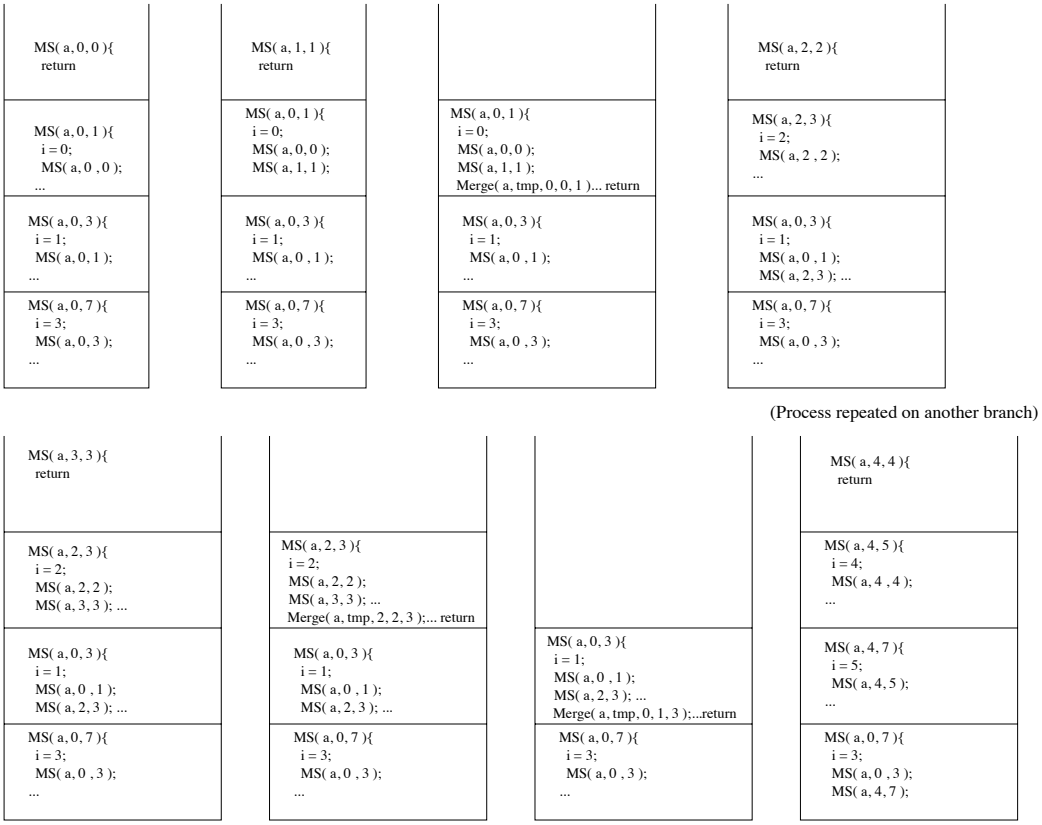
Mergesort Code (Recursive)

```
template <class T>
MergeSort( T a[], int left, int right)
{// Sort the elements in a[left:right].
   if (left < right) { // at least 2 elements
      int i = (left + right)/2; // midpoint
      MergeSort(a,left,i);
      MergeSort(a,i+1,right);
      // merge from a into an array b
      Merge(a,b,left,i,right);
      // put result back into a from b
      Copy(b,a,left,right);
      }
}
```

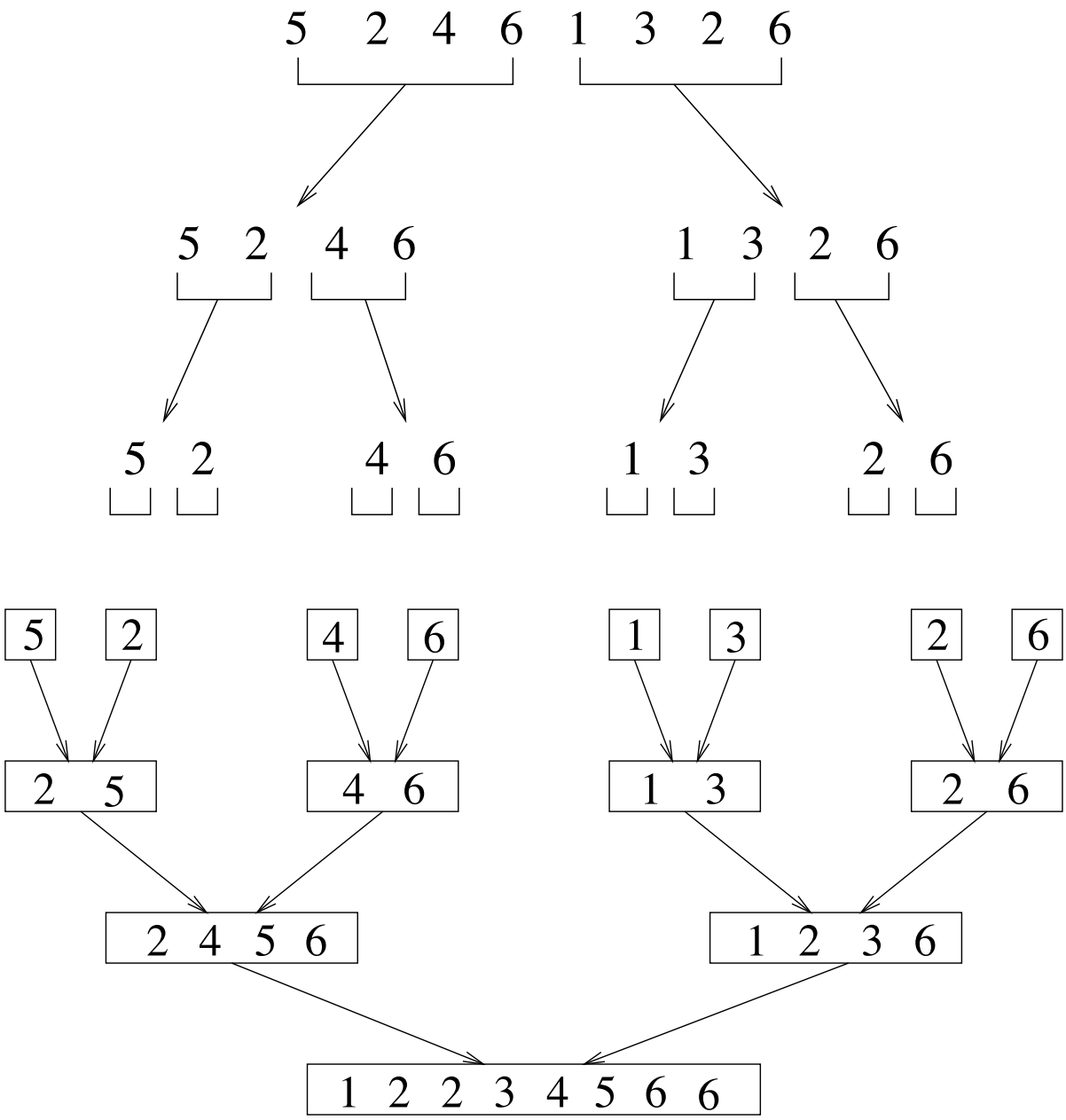
Assume for now that `Merge()` works correctly.

Calling `MergeSort(A,0,n-1)` will sort the array `A[0..n-1]`.

An Example on Calling Mergesort(a, 0, 7)



An Example


5 2 4 6 1 3 2 6
5 2 4 6
1 3 2 6
5 2
4 6
1 3
2 6
5
2
4
6
1
3
2
6
2 5
4 6
1 3
2 6
2 4 5 6
1 2 3 6
1 2 2 3 4 5 6 6

Describing `merge()` Using Pseudo-Code

Algorithm $merge(A, p, q, r)$
Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.
Output: $A[p..r]$ is sorted.
($* T$ is a temporary array. $*$)

1. $k = p$; $i = 0$; $l = q - 1$;
2. **while** $p \leq l$ and $q \leq r$ //merging steps
3. **do if** $A[p] \leq A[q]$
4. **then** $T[i] = A[p]$; $i = i + 1$; $p = p + 1$;
5. **else** $T[i] = A[q]$; $i = i + 1$; $q = q + 1$;
6. **while** $p \leq l$
7. **do** $T[i] = A[p]$; $i = i + 1$; $p = p + 1$;
8. **while** $q \leq r$
9. **do** $T[i] = A[q]$; $i = i + 1$; $q = q + 1$;
10. **for** $i = k$ to r
11. **do** $A[i] = T[i - k]$; //copyback

Note that

```
merge(a, left, i+1, right)
```

is equivalent to

```
Merge(a, b, left, i, right);
Copy(b, a, left, right);
```

on page 6.

Merging Steps and Quicksort

Please refer to ppt slides supplement.

Comparison Between Insertion Sort, Mergesort and Quicksort

n	ISORT (secs)	MSORT (secs)	Ratio
100	0.01	0.01	1
1000	0.18	0.01	18
2000	0.76	0.04	19
3000	1.67	0.05	33.4
4000	2.90	0.07	41
5000	4.66	0.09	52
6000	6.75	0.10	67.5
7000	9.39	0.14	67
8000	11.93	0.14	85

n	MSORT	QSORT
10000	0.18	0.07
20000	0.37	0.17
30000	0.62	0.25
40000	0.83	0.33
50000	1.00	0.45
60000	1.29	0.53
70000	1.50	0.59
80000	1.78	0.75

Time Comparisons

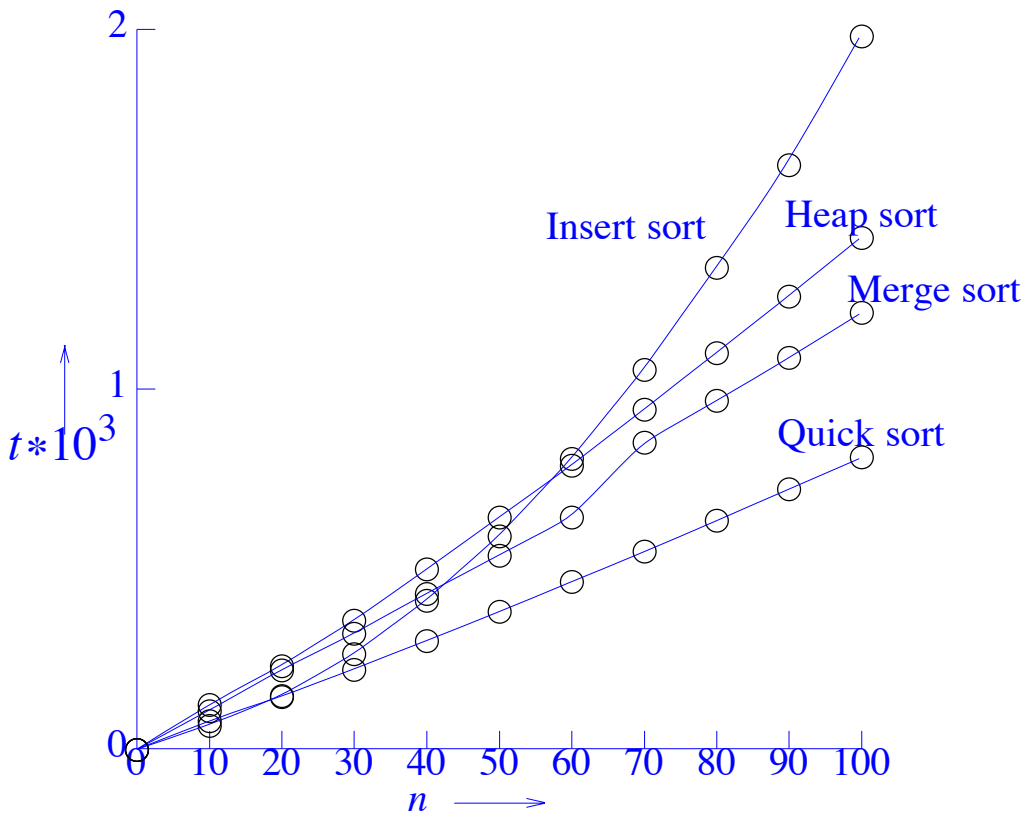


Figure 14.12 Plot of average times

- Quick sort outperforms the other sort methods for suitably large n
- There is a breakeven point (around 20) below which insertion sort is effective
- Use insertion sort when the number of elements is fewer than the breakeven point; otherwise use quicksort.
- In the case of quicksort, the recursion step can be replaced by insertion sort if the number of elements is lower than the breakeven point.

Selection

- Given an array `a[0:n-1]` of n elements, find the kth element.
  - For example, find the median
  - Sort it and return the kth element. This takes $O(n \log n)$ time, which is not necessary and fast enough
- Use a simpler quicksort to recursively focus only on the segment with the kth element
- Always choosing the left element as the pivot leads to poor performance if the array is sorted
  - For simplicity, we have used this in the `select` program

select

```
template<class T>
T select(T a[], int l, int r, int k)
{// Return k'th smallest in a[l:r].
   if (l >= r) return a[l];
   int i = l,      // left to right cursor
       j = r + 1;  // right to left cursor
   T pivot = a[l];

   // swap elements >= pivot on left side
   // with elements <= pivot on right side
   while (true) {
      do {// find >= element on left side
         i = i + 1;
         } while (a[i] < pivot);
      do {// find <= element on right side
         j = j - 1;
         } while (a[j] > pivot);
      if (i >= j) break;  // swap pair not found
      Swap(a[i], a[j]);
      }

   if (j - l + 1 == k) return pivot;

   // place pivot
   a[l] = a[j];
   a[j] = pivot;

   // recursive call on one segment
   if (j - l + 1 < k)
      return select(a, j+1, r, k-j+l-1);
   else return select(a, l, j-1, k);
}
```

Heapsort

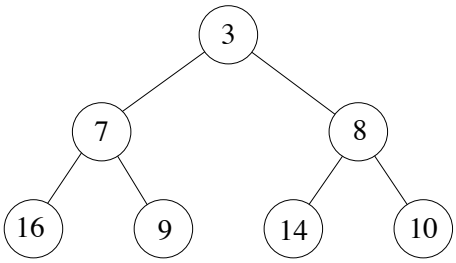
1. Select the current minimum element from the remaining numbers.
2. Output the current minimum and remove it.
3. Repeat the above two steps.

Introduce a new data structure called *heap* that can support the two operations efficiently in $O(\log n)$ time. Specifically, we can maintain a set of n elements in a heap such that

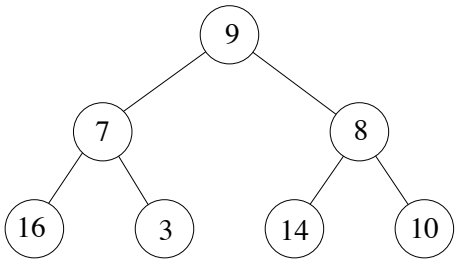
- can locate the current minimum in $O(1)$ time.
- can delete the current minimum in $O(\log n)$ time.

Heap

- A typical representation of a heap is a *complete* binary tree.
- Each node stores a value.
- The value stored at a node is smaller than or equal to the values stored at its children. We call this the *min-heap property* and the corresponding heap a *min-heap*. Symmetrically, one can also define the *max-heap property* and a *max-heap*.



min-heap



not a min-heap

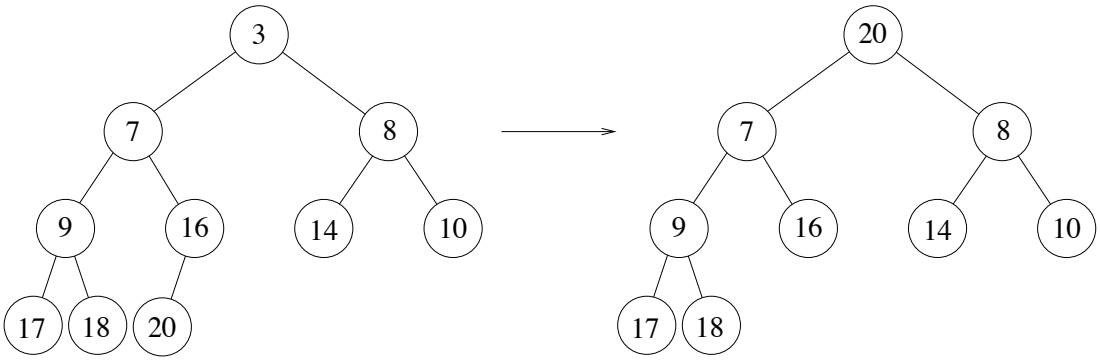
Definitions

- A max tree (min tree) is a tree in which the value in each node is greater (less) than or equal to those in its children (if any)
- A max heap (min heap) is a max (min) tree that is also a *complete* binary tree

Heap Root and the Minimum Number

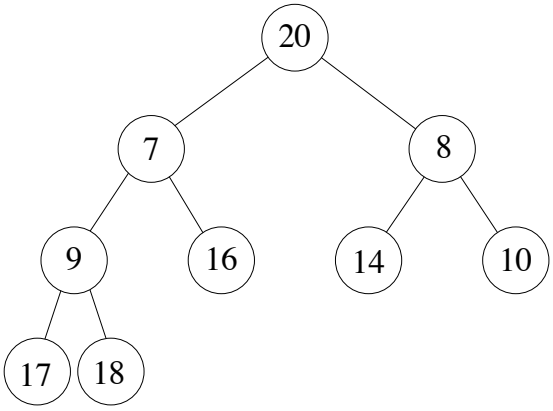
- The root of the tree contains the minimum number.
  - Hence, the minimum number can be accessed in $O(1)$ time.
- Deleting the minimum number
  1. Copy the last number to the root, thus overwriting the number stored at the root.
  2. Restore the min-heap property.

Heap Operation: Deletion

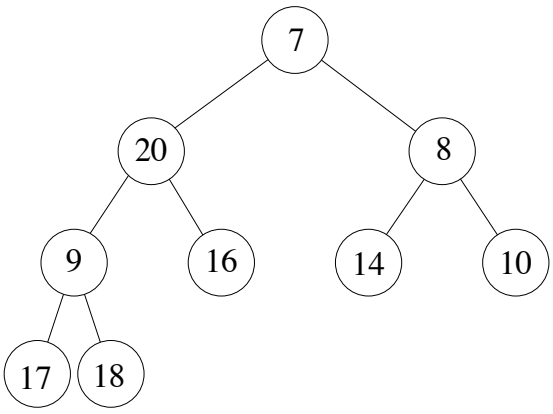


mn-heap property destroyed

Restore the min-heap property

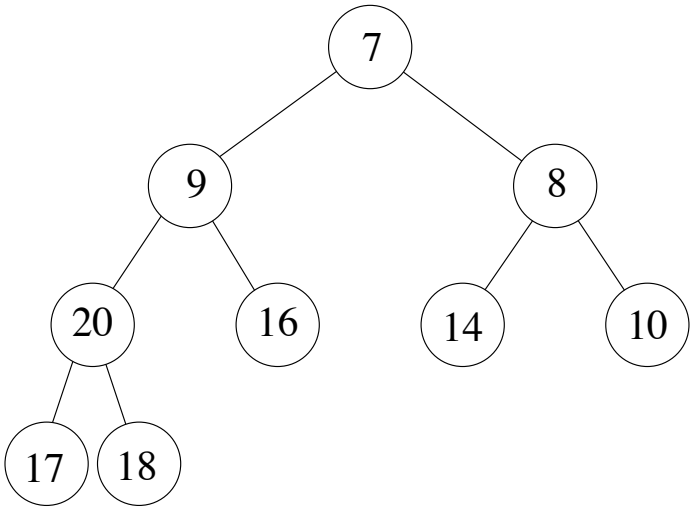


compare 20 with 7, the smallest of its two children

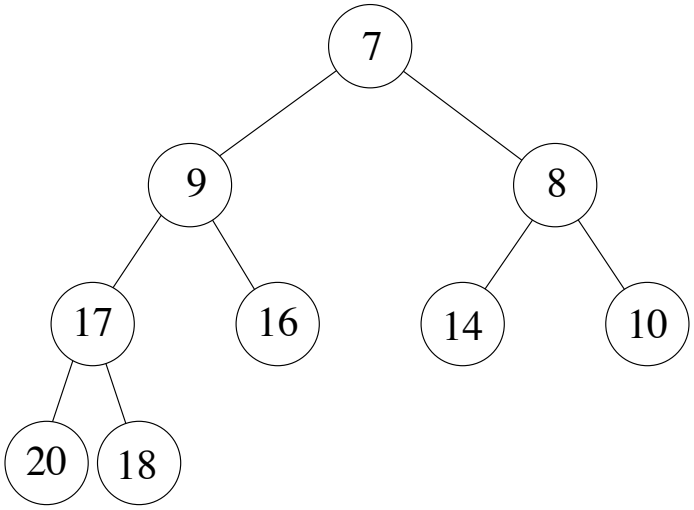


compare 20 with 9, the smallest of its two children

Deletion (Cont.)



compare 20 with 17, the smallest of its two children



A heap!
Time complexity
= O(height) = O(log n)

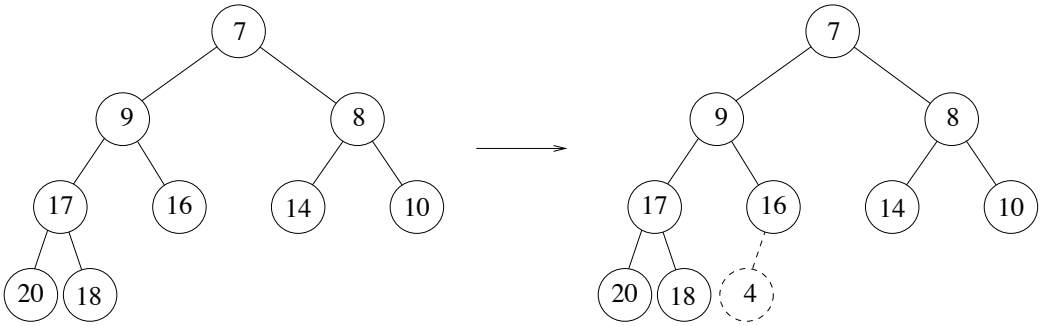
So we know how to delete the minimum number.

But how to build a heap in the first place?

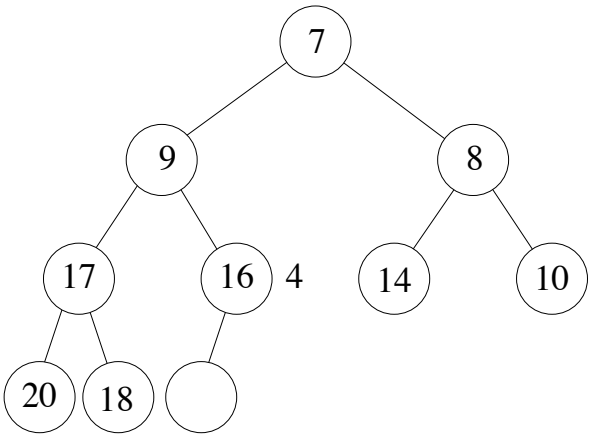
Building a Heap

We can do it by incremental insertion. So how to insert a new number into a heap?

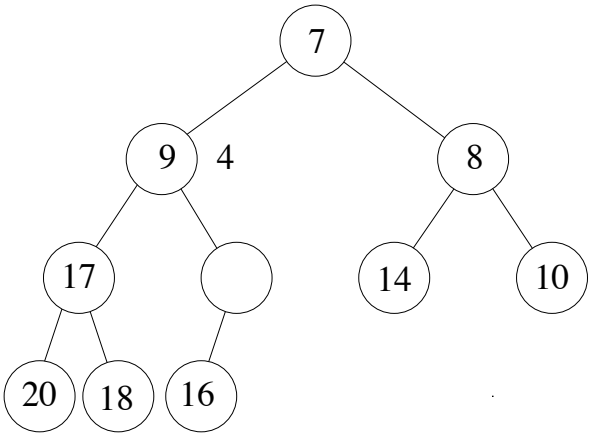
1. Add the number to the bottommost level.
2. min-heap property may be violated.



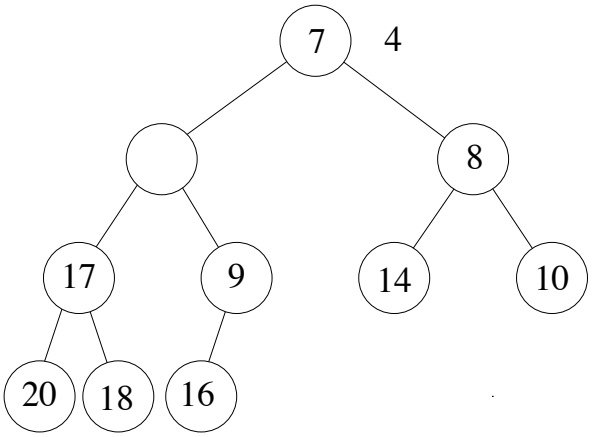
Restoring the Min-Heap Property



compare 4 with 16



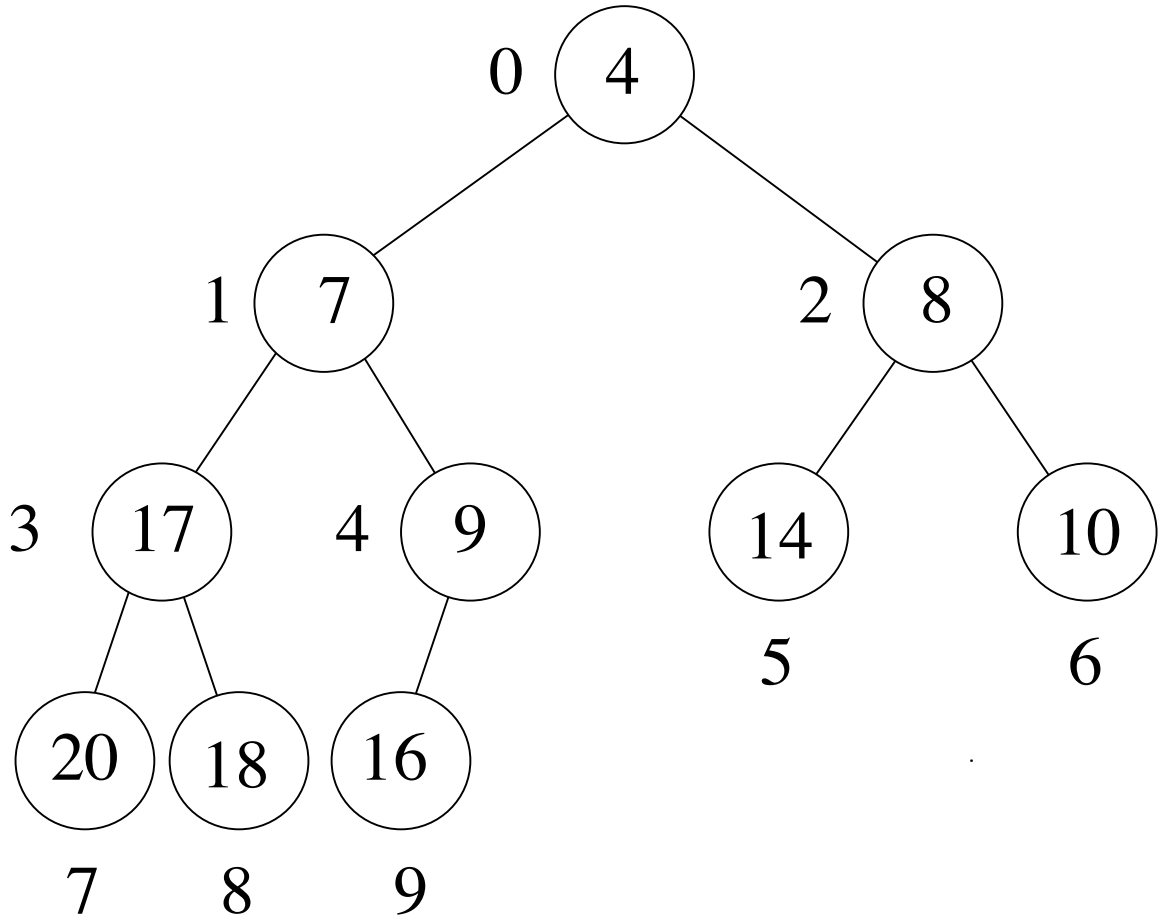
compare 4 with 9



compare 4 with 7

Array Representation of the Complete Binary Tree

- store the min-heap in an array.
- map the nodes in the complete binary tree to elements of the vector by numbering the nodes from level to level.



Array Values

Store the values in this order in an array A

4	7	8	17	9	14	10	20	18	16

- given a node at location i, its children are at locations $2i + 1$ and $2i + 2$.
- given a node at location $i > 0$, its parent is at location $\lfloor (i - 1)/2 \rfloor$.
- if n is the current number of values being stored in the heap, $A[n-1]$ stores the rightmost node at the bottommost level and $A[n]$ is the next available location.

In-Place Sorting

- If min-heap is used, a temporary buffer of size $O(n)$ is needed to hold the partially formed min-heap
  - Adding one element at a time at the end of a partial heap, followed by heap restoration
  - Deleting element by element back to the orignal array
- This is a high storage overhead
- We would like to sort *in-place* so that no extra storage is incurred
- Do it with max-heap instead of min-heap
- One way is to expand from the first element by adding one element at a time and restoring a max-heap each time.
- Aonther way is to work "backwards" from the middle element to build max-heap along the way. We will discuss it in the following.

Max Heap Initialization from an Array

Converting an array [20, 12, 35, 15, 10, 80, 30, 17, 2, 1] into a max heap:

1. Begin with the first element that has a child (i.e., 10). This element is at poistion $i = \lfloor n/2 \rfloor$.
2. If the subtree that rooted at this position is a max heap, then no work is done here
3. Otherwise, we adjust the subtree so that it is a heap
4. Following this adjustment, we examine the subtree whose root is at $i - 1$, then $i - 2$, and so on until we have examined the root of the entire binary tree, which is at position 1.
5. Note that one may build the max-heap by inserting numbers one by one instead of working backwards as above.

Max Heap Initialization: An Example

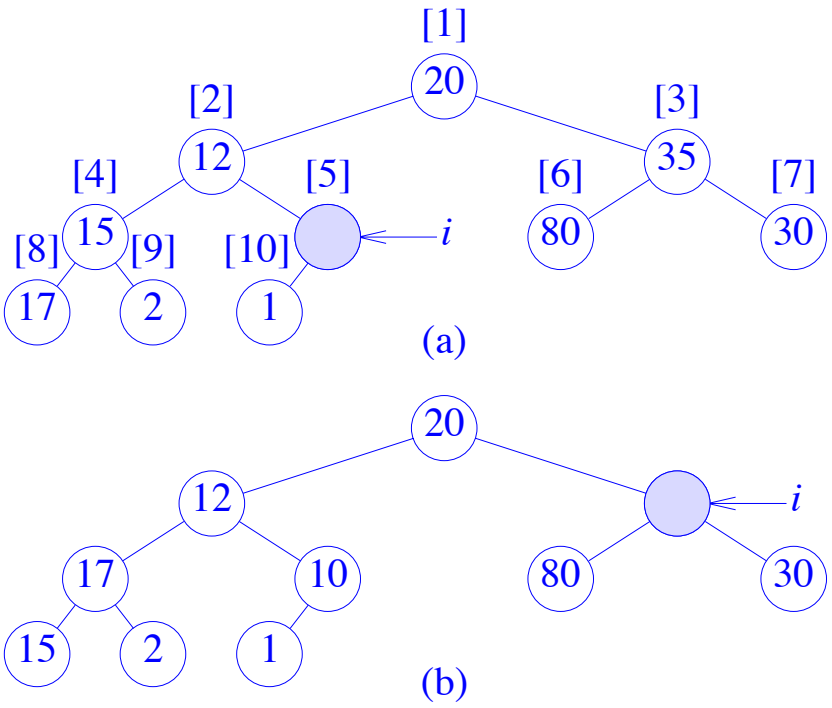


Figure 9.5 Initializing a max heap

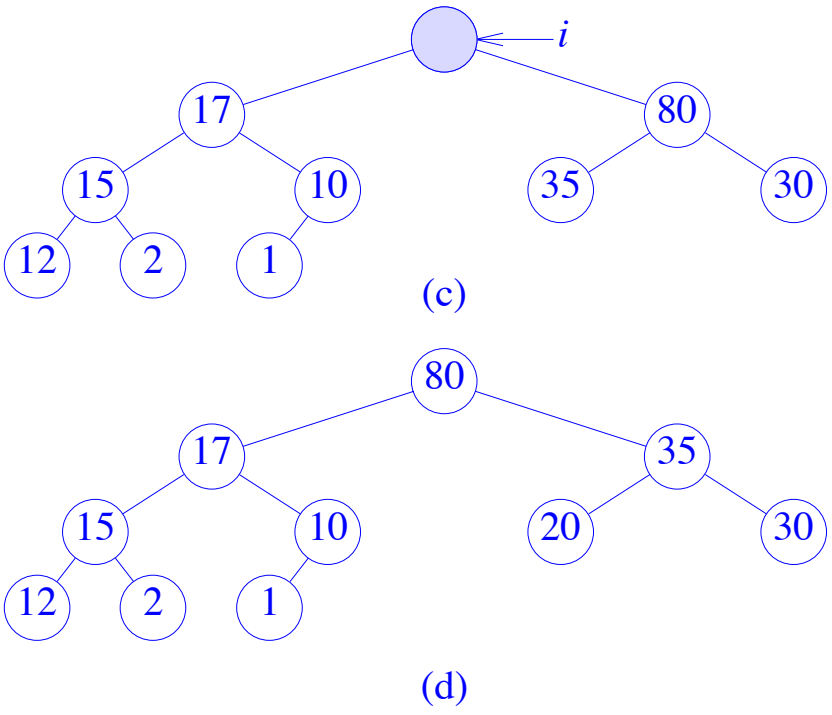
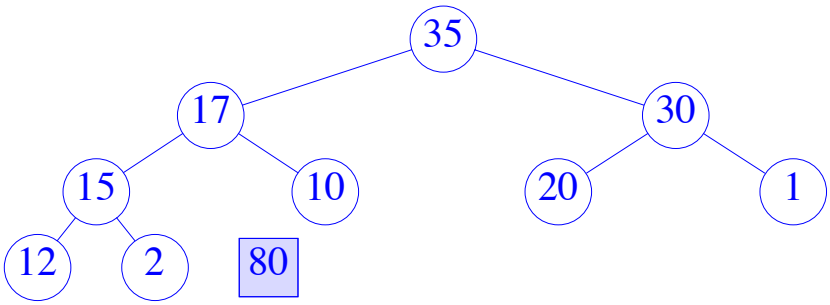
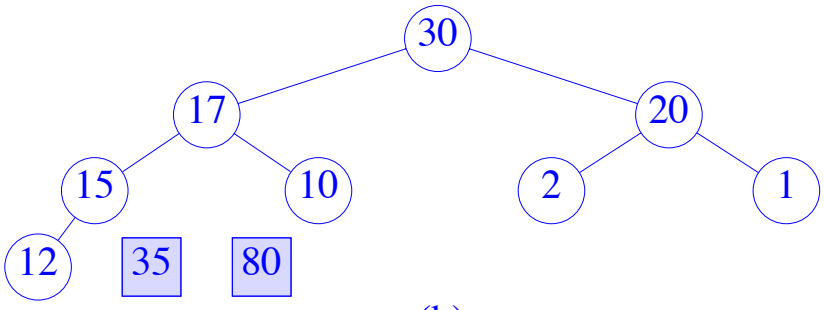


Figure 9.5 Initializing a max heap (Continuation)

In-Place HeapSort

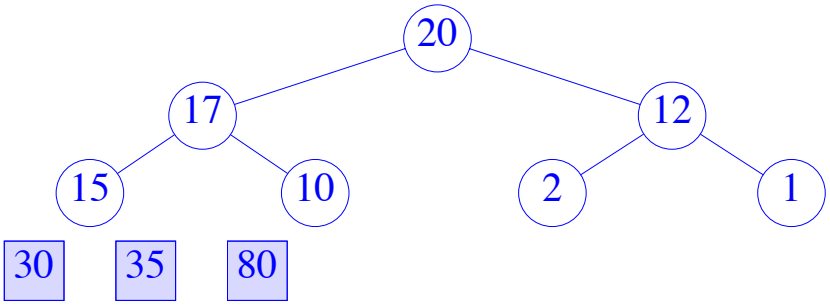


(a)

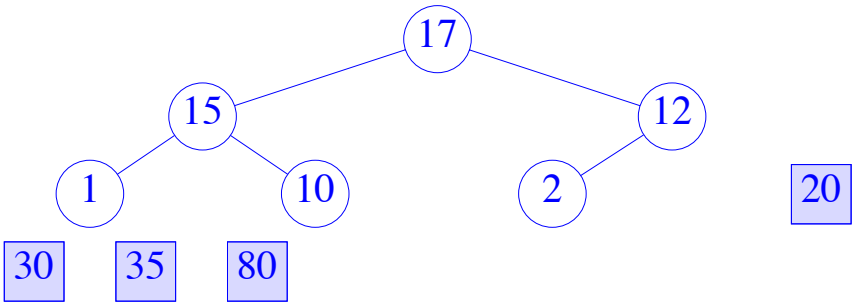


(b)

Figure 9.9 Heap Sort



(c)



(d)

Figure 9.9 Heap sort (Continuation)

MaxHeap Class

```
template<class T>
class MaxHeap {
   public:
      MaxHeap(int MaxHeapSize = 10);
      ~MaxHeap() {delete [] heap;}
      int Size() const {return CurrentSize;}
      T Max() {if (CurrentSize == 0)
                  throw OutOfBounds();
               return heap[1];}
      MaxHeap<T>& Insert(const T& x);
      MaxHeap<T>& DeleteMax(T& x);
      void Initialize(T a[], int size, int ArraySize);
      void Deactivate() {heap = 0;}
      void Output() const;
   private:
      int CurrentSize, MaxSize;
      T *heap;  // element array
};

template<class T>
MaxHeap<T>::MaxHeap(int MaxHeapSize)
{// Max heap constructor.
   MaxSize = MaxHeapSize;
   heap = new T[MaxSize+1];
   CurrentSize = 0;
}
```

Insertion

```
template<class T>
MaxHeap<T>& MaxHeap<T>::Insert(const T& x)
{// Insert x into the max heap.
   if (CurrentSize == MaxSize)
      throw NoMem(); // no space

   // find place for x
   // i starts at new leaf and moves up tree
   int i = ++CurrentSize;
   while (i != 1 && x > heap[i/2]) {
      // cannot put x in heap[i]
      heap[i] = heap[i/2]; // move element down
      i /= 2;              // move to parent
      }

   heap[i] = x;
   return *this;
}
```

Deletion

```
template<class T>
MaxHeap<T>& MaxHeap<T>::DeleteMax(T& x)
{// Set x to max element and delete
 // max element from heap.
   // check if heap is empty
   if (CurrentSize == 0)
      throw OutOfBounds(); // empty

   x = heap[1]; // max element

   // restucture heap
   T y = heap[CurrentSize--]; // last element

   // find place for y starting at root
   int i = 1,  // current node of heap
       ci = 2; // child of i
   while (ci <= CurrentSize) {
      // heap[ci] should be larger child of i
      if (ci < CurrentSize &&
          heap[ci] < heap[ci+1]) ci++;

      // can we put y in heap[i]?
      if (y >= heap[ci]) break;   // yes

      // no
      heap[i] = heap[ci]; // move child up
      i = ci;             // move down a level
      ci *= 2;
      }
   heap[i] = y;

   return *this;
}
```

Initialize a Nonempty Max Heap

```
template<class T>
void MaxHeap<T>::Initialize(T a[], int size,
                            int ArraySize)
{// Initialize max heap to array a.
   delete [] heap;
   heap = a;
   CurrentSize = size;
   MaxSize = ArraySize;

   // make into a max heap
   for (int i = CurrentSize/2; i >= 1; i--) {
      T y = heap[i]; // root of subtree

      // find place to put y
      int c = 2*i; // parent of c is target
                   // location for y
      while (c <= CurrentSize) {
         // heap[c] should be larger sibling
         if (c < CurrentSize &&
             heap[c] < heap[c+1]) c++;

         // can we put y in heap[c/2]?
         if (y >= heap[c]) break;  // yes

         // no
         heap[c/2] = heap[c]; // move child up
         c *= 2; // move down a level
         }
      heap[c/2] = y;
      }
}
```

HeapSort

```
template <class T>
void HeapSort(T a[], int n)
{// Sort a[1:n] using the heap sort method.
   // create a max heap of the elements
   MaxHeap<T> H(1);
   H.Initialize(a,n,n);

   // extract one by one from the max heap
   T x;
   for (int i = n-1; i >= 1; i--) {
      H.DeleteMax(x);
      a[i+1] = x;
      }

   // save array a from heap destructor
   H.Deactivate();
}
```

In-Place HeapSort

Go back to the ppt slides