COS 460 – Algorithms (week 05)

Sorting Algorithms. Quick Sort. Priority Queue.

Writing robust sorting functions that can sort any type of data into sorted order using the data type's natural order.

- Client passes array of objects to sorting routine.
- Sorting routine calls object's comparison function as needed.

Consistency. It is the programmer's responsibility to ensure that the comparison function specifies a **total order**.

- Transitivity: if $a < b$ and $b < c$, then $a < c$.
- Trichotomy: either (i) $a < b$ or (ii) $b < a$ or (iii) $a = b$.

Built-in comparable types: `int`, `double`, `char`, `string`, `bool`.

User-defined comparable types. Implementation of the comparable interface. Example:

```
#include<string>
#include<iostream>
using namespace std;

struct date
{
 int month, day, year;
 date(){};
 date(int m, int d, int y)
 {
  month = m;
  day = d;
  year = y;
 }
 bool cmp(date b)
 {
  if (year < b.year ) return true;
  if (year > b.year ) return false;
  if (month < b.month) return true;
  if (month > b.month) return false;
  if (day < b.day ) return true;
  if (day > b.day ) return false;
  return false;
  }
};
/*
bool cmp(date a, date b)
{
if (a.year < b.year) return true;
if (a.year > b.year) return false;
if (a.month < b.month) return true;
if (a.month > b.month) return false;
if (a.day < b.day) return true;
if (a.day > b.day) return false;
return false;
}
*/
```

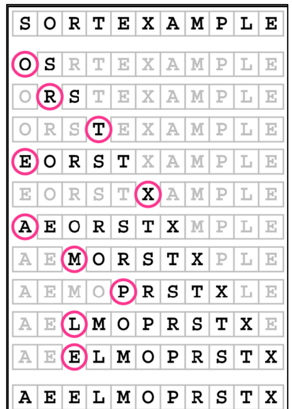
```
template<class T>
bool operator < (T a, T b)
{
  return a.cmp(b);
}

int main()
{
  date a(1,1,2007),b(2,1,2007);
  cout << (a<b)<<endl;
  int c=3,d=4;
  cout << (c<d)<<endl;
  system("pause");
}
```

Elementary Sorting Methods

Insertion sort.

- Scans from left to right by moving the position *i*.
- Element to right of *i* are not touched.
- Invariant: elements to the left of *i* are in ascending order.
- Inner loop: repeatedly swap element *i* with element to its left.

For each `i` from `2` to `N`, the elements `a[1]`, ..., `a[i]` are sorted by putting `a[i]` into position among the sorted list of elements in `a[1]`, ..., `a[i-1]`.



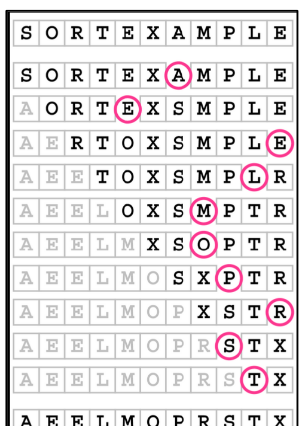
Insertion sort example

```
void InsertionSort(T a[], int N)
{
   for(int i=2; i<=N; i++)
   { T v=a[i]; int j=i;
     while(a[j-1]>v){a[j]=a[j-1];j--;}
     a[j]=v;
   }
}
```

Selection sort.

- Moving position *i* scans from left to right.
- Elements to the left of *i* are fixed and in ascending order.
- No element to left of *i* is larger than any element to its right.

The method works by repeatedly "selecting" the smallest remaining element and puts it at the last position of the sorted part.



Selection sort example

```
void SelectionSort(T a[], int N)
{
  for(int i=1; i < N; i++)
   { int min=i;
     for(int j=i+1; j<= N; j++)
      if(a[j] < a[min]) min=j;
     swap(a[i], a[min]);
   }
}
```

Digression: Bubble Sort. An elementary sorting method that is often taught in introductory classes: keep passing through the sequence, exchanging adjacent elements, if necessary; when no exchanges are required on some pass, the data are sorted.

```
void BubbleSort(T a[], int N)
{
 for(int i = N; i >= 1; i--)
 for(int j = 2; j <= i; j++)
  if(a[j-1] > a[j]) swap(a[j-1],a[j]);
}
```

Performance characteristics of elementary sorts

Property 1. Selection sort uses about $N^2/2$ comparisons and N exchanges.

Property 2. Insertion sort uses about $N^2/4$ comparisons and $N^2/8$ exchanges on the average, twice as many in the worst case.

.**Property 3.** Bubble sort uses about $N^2/2$ comparisons and $N^2/2$ exchanges on the average and in the worst case

Property 4. Insertion sort is linear for "almost sorted" data.

Sorting files with large records

It is desirable to arrange elements so that **any** sorting method uses only N "exchanges" of full records, by having the algorithm operate indirectly on the file (using an array of indices) and then do the rearrangement afterwards.

Specifically, if the array `a[1], ..., a[N]` consists of large records, then we prefer to manipulate an index array `p[1],...,p[N]` accessing the original array only for comparisons. If we define `p[i]=i` initially, many sorting algorithms need only be modified to refer to `a[p[i]]` rather then `a[i]` when using `a[i]` in a comparison, and to refer to `p` rather then `a` when doing data movement. This produces an algorithm that will sort the index array so that `p[1]` is the index of the smallest element in `a`, and so on. The cost of moving large records is avoided.

```
void InsertionSort_LargeRecords(T a[], int N, int p[])
{
   for(int i=0; i <= N; i++) p[i]=i;
   for(int i=2; i<=N; i++)
   { int v=p[i]; int j=i;
     while(a[p[j-1]]>a[v]){p[j]=p[j-1];j--;}
     p[j]=v;
   }
}
```

Distribution Counting

A very special situation for which there is a simple sorting algorithm is the following: "sort a file of N records whose keys are distinct integers between 1 and N". Using a temporary array `b`:

```
for(int i=1; i<=N; i++) b[a[i]] = a[i];
for(int i=1; i<=N; i++) a[i] = b[i];
```

Exercise: Solve this problem without an auxiliary array.

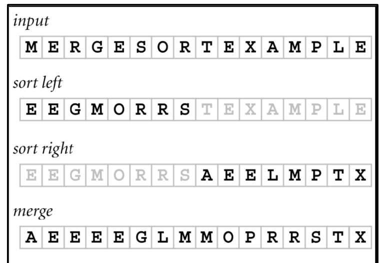
A more realistic example: "sort a file of N records whose keys are integers between 0 and $M-1$". If M is not too large, we may use an array `c` for *distribution counting*:

```
for(int j=0; j<M; j++) c[j]=0;
for(int i=1; i<=N; i++) c[a[i]]++;
for(int j=1; j<M; j++) c[j] += c[j-1];
for(int i=N; i>=1; i--) b[c[a[i]]--]=a[i];
for(int i=1; i<=N; i++) a[i] = b[i];
```

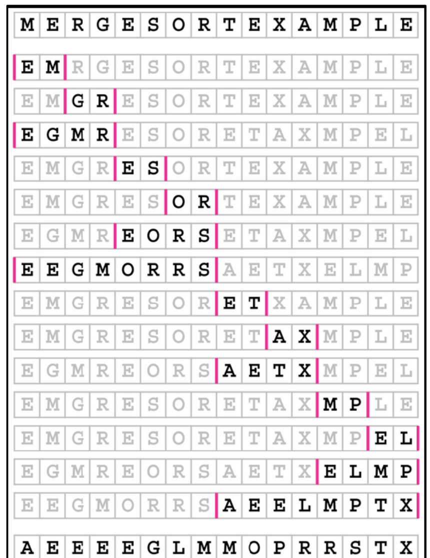
Mergesort and Quicksort: Two great sorting algorithms

Mergesort:

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.



Mergesort



Top-down mergesort

```
void merge(int a[],
           int beg, int mid, int end)
{ int n = end - beg + 1;
  int b[n];
  int i1 = beg;
  int i2 = mid + 1;
  int j = 0;
  while ((i1 <= mid) && (i2 <= end))
  { if (a[i1] < a[i2])
      {b[j]=a[i1]; i1++;}
    else {b[j]=a[i2]; i2++;}
    j++;
  }
  while(i1 <= mid)
   {b[j]=a[i1]; i1++; j++;}
  while(i2 <= end)
   {b[j]=a[i2]; i2++; j++;}
  for(j=0; j<n; j++) a[beg+j] = b[j];
}
```

Example run:

```
int a[]=
{5,9,10,12,17,8,11,20,32};
merge(a,0,4,8);
```

(5,9,10,12,17)(8,11,20,32) $\Rightarrow$ (1)
(9,10,12,17)(8,11,20,32) $\Rightarrow$ (1,5)
(9,10,12,17)(11,20,32) $\Rightarrow$ (1,5,8)
(10,12,17)(11,20,32) $\Rightarrow$ (1,5,8,9)
(12,17)(11,20,32) $\Rightarrow$ (1,5,8,9,10)
(12,17)(20,32) $\Rightarrow$ (1,5,8,9,10,11)
(17)(20,32) $\Rightarrow$ (1,5,8,9,10,11,12)
()(20,32) $\Rightarrow$ (1,5,8,9,10,11,12,17)
()(32) $\Rightarrow$ (1,5,8,9,10,11,12,17,20)
()() $\Rightarrow$ (1,5,8,9,10,11,12,17,20,32)

Merge Sort: Complete sorting program:

```
void merge_sort(int a[],int beg, int end)
{ if (beg == end) return;
  int mid = (beg + end) / 2;
  merge_sort(a, beg, mid);
  merge_sort(a, mid + 1, end);
  merge(a, beg, mid, end);
}
```

Mergesort Analysis: Memory

How much memory does mergesort require?

- Original input array = N.
- Auxiliary array for merging = N.
- Local variables: constant.
- Function call stack: $\log_2 N$.
- Total = 2N + O(log N).

How much memory do other sorting algorithms require?

- N + O(1) for insertion sort and selection sort.
- In-place = N + O(log N).

Challenge for the bored. In-place merge.

Mergesort Analysis: Running Time

Def. T(N) = number of comparisons to mergesort an input of size N.
Mergesort recurrence.

$$T(N) \le \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{T(\lceil N/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor N/2 \rfloor)}_{\text{solve right half}} + \underbrace{N}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. $T(N) = O(N \log_2 N)$.
Note: same number of comparisons for any input of size N.
We prove $T(N) = N \log_2 N$ when N is a power of 2, and = instead of ≤.

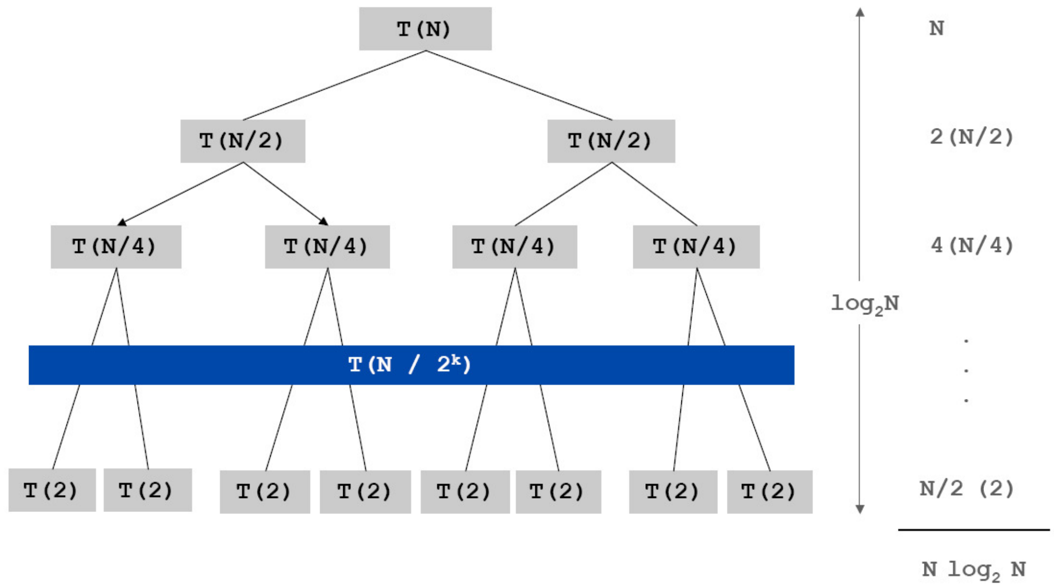
Proof by Induction

- Base case: n = 1.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n\left(\log_2 (2n) - 1\right) + 2n \\ &= 2n \log_2 (2n) \end{aligned}$$

Proof by Recursion Tree

$$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ \underbrace{2T(N/2)}_{\text{sorting both halves}} + \underbrace{N}_{\text{merging}} & \text{otherwise} \end{cases}$$



Good algorithms are better than supercomputers.
Home PC executes 10^8 comparisons/second. Supercomputer executes 10^{12} comparisons/second.

Insertion Sort (N^2)

computer	thousand	million	billion
home	instant	2.8 hours	317 years
super	instant	1 second	1.6 weeks

Mergesort (N log N)

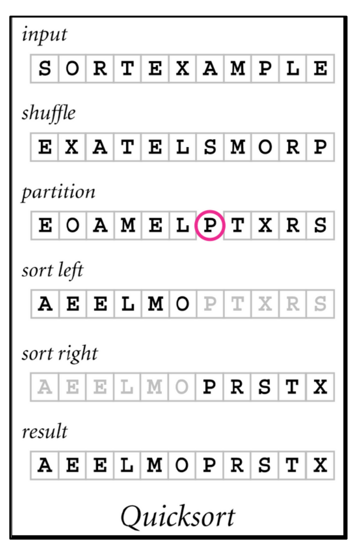
thousand	million	billion
instant	1 sec	18 min
instant	instant	instant

QuickSort

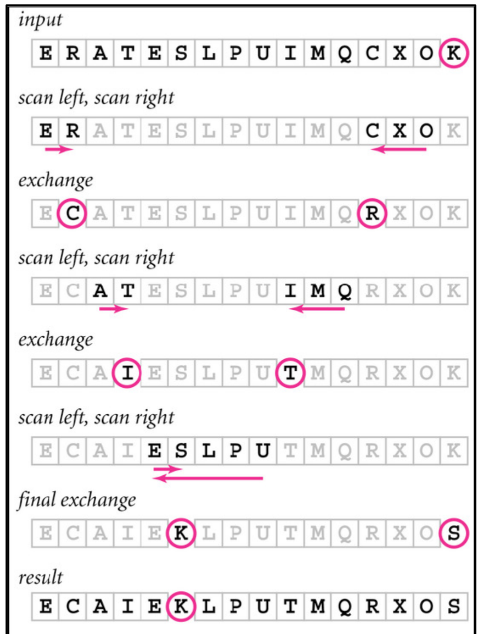
- Shuffle the array.
- Partition array so that:
  - element a[i] is in its final place for some i
  - no larger element to the left of i
  - no smaller element to the right of i
- Sort each piece recursively.

The basic algorithm

```
void quicksort(T a[], int L, int R)
{ if(L<R)
   {int p = partition(a,L,R);
    quicksort(a,L,p-1);
    quicksort(a,p+1,R);

   }
}
```



```
int partition(T a[], int L, int R)
{
  T v=a[R]; int i=L-1; int j=R;
  while(1)
  {
    while(a[++i]<v);
    while(a[--j]>v);
    if(i>=j) break;
    swap(a[i],a[j]);
  }
  swap(a[i],a[R]);
  return i;
}
```



Partitioning example

Quick Sort Algorithm (another implementation):

```
void qsort(int a[], int L, int R)
{ int i=L, j=R;
  int x=a[(i + j)/2];
  do
  { while (a[i] < x) i++;
    while (a[j] > x) j--;
    if (i <= j)
    { swap(a[i], a[j]); i++; j--; }
  }
  while (i <= j);
  if (L < j) qsort(a, L, j);
  if (i < R) qsort(a, i, R);
}
```

Quicksort: Performance Characteristics
Worst case. Number of comparisons is quadratic.

- $N + (N-1) + (N-2) + \ldots + 1 \approx N^2 / 2$.
- More likely that your computer is struck by lightning.

Caveat. Many textbook implementations go quadratic if input:

- Is sorted.
- Is reverse sorted.
- Has many duplicates.

Quicksort: Average Case
Average case running time.

- Roughly 2 N ln N comparisons.
- Assumption: file is randomly shuffled.

Remarks.

- 39% more comparisons than mergesort.
- Faster than mergesort in practice because of lower cost of other high-frequency instructions.
- Caveat: many textbook implementations have best case N^2 if duplicates, even if randomized!

Theorem. The average number of comparisons C_N to quicksort a random file of N elements is about 2N ln N.
The precise recurrence satisfies $C_0 = C_1 = 0$ and for $N \geq 2$:

$$
\begin{aligned}
C_N &= N+1+\tfrac{1}{N}\sum_{k=1}^{N}\left(C_k + C_{N-k}\right)\\
&= N+1+\tfrac{2}{N}\sum_{k=1}^{N} C_{k-1}
\end{aligned}
$$

Multiply both sides by N and subtract the same formula for N-1:

$$N\,C_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}$$

Simplify to:

$$NC_N = (N+1)\,C_{N-1} + 2N$$

Divide both sides by N(N+1) to get a telescoping sum:

$$
\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1}\\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}\\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1}\\
&= \vdots\\
&= \frac{C_2}{3} + \sum_{k=3}^{N} \tfrac{2}{k+1}
\end{aligned}
$$

Approximate the exact answer by an integral:

$$\frac{C_N}{N+1} \approx \sum_{k=1}^{N} \tfrac{2}{k} \approx \int_{k=1}^{N} \tfrac{2}{k} = 2\ln N$$

Finally, the desired result:

$$C_N \approx 2(N+1)\ln N \approx 1.39 N \log_2 N. \quad \blacksquare$$

Comparison Based Sorting Lower Bound

Theorem. Any comparison based sorting algorithm must use $\Omega(N \log_2 N)$ comparisons.
Proof.

- Suffices to establish lower bound when input consists of N distinct values a_1 through a_N.
- Worst case dictated by tree height h.
- N ! different orderings.
- (At least) one leaf corresponds to each ordering.
- Binary tree with N ! leaves must have height (Stirling formula):

$$
\begin{aligned}
h &\geq \log_2(N!)\\
&\geq \log_2(N/e)^N\\
&= N\log_2 N - N\log_2 e\ .
\end{aligned}
$$

Priority Queue:
Generalizes: stack, queue, randomized queue.

Data. Items that can be compared.

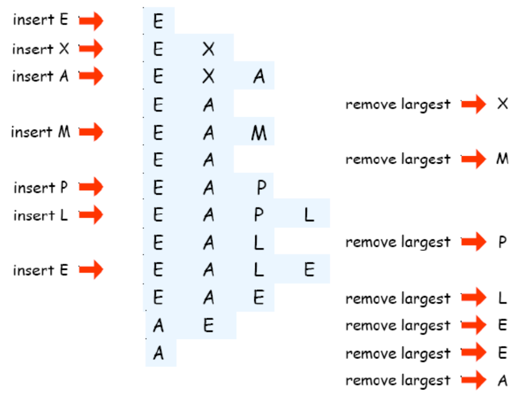
Basic operations.

- Insert.
- Remove largest.

defining ops

- Copy.
- Create.
- Destroy.
- Test if empty.

generic ops



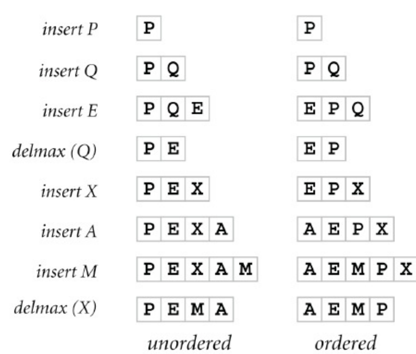
Applications.

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Computational number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [maintain largest M values in a sequence]
- Operating systems. [load balancing, interrupt handling]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

Two elementary implementations.

Implementation	Insert	Del Max
unordered array	1	N
ordered array	N	1

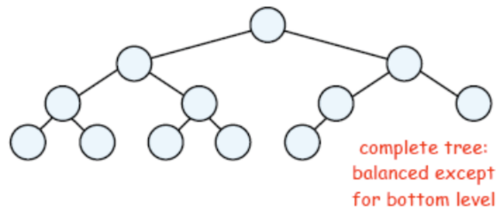
worst-case asymptotic costs for PQ with N items



Heap: Array representation of a heap-ordered complete binary tree.

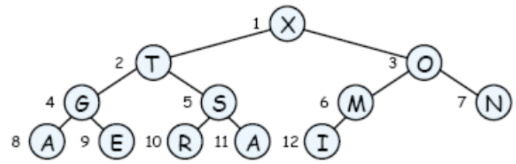
Binary tree.

- Empty or
- Node with links to left and right trees.



Heap-ordered binary tree.

- Keys in nodes.
- No smaller than children's keys.

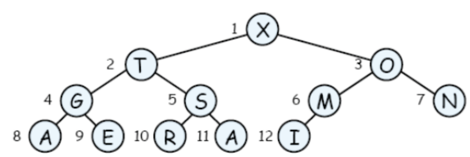


Array representation.

- Take nodes in level order.
- No explicit links needed since tree is complete.

1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

Property A. Largest key is at root.



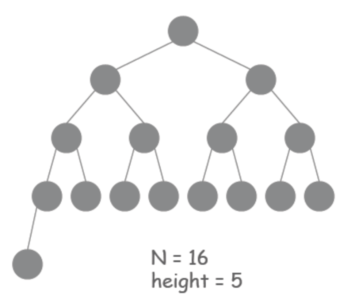
Property B. Can use array indices to move through tree.

- Note: indices start at 1.
- Parent of node at k is at k/2.
- Children of node at k are at 2k and 2k+1.

1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

Property C. Height of N node heap is $1 + \lfloor \lg N \rfloor$.

height only increases when
N is a power of 2



Priority queue
A simple implementation of a Priority queue using an array:

```
 class pq
{
  private:
   int n;
   int a[100];
  public:
   pq(int v)
   { n=1;
     a[1]=v;
   }
   int size(){return n;}
   void add(int v)
   {
    n++;
    a[n]=v;
    int p=n;
    while(p/2>0)
    {
     if(a[p] <= a[p/2]) return;
     if(p%2==0) if(p+1>n)
      {a[p]=a[p/2]; a[p/2]=v; p=p/2;
           continue;
      }
     p=2*(p/2);
     if(a[p+1]>a[p])
      {a[p+1]=a[p/2];a[p/2]=v;p=p/2;}
       else
      {a[p]=a[p/2]; a[p/2]=v; p=p/2;}
     }
   }
```

```
int get()
   { int w=a[1];
     int v=a[n];
     n--;
     a[1]=v;
     int p=1;
     while(1)
     {
      if(2*p>n) break;
      if(2*p+1>n)
       {if(a[p]<a[2*p])
        {a[p]=a[2*p]; a[2*p]=v;}
        break;}
      if((a[p]>a[2*p])&&(a[p]>a[2*p+1]))
        break;
      if(a[2*p]>a[2*p+1])
        {a[p]=a[2*p]; a[2*p]=v; p=2*p;}
      else
        {a[p]=a[2*p+1]; a[2*p+1]=v;
         p=2*p+1;
        }
     }
     return w;
   }
};
```

Operation	Insert	Remove Max	Find Max
ordered array	N	1	1
ordered list	N	1	1
unordered array	1	N	N
unordered list	1	N	N
binary heap	lg N	lg N	1

worst-case asymptotic costs for PQ with N items

Exercises:

1. Write a complete program to test the implementation of a Priority queue ADT.
2. Write a member function like `get()`, but that only returns the element, not to remove it.
3. What is the complexity of `add()` and `get()` functions?
4. Write a universal implementation of a Priority queue – replace data field `int` with a class.
5. Write a universal implementation of a Priority queue – replace ordering relation "<=" with an arbitrary function that has property of "order".
6. Write *naive* implementation of a Priority queue with O(n) complexity of `add()` and/or `get()` functions.
7. Write a function to sort a sequence of data using Priority queue ADT.
8. Find the complexity of the above sorting algorithm.
9. Make a study of STL priority_queue Class.

STL example:

```
#include<iostream>
#include<queue>
using namespace std;

struct data
 {string s; double b; int r;};
struct cmp:
 public binary_function<data, data, bool>
{ bool operator()(data x, data y)
   {if(x.b < y.b) return true;
    if(x.b==y.b) if(x.r > y.r) return true;
    return false;
   }
};
priority_queue<data, vector<data>, cmp> Q;
int main()
{int c=0;
 int n; cin >> n;
 for(int i=1;i<=n;i++)
 { int m; cin >> m;
   for(int j=1;j<=m;j++)
    { data d;
      cin >> d.s >> d.b;
      d.r=c; c++;
      Q.push(d);
    }
   int k; cin >> k;
   for(int j=1; j<=k; j++) Q.pop();
 }
 cout << (Q.top()).s << endl;
}
```