Computational Complexity

Topics

- Algorithm analysis
- Fast algorithm synthesis
- Empirical measurement of complexity

"Complexity" in the algorithm-analysis context

- means the cost of a program's ***execution***
  - (running time, memory, ...)

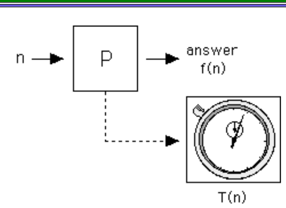
  rather than

- the cost of creating the program
  - (# of statements, development time)
- In this context, **less**-complex programs may require **more** development time.

Functions associated with a program

- Consider a program with one natural number as input.
- Two functions that can be associated with the program are:
  - f(n), the function *computed* by the program
  - T(n), the *running time* of the program

Running time T(n)



It is common to measure T based on the **size** of the input, rather than the input value itself.

Possible size measures for T(n)

- Total number of bits used to encode the input.
- Number of data values in the input (e.g. size of an array)

  The second is viewed as an *approximation* to the first.

Primitive Operations

- These are operations which we don't further decompose in our analysis.
- They are considered the fundamental building blocks of the algorithm, e.g.
  + * - / if( )
- Typically, the time they take is assumed to be **constant**.
- This assumption is not always valid.

Is multiplication really "primitive"?

- In doing arithmetic on arbitrarily-large numbers, the size of the numerals representing those numbers may have a definite effect on the time required.
  - 2 x 2

vs.

  - 26378491562329846 x 786431258901237

Size of Numerals

- For a single number (n) input, size of the corresponding numeral is typically on the order of log(n)
- e.g. decimal numeral encoding:
  - size(n) = #digits of n
  - = $\lceil \log_{10} n \rceil$
- $\lceil x \rceil$ = smallest integer $\geq$ x (called the "ceiling of" x)

Asymptotic Analysis

- Typically in measuring complexity, we look at the *growth-rate* of the time function rather than the value itself.
- There is therefore a tendency to pay less attention to **constant factors**.
- This is called ***asymptotic analysis***.
- It is only one part of the story; sometimes constants *are* important.

All logs are Similar*

- Logarithm functions have the same growth rate regardless of base.
- So we don't care too much about what base is used. $\log_2$ is the most common assumption.

* Ronald Reagan, former California Governor: "If you've seen one redwood tree, you've seen 'em all."

Chain Rule for Logs

- $\log_c(x) = \log_c(b)\ \log_b(x)$
- $\log_{10}(x) = \log_{10}(2)\ \log_2(x) = 0.301\ \log_2(x)$
- $\log_2(x) = 3.32\ \log_{10}(x)$

Step Counting

- Using exact running time to measure an algorithm requires calibration based on the type of machine, clock rate, etc.
- Instead, we usually just ***count steps*** taken in the algorithm.
- Often we will assume primitives take one step each.
- This is usually enough to give us an accurate view of the **growth rate** of running time.

Straight-Line Code

```
x = x + 1;
v = x / z;
w = x + v;
```

3 operations, therefore

3 steps

Loop Code

(These count as steps too.)

```
for( int i = 0; i < n; i++ )
  {
  x = x + 1;
  v = x / z;
  w = x + v;
  }
```

n iterations x 5 steps + 1

= 5n+1 steps

Non-Numeric Loop Control

```
for( Polylist L = A; !L.isEmpty(); L = L.rest(); )
  {
  ... loop body ...
  }
```

# of iterations = A.length()

Recursive Code

```
fac(n) = n == 0 ? 1 : n*fac(n-1)
```

2n +1 steps, if we count multiply and - as steps;

Steps are involved in the overhead for **function calls** too. The number of such steps would be **proportional** to n in this case.

Recurrence Formulas Represent Time for Recursive Expressions

```
fac(n) = n == 0 ? 1 : n*fac(n-1)
```

```
T(0) => 1;
T(n) => T(n-1) + 2;
```

recurrence for counting steps as a function of input n

Solving Recurrence Formulas
One Method: Repeated Substitution

```
T(0) => 1;
T(n) => T(n-1) + 2;

T(n) = T(n-1) + 2
     = T(n-1-1) + 2 + 2
     = T(n-1-1-1) + 2 + 2 + 2
     ...
     = T(0) + n*2
     = 2n+1
```

use the above formulas repeatedly

T(n)= 2n+1 is **closed form** solution

Solving Recurrence Formulas
Another Example

```
T(0) => 1;
T(n) => 2*T(n-1);

T(n) = 2*T(n-1)
     = 2*2*T(n-2)
     = 2*2*2*T(n-3)
     ...
     = 2^n*T(n-n)
     = 2^n
```

use the above formulas repeatedly

T(n)= 2^n is closed form solution

Try This One

```
T(0) => 0;
T(n) => n + T(n-1);
```

Gauss' 3rd Grade Technique

Compute 1+2+3+ … + 1000:

1+1000 + 2+999 + 3+998 + … + 500+501 =

500*1001 =

500500

In general, 1+2+3+…+n = n*(n+1)/2

"O" Notation

- "O" is letter "Oh" (for "order")
- This is "big-Oh"; "little-Oh" has a different meaning
- Used to express **upper bounds** on running time (number of steps)
- $T \in O(f)$ means that
  $(\exists c)(\forall n)\ T(n) < cf(n)$
- (Think of O(f) as the set of functions that grow no faster than a constant times f.)

"O" Notation

- Constants are irrelevant in "O" comparisons:
- If g is a function, then any function $n \to d*g(n)$, where g is a constant, is $\in O(g)$.
- Examples:
  - $n \Rightarrow 2.5*n^2 \in O(n \to n^2)$
  - $n \Rightarrow 100000000*n^2 \in O(n \to n^2)$

Notation Abuse

- It is customary to drop the n => and just use the body expression of the anonymous function
- Examples:
  - $O(n^2)$ instead of $O(n \to n^2)$
  - $O(n)$ instead of $O(n \to n)$
  - $O(1)$ instead of $O(n \to 1)$

Notation Abuse

- Examples:
  - $2.5 * n^2 \in O(n^2)$
  - $100000000 * n \in O(n)$
  - $10000000000 \in O(1)$

Words for Functions

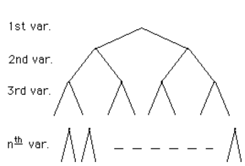
- A function f is called
  - ***cubic*** if $f \in O(n^3)$
  - ***quadratic*** if $f \in O(n^2)$
  - ***linear*** if $f \in O(n)$
  - ***constant*** if $f \in O(1)$
  - ***polynomial*** if $f \in O(n^k)$, for some constant k
  - ***logarithmic*** if $f \in O(\log(n))$
  - ***exponential*** if $f \in O(2^{p(n)})$, for some polynomial p(n)
- What are some algorithms that grow at these rates?

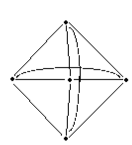
"O" Notation

- Example algorithms
  - $O(n^3)$: Computing the reachability matrix
  - $O(n^2)$: A naïve sorting program
  - $O(n)$: Search in an unordered array
  - $O(1)$: accessing an array given an index
  - $O(n^k)$: all of the above
  - $O(\log(n))$: binary search of an ordered array

Other Words

- "Slow" algorithms (maybe the problem's fault)
  - $O(2^n)$: Tautology checking using Boole/Shannon
  - $O(n!)$: Traveling salesman





"O" Notation

- $O(\log(n))$: a desirable complexity
- As the problem doubles in size, the number of steps added increases by just 1.
- Recurrence:

  T(0) => 1;

  T(2*n) => T(n+1);

"O" Notation

- Recurrence:
  T(1) => 0;
  T(2*n) => T(n) + 1;
- T(N) = T(N/2) + 1
  = T(N/4) + 2
  = T(N/8) + 3
  …
  = T(N/2^k)+k
  = 0 + log(N) when N = 2^k

Rules for "O"
Additive Rule

- f+g ∈ O(max(f, g))
  Here f+g means n => f(n) + g(n)
  max(f, g) means n => max(f(n), g(n))
- If g ∈ O(f), then f+g ∈ O(f)
- For polynomials, only the highest order term matters, e.g.
  n^5+1000000n^3+10000n^2 ∈ O(n^5)

Multiplicative Rule

- If f ∈ O(g), then h*f ∈ O(h*g)
  where h*f means n => h(n)*f(n)
- For example,
  n*log(n) ∈ O(n^2)
  since
  log(n) ∈ O(n)

Transitive Rule

- If f ∈ O(g) and g ∈ O(h),
  then f ∈ O(h).

Derivative Rule

- If f' ∈ O(g'),

  where ' denotes the derivative,

  then f ∈ O(g).
- Example: log(n) ∈ O(n).
  This follows from the derivative rule since 1/n ∈ O(1).

Limit Rule

- If $\lim_{n\to\infty} f(n)/g(n) = k$
  then
  - If k = 0, f ∈ O(g), but not conversely
  - If k > 0, f ∈ O(g), and g ∈ O(f)

Tight Bounds

- A bound $f \in O(g)$ is *tight* if $g \in O(f)$ also

Example: $\log(n) \varepsilon O(n^{1/2})$.

- This holds provided $1/n \in O(n^{-1/2})$, by the derivative rule.
- Apply the limit rule:

  $\lim (1/n) / n^{-1/2}$

  $= \lim (n^{1/2}/n)$

  $= \lim (1/ n^{1/2})$

  $= 0$
- This bound is not tight, since the limit is 0, and therefore not $n^{-1/2} \in O(1/n)$.

Why it Matters

Running Time as a function of Complexity

log n	3.3219	6.6438	9.9658	13.287	16.609	19.931
$\log^2 n$	10.361	44.140	99.317	176.54	275.85	397.24
sqrt n	3.162	10	31.622	100	316.22	1000
n	**10**	**100**	**1000**	**10000**	**100000**	**1000000**
n log n	33.219	664.38	9965.8	132877	$1.66*10^6$	$1.99*10^7$
$n^{1.5}$	31.6	10^3	$31.6*10^4$	10^6	$31.6*10^7$	10^9
n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
n^3	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	1024	10^{30}	10^{301}	10^{3010}	10^{30103}	10^{301030}
n!	3 628 800	$9.3*10^{157}$	10^{2567}	10^{35659}	10^{456573}	$10^{5565710}$

Values of various functions vs. values of argument n.

Even a computer a trillion times faster won't help in this region

Allowable Problem Size as a Function of Available Time

Time Multiple	**10**	**100**	**1000**	**10000**	**100000**	**1000000**
log n	1024	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{30000}
$\log^2 n$	8	1024	$3*10^9$	$1.2*10^{30}$	$1.5*10^{95}$	$1.1*10^{301}$
sqrt n	100	10^4	10^6	10^8	10^{10}	10^{12}
n	10	10^2	10^3	10^4	10^5	10^6
n log n	4.5	22	140	1000	$7.7*10^3$	$6.2*10^4$
$n^{1.5}$	4	21	100	210	2100	10000
n^2	3	10	32	100	320	1000
n^3	2	4	10	21	46	100
2^n	3	6	9	13	16	19
n!	3	4	6	7	8	9

Increase in size of problem that can be run based on increase in allowed time, assuming algorithm runs problem size 1 in time 1.

Doubling Input Size

Complexity	**Doubling the input causes execution time to**
O(1)	stay the same
O(log n)	increase by an additive constant
$O(n^{1/2})$	increase by a factor of sqrt(2)
O(n)	double
O(n log n)	double, plus increase by a constant factor times n
$O(n^2)$	increase by a factor of 4
$O(n^k)$	increase by a factor of 2^k
$O(k^n)$	square

Black-box vs. White-box Complexity

- Black-box: We have a copy of the program, with no code. We can run it for different sizes of input.
- White-box (aka "Clear box"): We have the code. We can analyze it.

Black-Box Complexity

- Run the program for different sizes of data set; try to get a fix on the growth rate.
- What sizes?
  - An approach is to repeatedly **double** the input size, until testing becomes infeasible (e.g. it takes too long, or the program breaks).

Black-Box Complexity

- Run on sizes 32, 64, 128, 512, … or 10, 100, 1000, 10000, …
- For each n, get time T(n).
- How can we **estimate** the order of run-time (e.g. $O(n^2)$, $O(n^3)$, etc.)?

Black-Box Complexity

- Suppose we are trying to establish correctness of the hypothesis
  $T(n) \in O(f(n))$
- From the definition, we know this means there is a constant c such that
  for all n, $T(n) \leq c\, f(n)$
- If our hypothesis is correct, then we expect for all n,
  $T(n)/ f(n) \leq c$
- We can simply examine this ratio.

Black-Box Complexity

- If we see
  $T(n)/ f(n) \leq c$
- then the hypothesis $T(n) \in O(f(n))$ is supported.
- If T(n)/ f(n) is approximately a constant as n increases, then the bound appears to be ***tight***.
- If T(n)/ f(n) decreases as n increase, the bound is loose.
- If T(n)/ f(n) increases, we don't have a bound.

Use Analysis to Predict

- If $T(n) \in O(f(n))$ is supported, we can **predict** an upper bound for any data set size n using f and knowing the implied constant.

Examples: Sorting Programs

- See turing:/cs/cs60/examples/java/Sort
- Use unix command:
  run <type>
  which will try random dataset sizes in the ranges 10, 100, 1000, 10000
- type can be one of:
  - quicksort, heapsort, minsort, radixsort, bucketsort

Example: minsort

```
Sorting 10 elements using minsort took 1 ms
Sorting 100 elements using minsort took 4 ms
Sorting 1000 elements using minsort took 337 ms
Sorting 10000 elements using minsort took 35728 ms
```

Suppose we hypothesize $T(n) \in O(n^2)$.

n	T(n)	T(n)/n*n
10	1	0.0100000
100	4	0.0004000
1000	337	0.0003370
10000	35728	0.0003573

The hypothesis is supported. Moreover, we can predict that T(n) is about 0.0004 n^2 milliseconds for large n.

Example: minsort

We predict that T(n) is about 0.0004 n^2 milliseconds for large n.

How long will it take to sort 100,000 items? one million items?

n	predicted T(n) (ms)	days
10000	40000	0.000462964
100000	4000000	0.0462964
1000000	400000000	4.62964

White-Box Complexity

- Here examine the code (loop structure, etc.)

Analysis of Typical Loops

```
for( int j = 0; j < n; j++ )
  {
     O(1)
  }
```

Means some constant-time computation.

Assume it does not modify loop index.

Complexity: $O(n)$

Analysis of Typical Loops

```
for( int j = n; j > 0; j-- )
  {
     O(1)
  }
```

Analysis of Typical Loops

```
for( int j = 0; j < n; j++ )
  for( int k = 0; k < n; k++ )
  {
     O(1)
  }
```

Analysis of Typical Loops

```
for( int j = 0; j < n; j++ )
  for( int k = 0; k < j; k++ )
  {
     O(1)
  }
```

Analysis of Typical Loops

```
for( int j = 1; j <= n; j = 2*j )
  {
     O(1)
  }
```

j doubles each time.
The loop stops by iteration k, where 2^k>= n, i.e. by ceiling(log(n)) iterations.

Complexity: $O(\log(n))$ This bound is tight.

Analysis of Typical Loops

```
for( int j = n; j > 0; j = j/2 )
  {
     O(1)
  }
```

Analysis of Typical Loops

```
for( int j = 1; j < n; j++ )
  for( int k = j; k > 0; k = k/2 )
    {
     O(1)
    }
```

Analysis of Typical Loops

```
for( int j = 1; j < n; j++ )
  for( int k = j; k > 0; k = k/2 )
    {
     O(1)
    }
```

$\log(1) + \log(2) + \underbrace{\ldots + \log(n-2) + \log(n-1)}_{n/2 \text{ terms, each } \geq \log(n/2) \Rightarrow \text{bound is tight}} \in O(n \log(n))$.

Analysis of Typical Loops

```
for( int j = n; j > 0; j = j/2 )
  for( int k = 1; k <= j; k++ )
    {
     O(1)
    }
```

Analysis of Typical Loops

```
for( int j = n; j > 0; j = j/2 )
  for( int k = 1; k <= j; k++ )
    {
     O(1)
    }
```

Complexity: $O(n \log(n))$, but this is **not tight**.

Analysis of Typical Loops

```
for( int j = n; j > 0; j = j/2 )
  for( int k = 1; k <= j; k++ )
    {
     O(1)
    }
```

$n + n/2 + n/4 + \ldots + 1 < 2n$

Therefore $\in O(n)$.

Analysis of Actual Programs

Sorting Programs:

The "fruit flies" of algorithm analysis

minsort: sorting by selecting minima

```
class minsort
  {
  private double array[];       // The array being sorted

  //  Calling minsort constructor on array of doubles sorts
  //  the array elements 0..(N-1).

  minsort(double array[], int N)
    {
    this.array = array;

    for( int i = 0; i < N; i++ )
      {
      swap(i, findMin(i, N));
      }
    }
```

minsort

```
// swap(i, j) interchanges the values
// in array[i] and array[j]

  void swap(int i, int j)
    {
    double temp = array[i];
    array[i] = array[j];
    array[j] = temp;
    }
```

minsort

```
// findMin(M, N) finds the index of the minimum among
// array[M], array[M+1], ...., array[N-1].

  int findMin(int minSoFar, int N)
    {
    // by default, the element at minSoFar is the minimum

    for( int j = minSoFar+1; j < N; j++ )
      {
      if( array[j] < array[minSoFar] )
        {
        minSoFar = j;   // a smaller value is found
        }
      }
    return minSoFar;
    }
```

minsort

```
// findMin(M, N) finds the index of the minimum among
// array[M], array[M+1], ...., array[N-1].

  int findMin(int minSoFar, int N)
    {
    // by default, the element at minSoFar is the minimum

    for( int j = minSoFar+1; j < N; j++ )
      {
      if( array[j] < array[minSoFar] )
        {
        minSoFar = j;   // a smaller value is found
        }
      }
    return minSoFar;
    }
```

Analysis: $\leq$ N - minSoFar - 2 steps

minsort

```
// swap(i, j) interchanges the values
// in array[i] and array[j]

  void swap(int i, int j)
    {
    double temp = array[i];
    array[i] = array[j];
    array[j] = temp;
    }
```

Analysis: 3 steps

minsort

```
class minsort
  {
  private double array[];       // The array being sorted

  //  Calling minsort constructor on array of doubles sorts
  //  the array elements 0..(N-1).

  minsort(double array[], int N)
    {
    this.array = array;

    for( int i = 0; i < N; i++ )
      {
      swap(i, findMin(i, N));
      }
    }
```

N-i+2 times

$\leq$ N-i-2 steps

3 steps

minsort

N-i+2 steps
i ranges from 0 to N-1
(N+2) + (N+1) + … +3 steps

O(n^2) steps

Similar analysis, with the same result, is obtained for:
bubble sort
simple insertion sort

(For a live demo, see:
http://www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html)

insertion sort in rex

```
isort([]) => [];

isort([A | X]) => insert(A, isort(X));

// insert inserts the first item into a list
// in the proper place, assuming the list
// is in order

insert(A, []) => [A];

insert(A, [B | X]) =>
  A < B ? [A, B | X] : [B | insert(A, X)];
```

recurrence for isort

```
isort([]) => [];

isort([A | X]) => insert(A, isort(X));
```

argument below is the length of the list

$T_{isort}(0)$ => 0;

$T_{isort}(N)$ => $T_{isort}(N-1) + T_{insert}(N-1)$;

recurrence for insert

```
insert(A, []) => [A];

insert(A, [B | X]) =>
  A < B ? [A, B | X] : [B | insert(A, X)];
```

$T_{insert}(0) = 1$

$T_{insert}(N) \leq 1 + T_{insert}(N-1)$;

Solving:

$T_{insert}(N) \in O(N)$.

returning to recurrence for isort

$T_{isort}(0)$ => 0;

$T_{isort}(N)$ => $T_{isort}(N-1) + T_{insert}(N-1)$;

$\leq T_{isort}(N-1) + cN$

Solving

$T_{isort}(N) = c(1+2+ \ldots + N)$

$O(n^2)$ steps

Is $O(n^2)$ the best we can do for sorting?

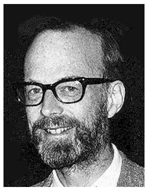
Algorithm Speedup Techniques

- Divide and Conquer
- The "digital" principle: Use data values to do selection or direct accessing
- Try trees instead of linear arrangement
- "Dynamic" programming (later)

Technique #1: Divide-and-Conquer

- Rearrange the array to be sorted:
  - Low elements on the bottom
  - High elements on the top
  - Use some element as the "pivot" value
- Sort the low and high portions recursively
- Called "quicksort"
- Invented by C.A.R. (Tony) Hoare

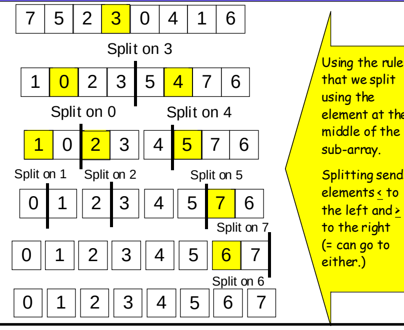
Tony Hoare



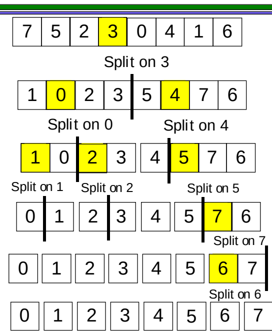
Professor C.A.R. Hoare, FRS

James Martin Professor of Computing
Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, England.

Quicksort Illustrated



Quicksort Illustrated



Splitting uses O(n) steps at each level.

Overall steps = cn x number of levels.

Quicksort Analysis

- Overall steps = cn x number of levels.
- O(log(n)) levels in *optimistic* case.
- O(n) levels in *pessimistic* case.
- Overall O(n^2).
- The ***average*** case can be shown to be **O(n log(n))** based on a probabilistic argument, assuming the data are initially randomly distributed.

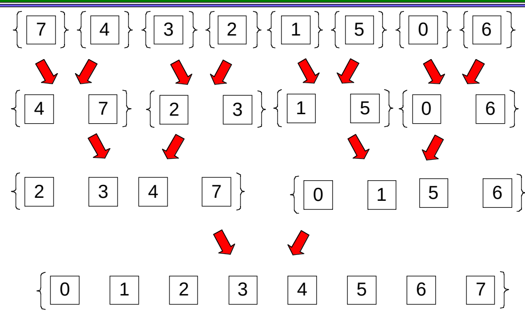
Another Divide-and-Conquer Sort Mergesort

- Sort by successively merging longer and longer sorted sequences.
- Useful with linked lists, or large files.
- More difficult to program for arrays.
- Different versions exists:
  - **Top-Down**: Split unordered sequence, vs.
  - **Bottom-Up**: Start with sequences of length 1 and create increasingly longer ones.

Bottom-Up Mergesort

- Start with sequences of length 1; these are sorted by default.
- Merge pairs of sorted sequences to form single sorted sequences,
- until there is only one sequence left.

Bottom-Up Mergesort Example



Bottom-Up Mergesort Analysis

- Merging two sequences of length n/2 each can be done in O(n) if linked lists are used.
- In merge sort of n elements, we merge:
  - n/2 pairs of sequences of length 1
  - n/4 pairs of sequences of length 2
  - n/8 pairs of sequences of length 4
  - …
  - 1 pair of sequences of length n/2

Bottom-Up Mergesort Analysis

- At each "level" O(n) steps are used.
- There are log(n) levels.
- Therefore mergesort is O(n log(n)) worst case.

Bottom-Up Mergesort Analysis

- Let T(j) = steps at levels $\leq$ j
- Then
  - T(1) = c n, c some constant
  - T(j+1) = T(j) + c n
- T(log(n)) is the time to sort n elements = c n log(n)

Bottom-Up Mergesort in rex (1)

```
// first the initial list is transformed to a list of 1-element
// lists, then those lists are merged repeatedly

merge_sort(List) = merge_repeatedly( map((X) => [X], List ) );


// merge_repeatedly merges pairs in a list of lists until there
// is only one list left.

merge_repeatedly([A]) => A;          // only one list left

merge_repeatedly(Lists) =>           // more than one list left
  merge_repeatedly( merge_pairs(Lists) );
```

Bottom-Up Mergesort in rex (2)

```
// merge_pairs merges pairs of lists in a list until none is left.

merge_pairs([]) => [];                            // no more lists

merge_pairs([A]) => [A];                          // only one list

merge_pairs([A, B | L]) => [merge(A, B) | merge_pairs(L)];


// merge creates a single ordered list from two ordered lists

merge(L, []) => L;

merge([], M) => M;

merge([A | L], [B | M]) =>
  A <= B ? [A | merge(L, [B | M])] : [B | merge([A | L], M)];
```

Top-Down Mergesort Example

{ 7 4 3 2 1 5 0 6 }

split into two

{ 7 4 3 2 } { 1 5 0 6 }

recursively mergesort each half

{ 2 3 4 7 } { 0 1 5 6 }

merge the sorted halves

{ 0 1 2 3 4 5 6 7 }

Alternate ways to split

{ 7 4 3 2 1 5 0 6 }

split into two

{ 7 3 1 0 } { 4 2 5 6 }

Top-Down Mergesort Analysis

- $T_{merge}(n) = cn$ Time to merge two lists of length n/2
- $T_{split}(n) = dn$ Time to split a list of length n
- $T_{sort}(1) = 1;$
- $T_{sort}(n) = T_{split}(n) + 2\ T_{sort}(n/2) + T_{merge}(n)$

$= 2\ T_{sort}(n/2) + en$

where $e = c+d$

Top-Down Mergesort Analysis

- $T_{sort}(1) = 1;$
- $T_{sort}(n) = 2\ T_{sort}(n/2) + en$
- Substituting

$T(n) = 2\ T(n/2) + en$

$= 2\ (T(n/4) + en/2) + en$

$= 2\ (2(T(n/8) + en/4) + en/2) + en$

$= \ldots$

$= 2^{\log(n)} * 1 + \log(n) * en$

$= n + en \log(n)$

$\in O(n\log(n))$

Technique #2: Using data values to do selection

- Under this category, we have
  - bucket sort
  - radix sort
- We make non-general assumptions about the data:
  - The size of keys to be sorted is limited to integers with a fixed upper bound.

Bucket Sort

- Related to hashing
  - Both use indexing, which is O(1), to find "bucket"
- Suppose the set of keys to be sorted is known to be limited to a relatively small integer range, say {0, 1, …, R-1}.
- Create an array of size R of **lists**, each entry corresponding to a key value.
- Go through the data once, putting each element in the corresponding list.
- Concatenate the resulting lists.

Analysis of Bucket Sort

- Assume that the number of data elements is comparable to, or larger than, the number of buckets.
- Go through the data once, putting each element in the corresponding list.
  This is O(n).
- Concatenate the resulting lists.
  This is also O(n).
- Therefore we have O(n) overall.
- **Remember that bucket sorting makes special assumptions about the data.**

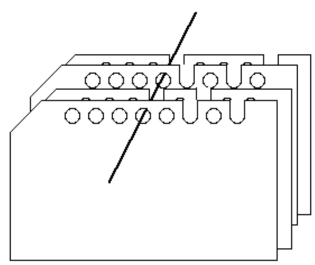
Radix Sort

- Like bucket sort, but with smaller arrays.
- Trades array size for multiple passes.
- Represent the range as radix b integers.
- Make $P = \log_b(R)$ passes.
- Sort on the least significant digit first, progressing toward the most.
- Re-collect the data after each pass and redistributed.

Analysis of Radix Sort

- Assuming a bounded range, the number of passes P is a fixed constant.
- Each pass uses O(n).
- Therefore we have O(n) overall.

Demo of Radix Sort



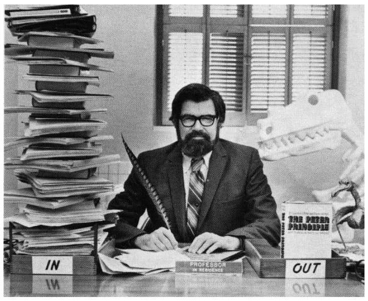
Technique #3: Use a Tree instead of Linear List

- For the same amount of data, a sufficiently balanced tree uses only O(log n) to traverse a chain rather than O(n) (as in naïve bubble, selection, or insertion sort).
- Heapsort is a form of sorting that uses trees.

Heapsort

- Loosely based on the "Peter Principle" (Dr. Laurence J. Peter)
- books by Laurence Peter

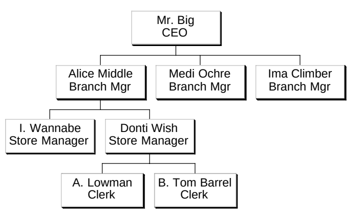
Laurence J. Peter



The Peter Principle

- "In a hierarchy, everyone rises to his/her level of *in*competence".
- [A person having reached this level is said to have the "final placement syndrome".]

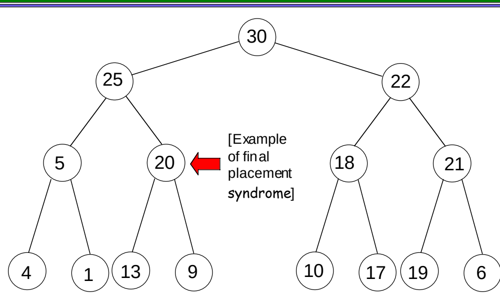
A Hierarchy



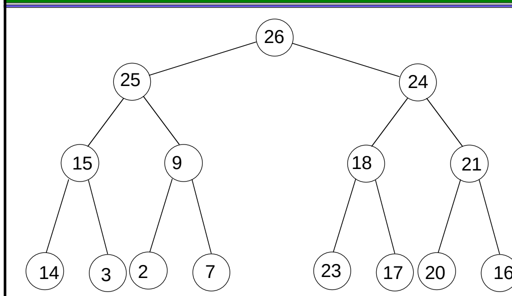
The Peter Principle applied to Sorting

- A **heap** is a binary tree structure in which, for each sub-tree, the root is ≥ all elements in that sub-tree.
- It suffices for *each* root to be ≥ its children.
- [This is one definition of heap used in CS. The other definition is the area in memory from which storage is allocated dynamically, e.g. when *new* is called.]

A Heap



A Heap?



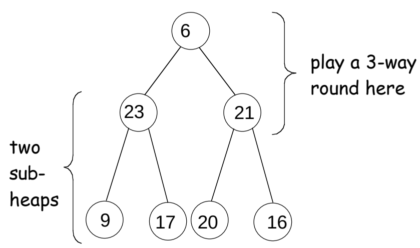
The Two Phases in Heapsort

- Phase I: Form the data into a heap
- Phase II: Transform the heap into a linear sequence

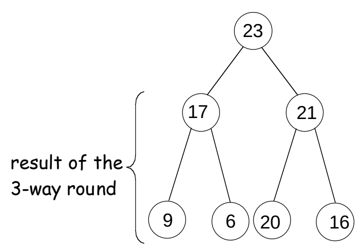
Phase I: Forming a Heap

- This is the actual Peter Principle.
- Start with the data distributed randomly in the tree.
- Form "sub-heaps" beginning at the leaves and working toward the root.
- Successively combine two sub-heaps with a common root into a single heap.

Combination as a Tournament



Combination as a Tournament



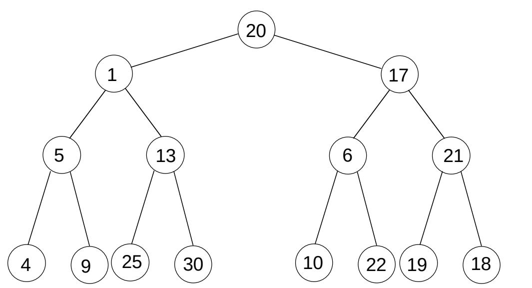
Analysis of forming a heap from two sub-heaps

- 3-way rounds are played from the parent downward to the leaves
- Only one path toward the leaves is followed, since other sub-trees along the path don't change
- A 3-way round uses O(1) steps (2 comparisons, possible exchange)
- The time to form a heap at level k from the leaves is therefore O(k).
- In the worst case, this is O(log(n)).

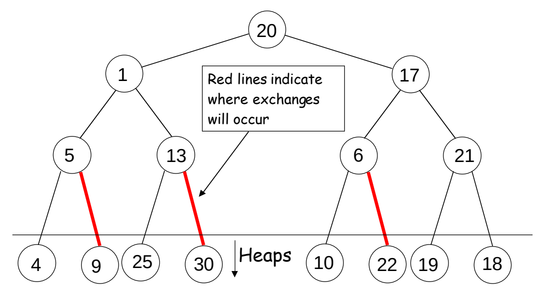
Iterating sub-heap formation from leaves to root

- The leaves are already heaps by themselves.
- We need to play the tournament (O(k) rounds at level k) n/2 times, corresponding to the non-leaves.
- Coarsely, Phase I is O(n log(n)) overall.
- Is this bound tight?

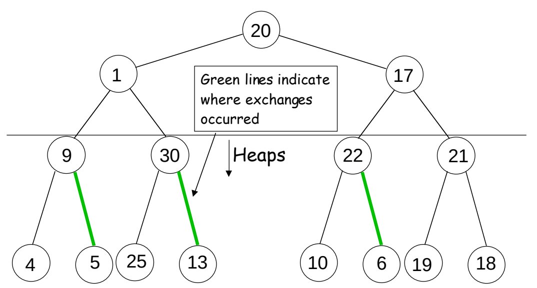
Illustrating Phase I



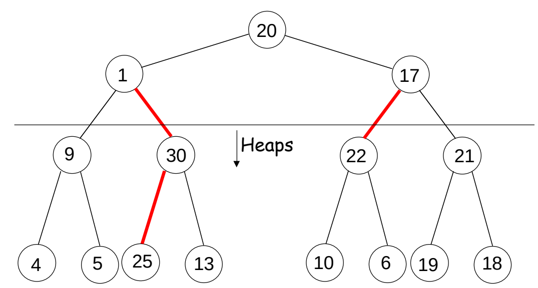
Illustrating Phase I



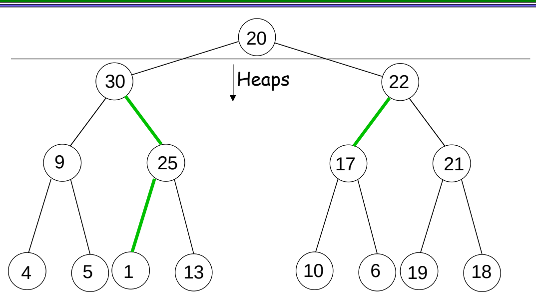
After Level 1 Heap Formation



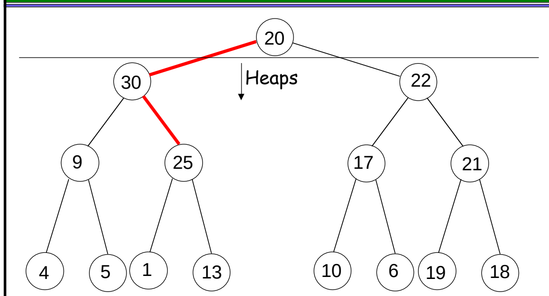
Level 2 Heap Formation



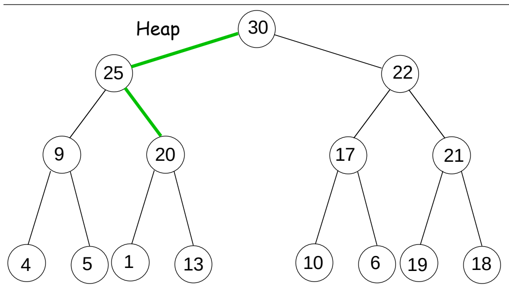
After Level 2 Heap Formation



Level 3 Heap Formation

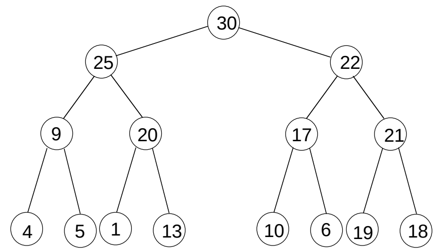


After Level 3 Heap Formation



Heapsort Phase II

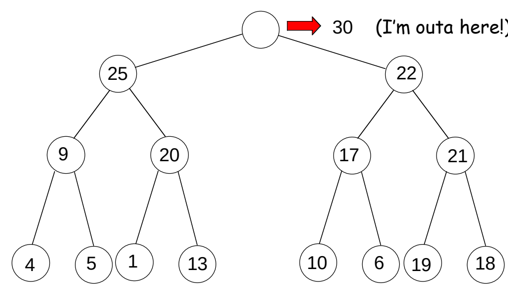
- Now that we have a heap, what do we do?



Heapsort Phase II

- One antidote for final-placement syndrome is that people **retire**.
- In our heap, we "retire" the maximum, which is guaranteed to be at the root.
- This leaves a hole that needs to be filled.

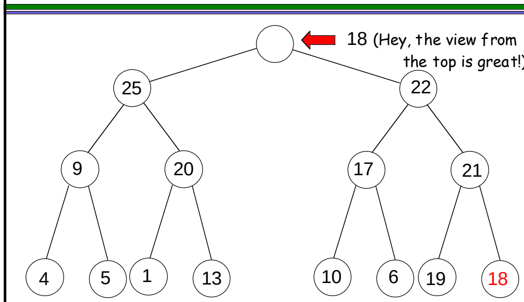
Retiring the maximum



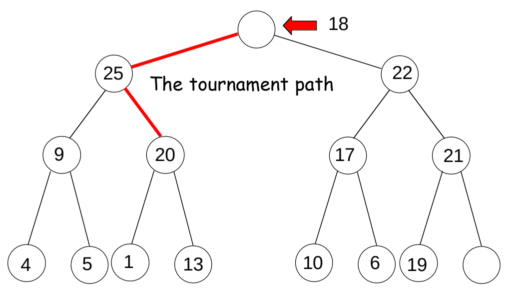
Filing the hole

- The way this happens is **different** from in a corporation.
- We pick the rightmost leaf, and tentatively place it in the hole at the root.
- Then we play the tournament from the root so that the new value reaches its proper level.

Filling the hole



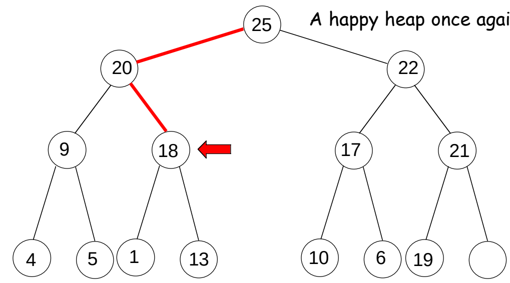
Filling the hole



30

Result of the tournament

A happy heap once again.

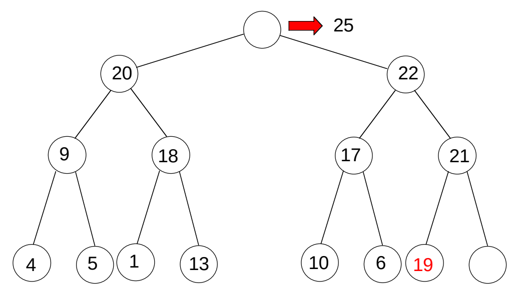


30

The cost of keeping the heap happy

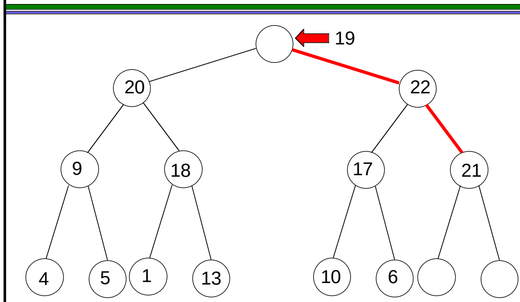
- The tournament path could be as long as the path from the root to a leaf.
- Along the path, a number of O(1) rounds were played.
- The cost for one post-retirement adjustment is therefore O(log(n)).
- Retiring all n elements in sequence gives us the sorted order.
- Overall then, we have O(n log(n)) for Phase II.

The Rest of Phase II illustrated



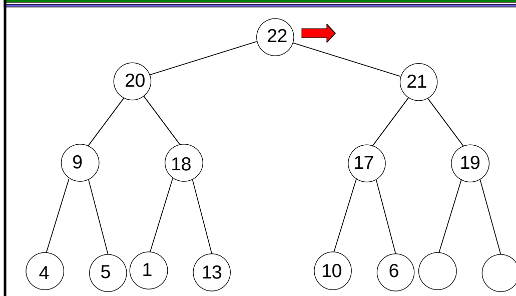
30

Phase II, step 2, continued



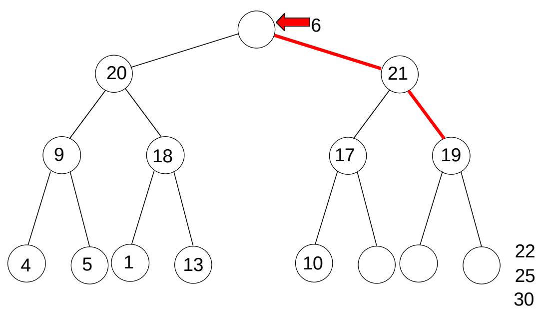
25
30

Phase II, step 3

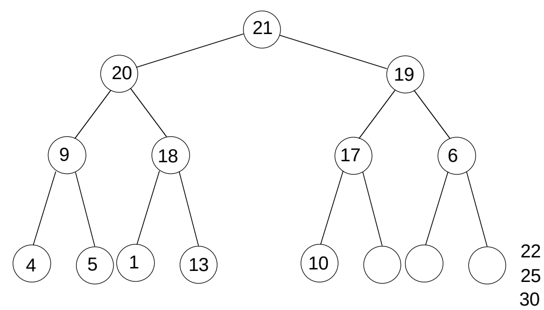


25
30

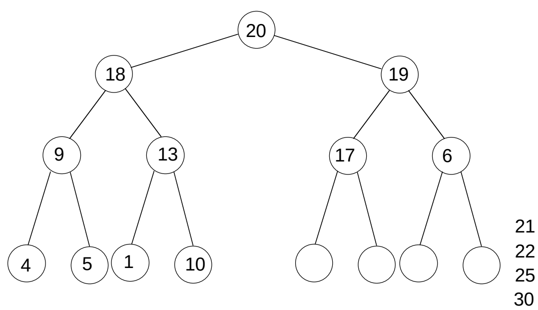
Phase II, step 3, continued



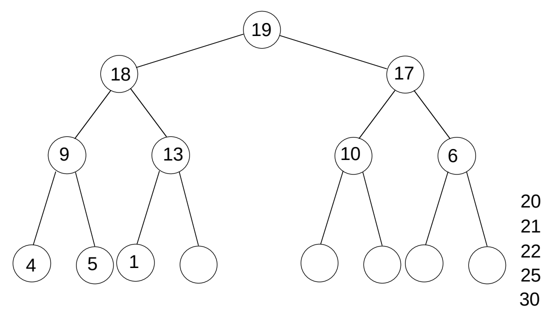
Phase II, step 4



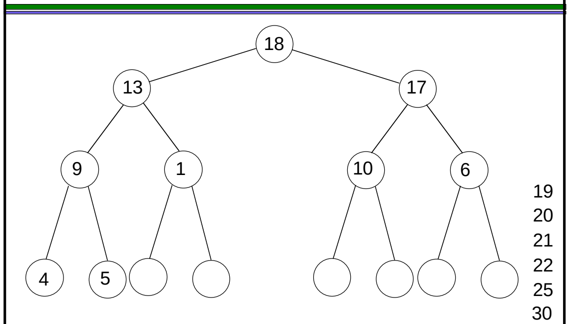
Phase II, step 5



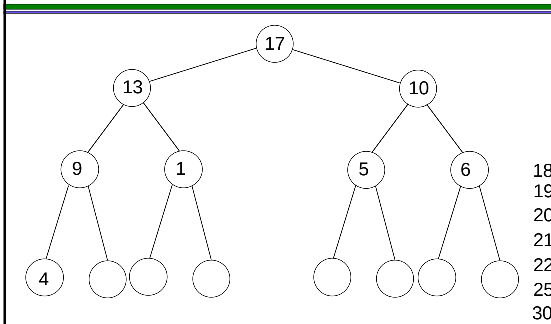
Phase II, step 6



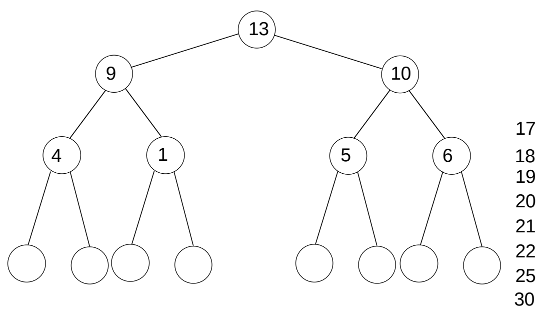
Phase II, step 7



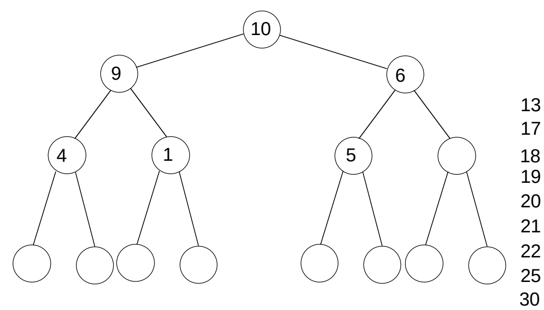
Phase II, step 8



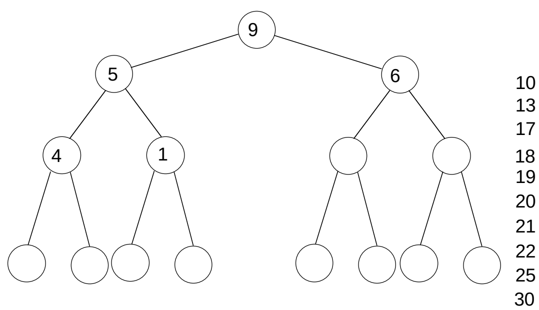
Phase II, step9



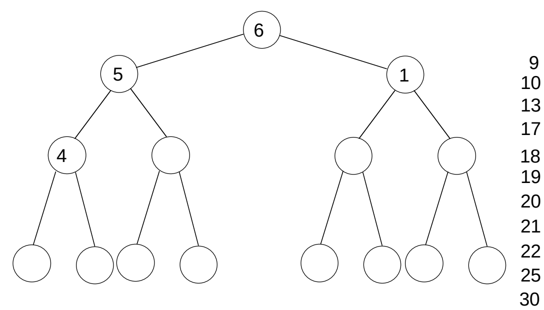
Phase II, step 10



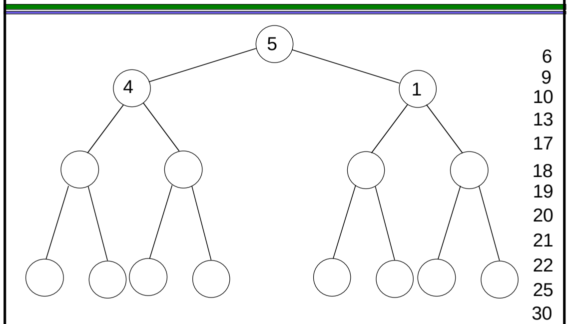
Phase II, step 11



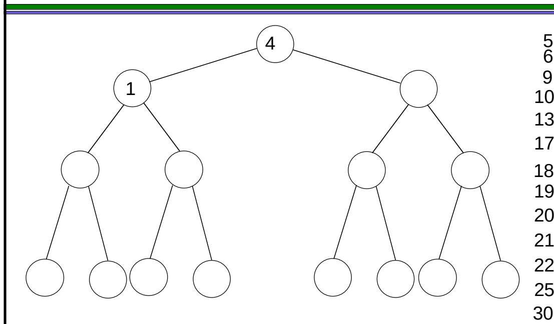
Phase II, step 12



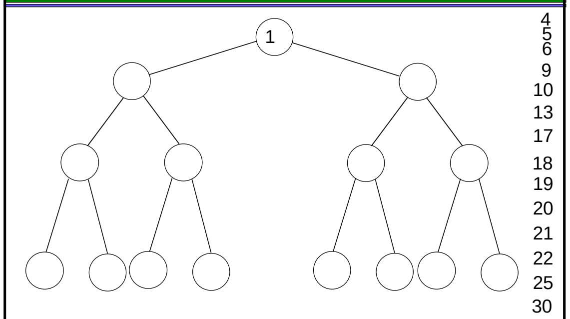
Phase II, step 13



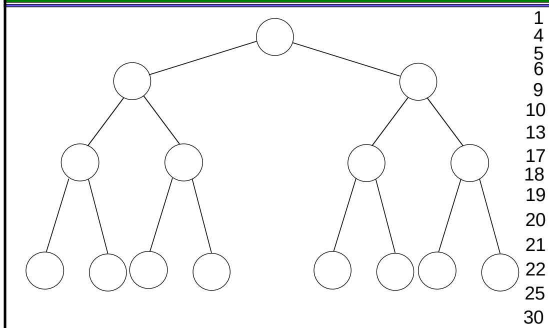
Phase II, step 14



Phase II, step 15



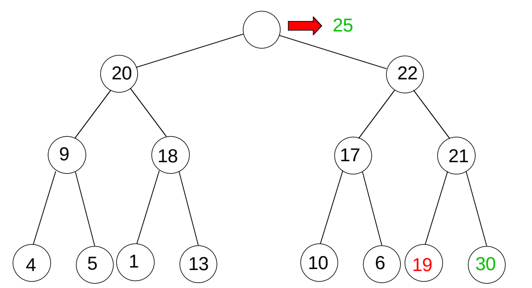
Conclusion of Phase II



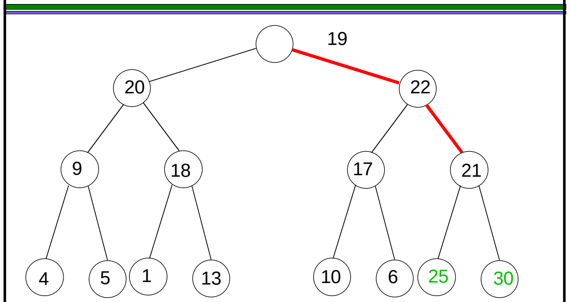
Where to put the retirees?

- We can put the retirees back in the tree, as long as we know not to play any more tournaments with them.
- There is an easy way to keep track of this, as we shall see.

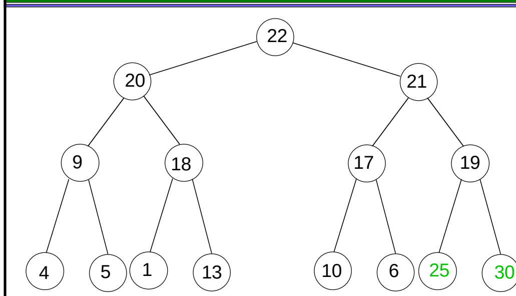
Phase II with Retiree Placement



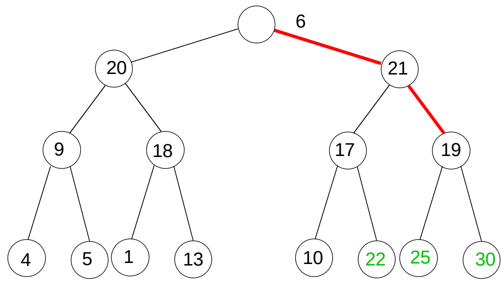
Phase II with Retiree Placement



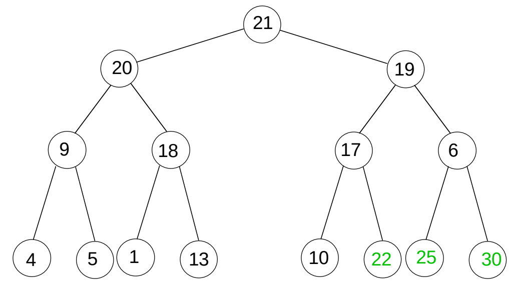
Phase II with Retiree Placement



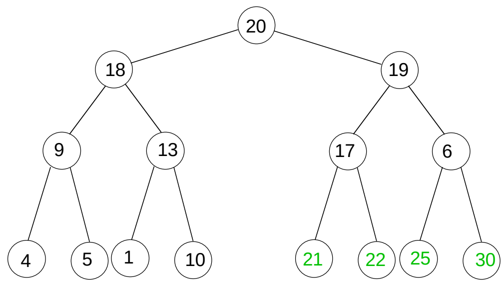
Phase II with Retiree Placement



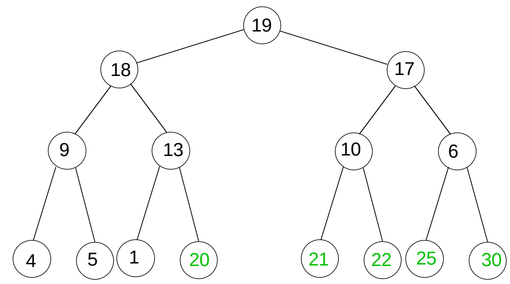
Phase II with Retiree Placement



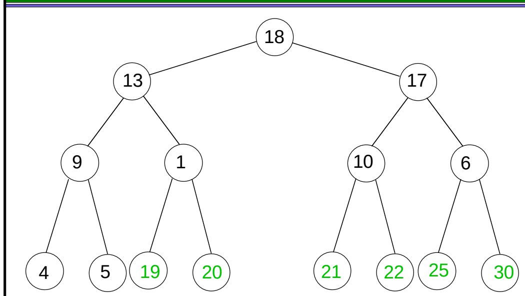
Phase II with Retiree Placement



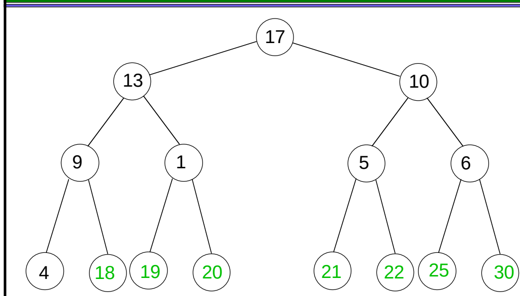
Phase II with Retiree Placement



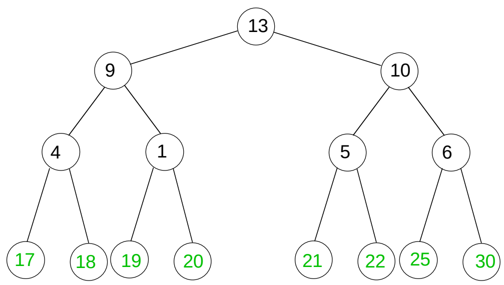
Phase II with Retiree Placement



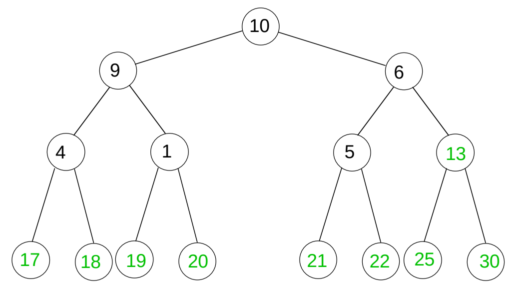
Phase II with Retiree Placement



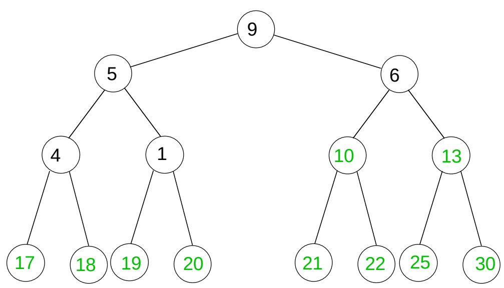
Phase II with Retiree Placement



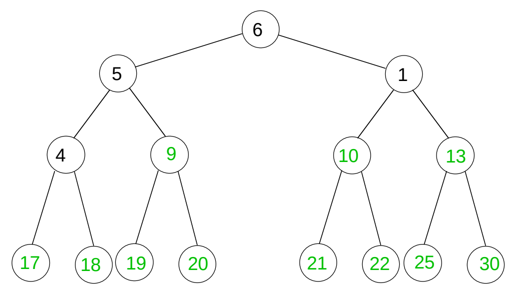
Phase II with Retiree Placement



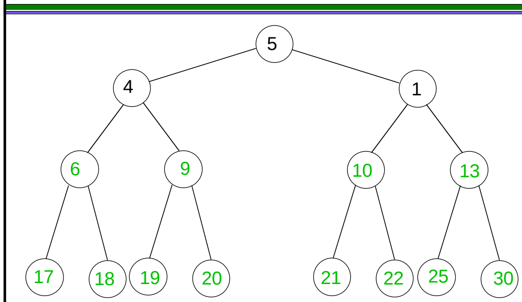
Phase II with Retiree Placement



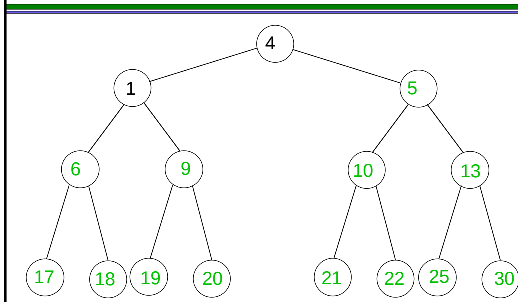
Phase II with Retiree Placement



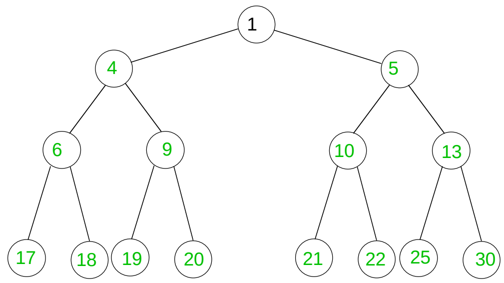
Phase II with Retiree Placement



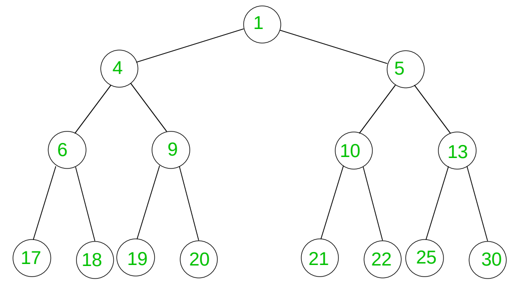
Phase II with Retiree Placement



Phase II with Retiree Placement



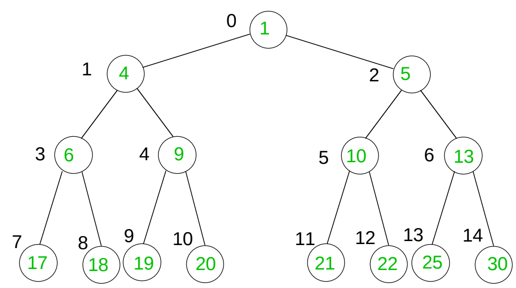
Phase II with Retiree Placement



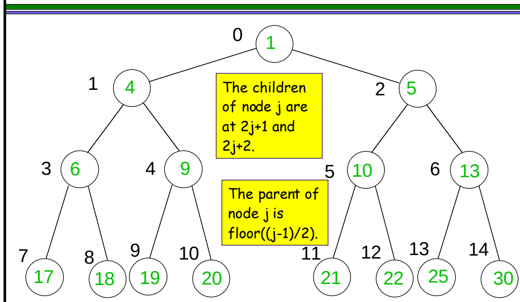
Node Numbering

- Note that the sorted sequence is readable from the nodes in breadth-first order.
- Further, we can maintain the entire tree as an array (with no explicit links), as shown next.

Array Node Numbering Permits Read-out of Sorted Data



Use as a Tree: Finding your parents/children



Heapsort Code (1)

```
heapsort(float array[], int N)
   {
   this.array = array;
   int Last = N-1;

   // phase 1: form heap
   for( int Top = Last/2; Top >= 0; Top-- )
     {
     adjust(Top, Last);
     }

   // phase 2: use heap to sort
   while( Last > 0 )
     {
     swap(0, Last);
     adjust(0, --Last);
     }
   }
```

Heapsort Code (2)

```
void adjust(int Top, int Last)
    {
    float TopVal = array[Top];                  // Set aside top of heap
    int Parent, Child;

    for( Parent = Top; ; Parent = Child )       // Iterate down through tree
      {
      Child = 2*Parent+1;                       // Child means left child

      if( Child > Last )
        break;                                  // Left child non-existent

      if( Child+1 <= Last                       // Right child exists
           && array[Child] < array[Child+1] )   // and right child is larger
        Child++;                                // Child is the larger child

      if( TopVal >= array[Child] )
        break;                                  // Location for TopVal found

      array[Parent] = array[Child];             // Move larger child up in tree
      }

    array[Parent] = TopVal;                     // Install TopVal in place
    }
```

Heapsort Summary

- O(n log(n)) steps
- In-place array implementation

Priority Queue

- A priority queue is a data repository that has two methods:
  - insert
  - removeMax (or removeMin, depending on the version)
- A **heap** is a natural implementation of a priority queue:
  - Both insertions and deletions are O(log n)

Priority Queue Applications

- A priority queue (with min removal) is often found in simulation applications.
- Events are time-stamped and put in a priority queue, which orders them by smallest time first.
- On a typical simulation cycle:
  - The event with the next timestamp is removed.
  - The event may cause the insertion of new events with later timestamps.

Lower-Bound for Sorting

- Sorting based on pairwise comparisons (which excludes radix and bucket sorts) requires a **minimum** of c n $\log_2(n)$ steps.
- Sorting methods that achieve this lower bound are called “optimal”.
- Heapsort and mergesort are the two optimal sorts we’ve studied.

Derivation of the Sorting Lower-Bound

- A sorting algorithm effectively establishes which of n! permutations the data are originally in and through a series of comparisons and exchanges permutes the data to a fixed order.
- This can be viewed as a tree with n! nodes, constructed with binary internal nodes for comparisons.

Derivation of the Sorting Lower-Bound

- The worst-case run time of the sorting algorithm is the height of a tree having has n! nodes.
- The minimum height of such a tree is $\log_2(n!)$.
- By Stirling's formula, $n! \sim (2 \pi n)^{1/2} (n/e)^{n}$, so $\log_2(n!)$ is $O(n \log(n))$.

Sorting Summary

- Minsort, insertion sort, bubble sort
  - easy to code
  - slow
  - avoid for large data sets
- Quicksort
  - fast on average
  - slow worst case
- Bucket sort / Radix sort
  - fastest asymptotic performance
  - special assumptions about data
- Heap sort
  - optimal performance
  - sorts arrays in-place
- Merge sort
  - optimal performance
  - sorts lists
  - more difficult for arrays