Different types of Sorting Techniques used in Data Structures

Sorting: Definition

Sorting: an operation that segregates items into groups according to specified criterion.

A = { 3 1 6 2 1 3 4 5 9 0 }

A = { 0 1 1 2 3 3 4 5 6 9 }

Sorting

- Sorting = ordering.
- Sorted = ordered based on a particular way.
- Generally, collections of data are presented in a sorted manner.
- Examples of Sorting:
  - Words in a dictionary are sorted (and case distinctions are ignored).
  - Files in a directory are often listed in sorted order.
  - The index of a book is sorted (and case distinctions are ignored).

Sorting: Cont'd

- Many banks provide statements that list checks in increasing order (by check number).
- In a newspaper, the calendar of events in a schedule is generally sorted by date.
- Musical compact disks in a record store are generally sorted by recording artist.
- Why?
  - Imagine finding the phone number of your friend in your mobile phone, but the phone book is not sorted.

Review of Complexity

Most of the primary sorting algorithms run on different space and time complexity.

Time Complexity is defined to be the time the computer takes to run a program (or algorithm in our case).

Space complexity is defined to be the amount of memory the computer needs to run a program.

Complexity (cont.)

Complexity in general, measures the algorithms efficiency in internal factors such as the time needed to run an algorithm.

External Factors (not related to complexity):

- Size of the input of the algorithm
- Speed of the Computer
- Quality of the Compiler

O(n), Ω(n), & Θ(n)

- An algorithm or function T(n) is O(f(n)) whenever T(n)'s rate of growth is less than or equal to f(n)'s rate.
- An algorithm or function T(n) is Ω(f(n)) whenever T(n)'s rate of growth is greater than or equal to f(n)'s rate.
- An algorithm or function T(n) is Θ(f(n)) if and only if the rate of growth of T(n) is equal to f(n).

Types of Sorting Algorithms

There are many, many different types of sorting algorithms, but the primary ones are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Shell Sort
- Radix Sort
- Swap Sort
- Heap Sort

CSE102_Anuradha_G

Bubble Sort: Idea

- Idea: bubble in water.
  - Bubble in water moves upward. Why?
- How?
  - When a bubble moves upward, the water from above will move downward to fill in the space left by the bubble.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

Bubblesort compares the numbers in pairs from left to right exchanging when necessary. Here the first number is compared to the second and as it is larger they are exchanged.

Now the next pair of numbers are compared. Again the 9 is the larger and so this pair is also exchanged.

In the third comparison, the 9 is not larger than the 12 so no exchange is made. We move on to compare the next pair without any change to the list.

The 12 is larger than the 11 so they are exchanged.

The twelve is greater than the 9 so they are exchanged

The end of the list has been reached so this is the end of the first pass. The twelve at the end of the list must be largest number in the list and so is now in the correct position. We now start a new pass from left to right.

The 12 is greater than the 7 so they are exchanged.

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Notice that this time we do not have to compare the last two numbers as we know the 12 is in position. This pass therefore only requires 6 comparisons.

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

This time the 11 and 12 are in position. This pass therefore only requires 5 comparisons.

CSE102_Anuradha_G

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass 2, 6, 3, 7, 9, 9, 11, 12

Each pass requires fewer comparisons. This time only 4 are needed.

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass 2, 6, 3, 7, 9, 9, 11, 12

Fifth Pass 2, 3, 6, 7, 9, 9, 11, 12

The list is now sorted but the algorithm does not know this until it completes a pass with no exchanges.

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass 2, 6, 3, 7, 9, 9, 11, 12

Fifth Pa

This pass no exchanges are made so the algorithm knows the list is sorted. It can therefore save time by not doing the final pass. With other lists this check could save much more work.

Sixth Pass 2, 3, 6, 7, 9, 9, 11, 12

Bubble Sort Example

Quiz Time

1. Which number is definitely in its correct position at the end of the first pass?

Answer: The last number must be the largest.

2. How does the number of comparisons required change as the pass number increases?

Answer: Each pass requires one fewer comparison than the last.

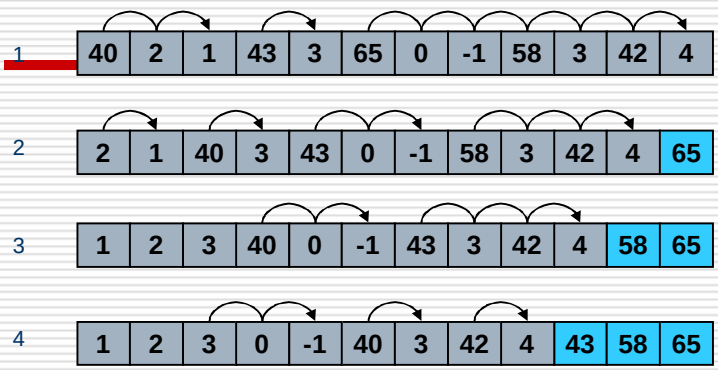
3. How does the algorithm know when the list is sorted?

Answer: When a pass with no exchanges occurs.

4. What is the maximum number of comparisons required for a list of 10 numbers?

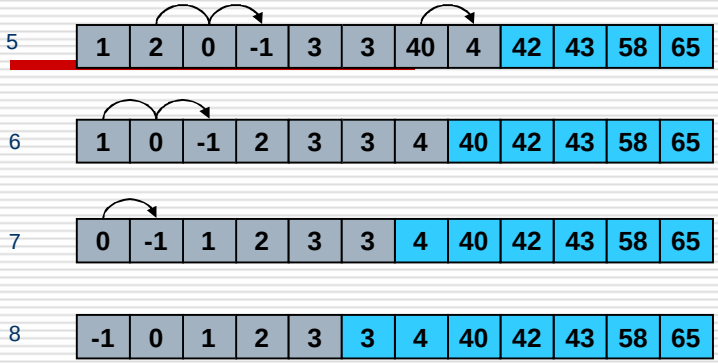
Answer: 9 comparisons, then 8, 7, 6, 5, 4, 3, 2, 1 so total 45

Bubble Sort: Example



- Notice that at least one element will be in the correct position each iteration.

Bubble Sort: Example




```
for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1]) /* For decreasing order
use < */
      {
        swap     = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }
```

Bubble Sort: Analysis

- Running time:
  - Worst case: $O(N^2)$
  - Best case: O(N)
- Variant:
  - bi-directional bubble sort
    - original bubble sort: only works to one direction
    - bi-directional bubble sort: works back and forth.

CSE102_Anuradha_G

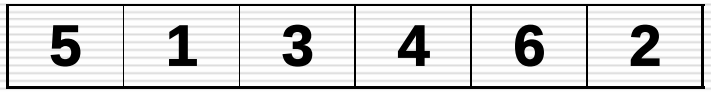
Selection Sort: Idea

1. We have two group of items:
   - sorted group, and
   - unsorted group
2. Initially, all items are in the unsorted group. The sorted group is empty.
   - We assume that items in the unsorted group unsorted.
   - We have to keep items in the sorted group sorted.

Selection Sort: Cont'd

1. Select the "best" (eg. smallest) item from the unsorted group, then put the "best" item at the end of the sorted group.
2. Repeat the process until the unsorted group becomes empty.

Selection Sort

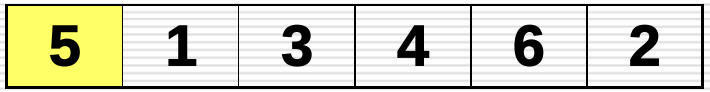


Comparison

Data Movement

Sorted

Selection Sort



Comparison

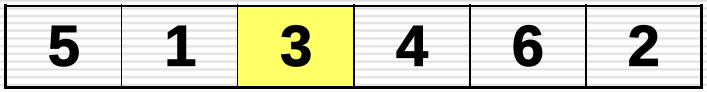
Data Movement

Sorted

Selection Sort

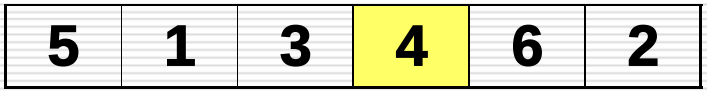
Comparison
Data Movement
Sorted

Selection Sort



Comparison
Data Movement
Sorted

Selection Sort



Comparison
Data Movement
Sorted

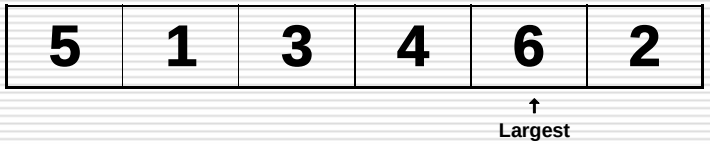
Selection Sort

Comparison
Data Movement
Sorted

Selection Sort

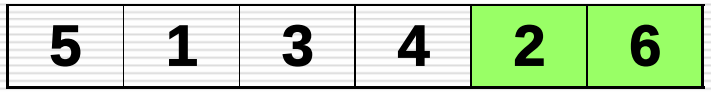
Comparison
Data Movement
Sorted

Selection Sort



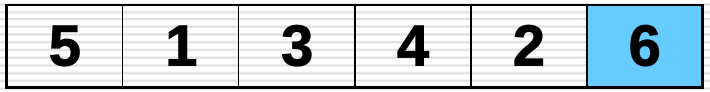
Comparison
Data Movement
Sorted

Selection Sort



Comparison
Data Movement
Sorted

Selection Sort

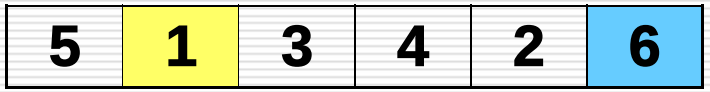


Comparison
Data Movement
Sorted

Selection Sort

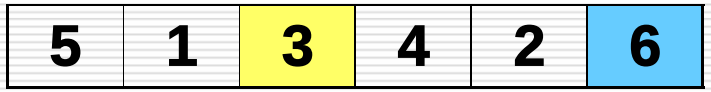
Comparison
Data Movement
Sorted

Selection Sort



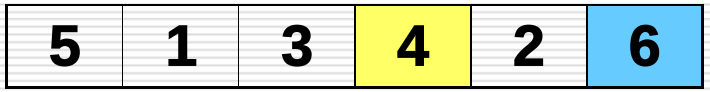
Comparison
Data Movement
Sorted

Selection Sort



Comparison
Data Movement
Sorted

Selection Sort



Comparison
Data Movement
Sorted

Selection Sort

Comparison

Data Movement

Sorted

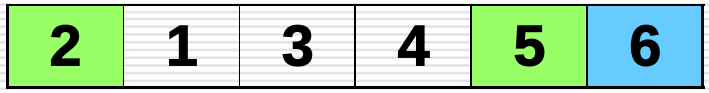
Selection Sort

Comparison

Data Movement

Sorted

Selection Sort

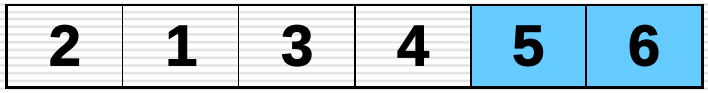


Comparison

Data Movement

Sorted

Selection Sort



Comparison

Data Movement

Sorted

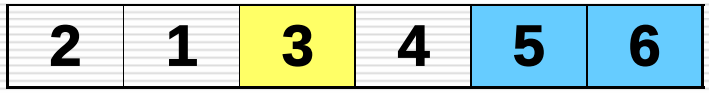
Selection Sort

Comparison
Data Movement
Sorted

Selection Sort

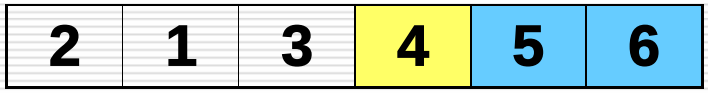
Comparison
Data Movement
Sorted

Selection Sort



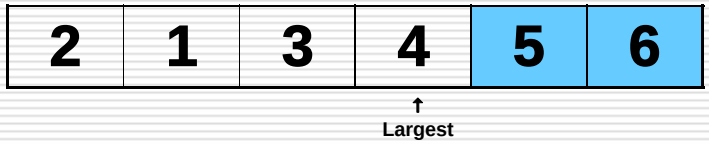
Comparison
Data Movement
Sorted

Selection Sort



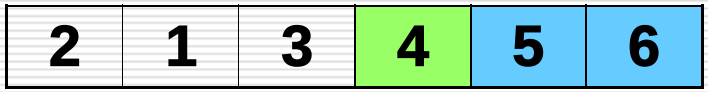
Comparison
Data Movement
Sorted

Selection Sort



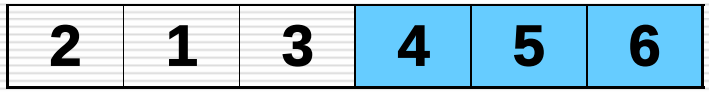
Comparison
Data Movement
Sorted

Selection Sort



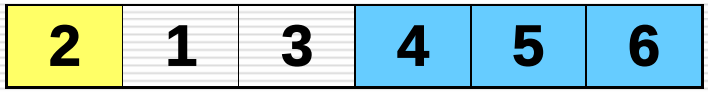
Comparison
Data Movement
Sorted

Selection Sort



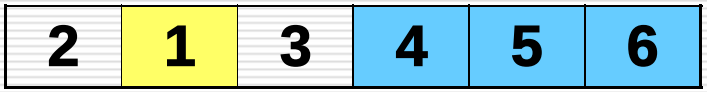
Comparison
Data Movement
Sorted

Selection Sort



Comparison
Data Movement
Sorted

Selection Sort

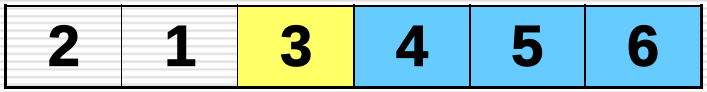


Comparison

Data Movement

Sorted

Selection Sort

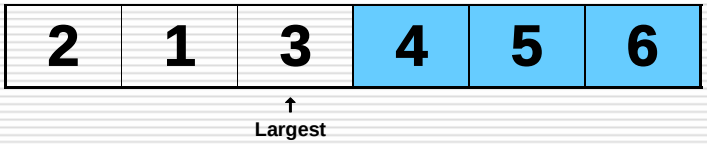


Comparison

Data Movement

Sorted

Selection Sort



Comparison

Data Movement

Sorted

Selection Sort

Comparison

Data Movement

Sorted

Selection Sort

Comparison

Data Movement

Sorted

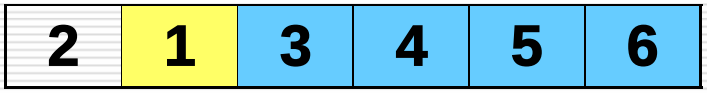
Selection Sort

Comparison

Data Movement

Sorted

Selection Sort

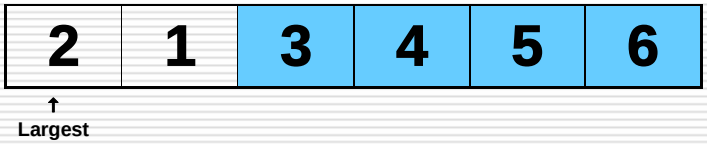


Comparison

Data Movement

Sorted

Selection Sort



Comparison

Data Movement

Sorted

CSE102_Anuradha_G

Selection Sort

Comparison

Data Movement

Sorted

Selection Sort

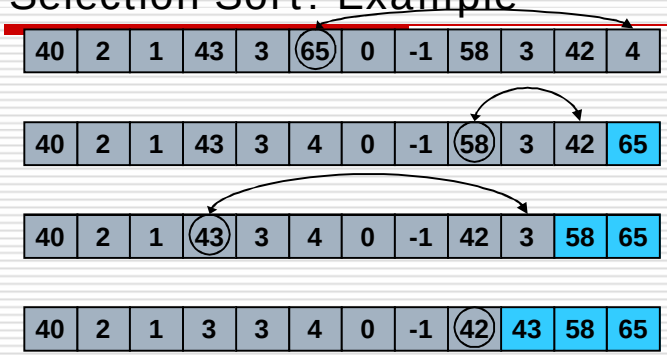
DONE!

Comparison

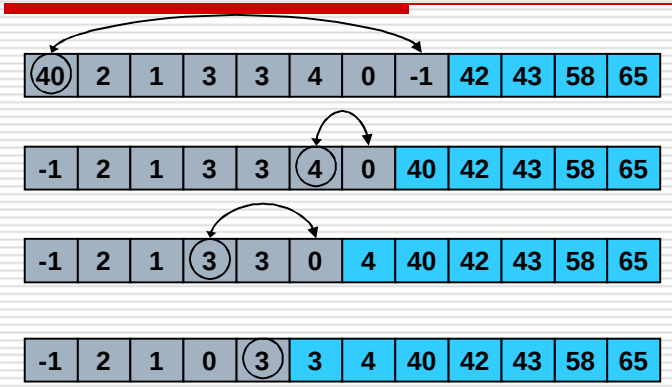
Data Movement

Sorted

Selection Sort: Example

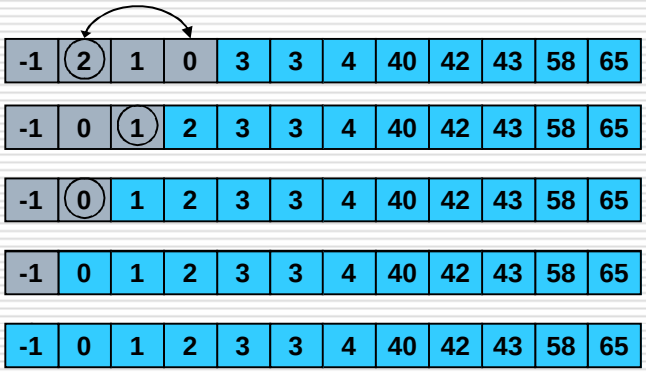


Selection Sort: Example



CSE102_Anuradha_G

Selection Sort: Example




```
for ( c = 0 ; c < ( n - 1 ) ; c++ )
  {
    position = c;

    for ( d = c + 1 ; d < n ; d++ )
    {
      if ( array[position] > array[d] )
        position = d;
    }
    if ( position != c )
    {
      swap = array[c];
      array[c] = array[position];
      array[position] = swap;
    }
  }
```

Selection Sort: Analysis

- Running time:
  - Worst case: $O(N^2)$
  - Best case: $O(N^2)$

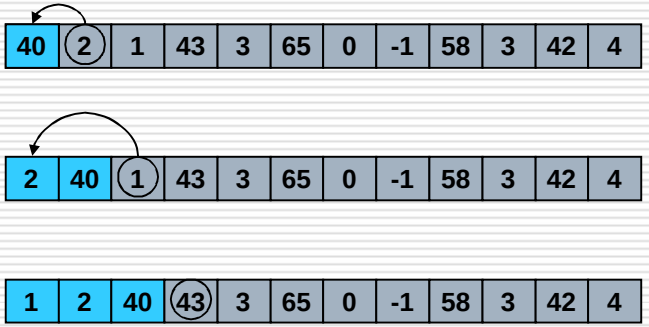
Insertion Sort: Idea

- Idea: sorting cards.
  - **8 | 5 9 2 6 3**
  - **5 8 | 9 2 6 3**
  - **5 8 9 | 2 6 3**
  - **2 5 8 9 | 6 3**
  - **2 5 6 8 9 | 3**
  - **2 3 5 6 8 9 |**

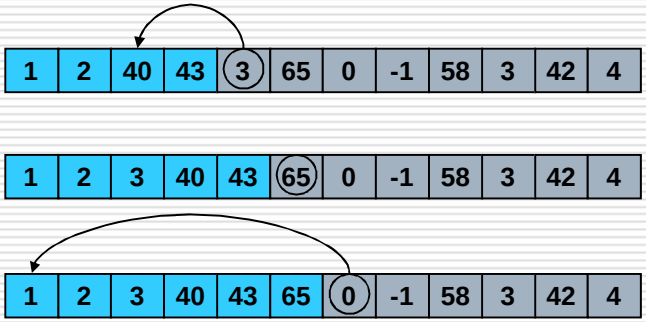
Insertion Sort: Idea

1. We have two group of items:
   - sorted group, and
   - unsorted group
2. Initially, all items in the unsorted group and the sorted group is empty.
   - We assume that items in the unsorted group unsorted.
   - We have to keep items in the sorted group sorted.
3. Pick any item from, then insert the item at the right position in the sorted group to maintain sorted property.
4. Repeat the process until the unsorted group becomes empty.

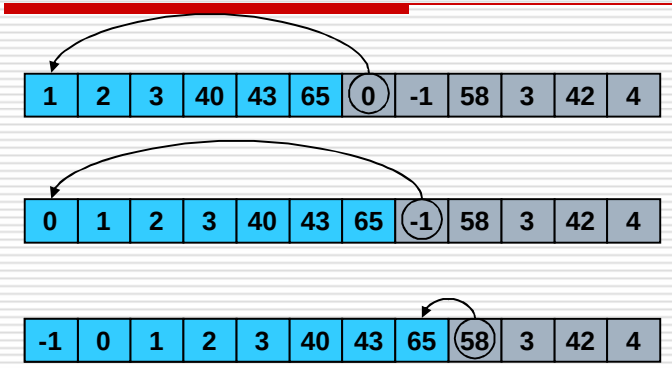
Insertion Sort: Example



Insertion Sort: Example

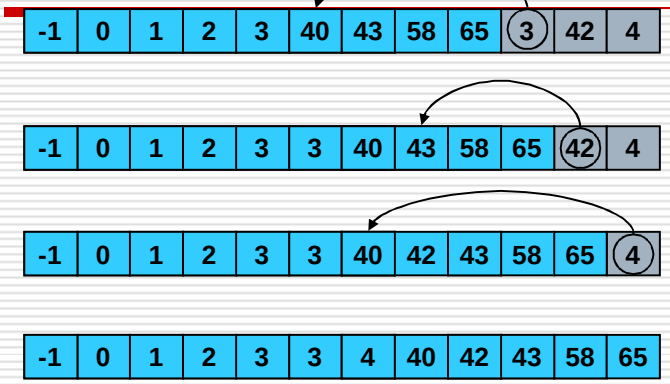


Insertion Sort: Example



CSE102_Anuradha_G

Insertion Sort: Example




```
for (c = 1 ; c <= n - 1; c++)
{
   d = c;
    while ( d > 0 && array[d] < array[d-1])
{
     t          = array[d];
     array[d]   = array[d-1];
     array[d-1] = t;
      d--;
   }
 }
```

Insertion Sort: Analysis

- Running time analysis:
  - Worst case: $O(N^2)$
  - Best case: $O(N)$

A Lower Bound

- Bubble Sort, Selection Sort, Insertion Sort all have worst case of $O(N^2)$.
- Turns out, for any algorithm that exchanges adjacent items, this is the best worst case: $\Omega(N^2)$
- In other words, this is a lower bound!

CSE102_Anuradha_G

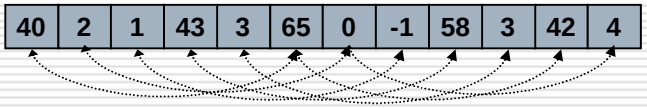
Shell Sort: Idea

Donald Shell (1959): Exchange items that are far apart!

Original:

40	2	1	43	3	65	0	-1	58	3	42	4

5-sort: Sort items with distance 5 element:



Shell Sort: Example

Original:

40	2	1	43	3	65	0	-1	58	3	42	4

After 5-sort:

40	0	-1	43	3	42	2	1	58	3	65	4

After 3-sort:

2	0	-1	3	1	4	40	3	42	43	65	58

After 1-sort:

-1	0	1	2	3	3	4	40	42	43	58	65

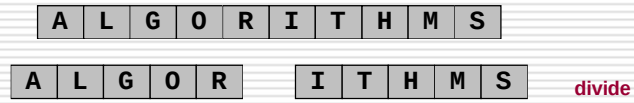
```
for(i=num/2; i>0; i=i/2)
 {
  for(j=i; j<num; j++)
  {
   for(k=j-i; k>=0; k=k-i)
   {
     if(arr[k+i]>=arr[k])
        break;
     else
     {
        tmp=arr[k];
        arr[k]=arr[k+i];
        arr[k+i]=tmp;
     }
   }
  }
 }
```

Shell Sort: Gap Values

- Gap: the distance between items being sorted.
- As we progress, the gap decreases. Shell Sort is also called Diminishing Gap Sort.
- Shell proposed starting gap of N/2, halving at each step.
- There are many ways of choosing the next gap.

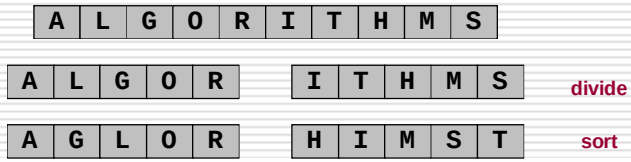
Mergesort

- Mergesort (divide-and-conquer)
  - Divide array into two halves.



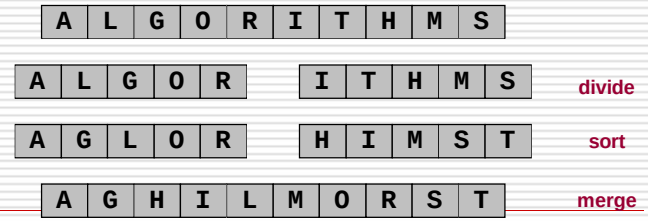
Mergesort

- Mergesort (divide-and-conquer)
  - Divide array into two halves.
  - Recursively sort each half.



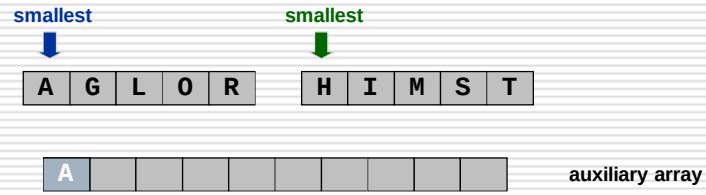
Mergesort

- Mergesort (divide-and-conquer)
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.



Merging

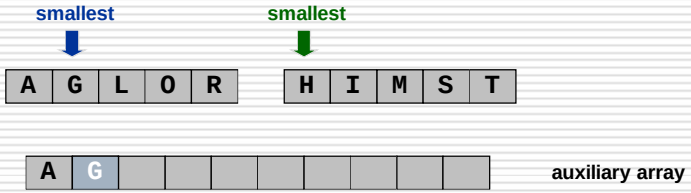
- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



CSE102_Anuradha_G

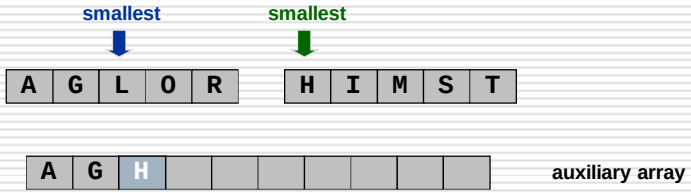
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



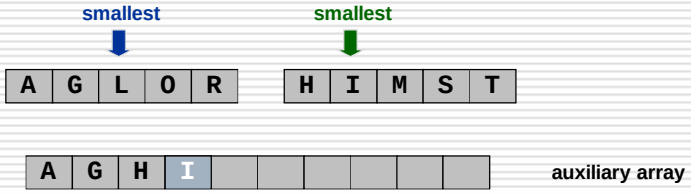
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



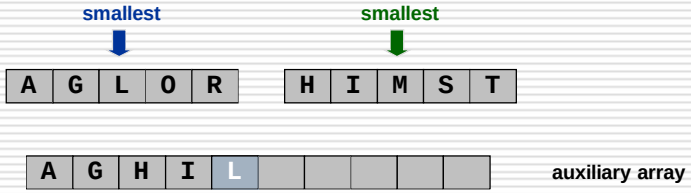
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Merging

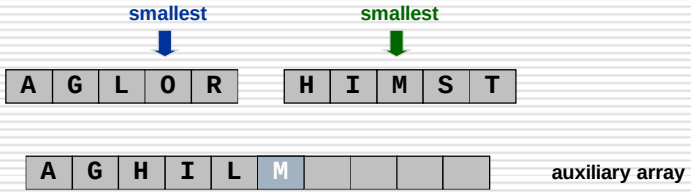
- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



CSE102_Anuradha_G

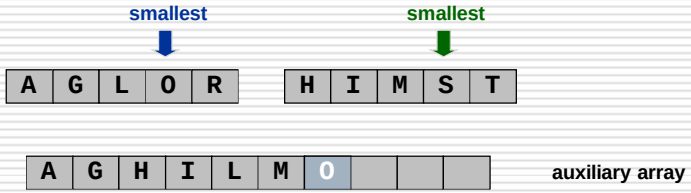
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



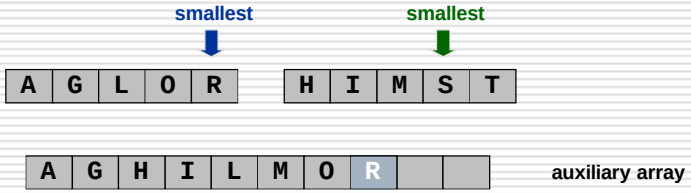
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



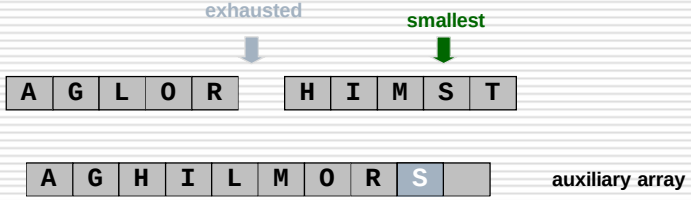
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Merging

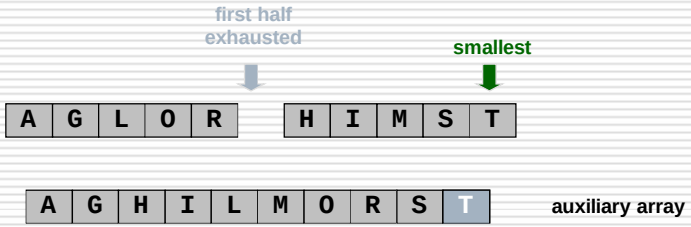
- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



CSE102_Anuradha_G

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.




```
void part(int arr[],int min,int max)
{
 int mid;
 if(min<max)
 {
   mid=(min+max)/2;
   part(arr,min,mid);
   part(arr,mid+1,max);
   merge(arr,min,mid,max);
 }
}
```

```
void merge(int arr[],int min,int
mid,int max)
{
 int tmp[30];
 int i,j,k,m;
 j=min;
 m=mid+1;
 for(i=min; j<=mid && m<=max
; i++)
 {
   if(arr[j]<=arr[m])
   {
      tmp[i]=arr[j];
      j++;
   }
   else
   {
      tmp[i]=arr[m];
      m++;
   }
 }
```

```
if(j>mid)
 {
   for(k=m; k<=max; k++)
   {
      tmp[i]=arr[k];
      i++;
   }
 }
 else
 {
   for(k=j; k<=mid; k++)
   {
      tmp[i]=arr[k];
      i++;
   }
 }
 for(k=min; k<=max; k++)
   arr[k]=tmp[k];
}
```

Notes on Quicksort

- Quicksort is more widely used than any other sort.
- Quicksort is well-studied, not difficult to implement, works well on a variety of data, and consumes fewer resources that other sorts in nearly all situations.
- Quicksort is O(n*log n) time, and O(log n) additional space due to recursion.

Quicksort Algorithm

- Quicksort is a divide-and-conquer method for sorting. It works by partitioning an array into parts, then sorting each part independently.
- The crux of the problem is how to partition the array such that the following conditions are true:
  - There is some element, a[i], where a[i] is in its final position.
  - For all l < i, a[l] < a[i].
  - For all i < r, a[i] < a[r].

Quicksort Algorithm (cont)

- As is typical with a recursive program, once you figure out how to divide your problem into smaller subproblems, the implementation is amazingly simple.

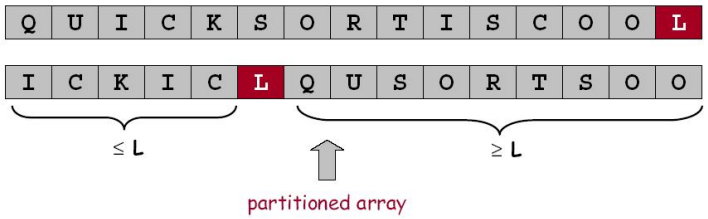
```
int partition(Item a[], int l, int r);
void quicksort(Item a[], int l, int r)
  { int i;
    if (r <= l) return;
    i = partition(a, l, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
```

Quicksort

Quicksort.
- Partition array so that:
  - some partitioning element a[m] is in its final position
  - no larger element to the left of m
  - no smaller element to the right of m

C. A. R. Hoare

partitioning element

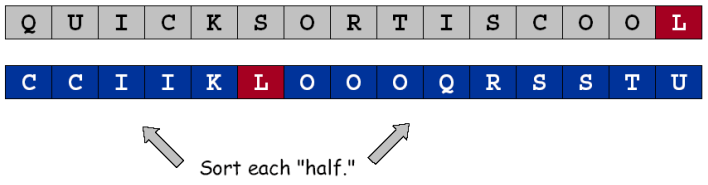


21

Quicksort

Quicksort.
- Partition array so that:
  - some partitioning element a[m] is in its final position
  - no larger element to the left of m
  - no smaller element to the right of m
- Sort each "half" recursively.

partitioning element



22

CSE102_Anuradha_G

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me swap me

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

partition element | unpartitioned | left | partitioned | right

CSE102_Anuradha_G

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

swap me　　swap me

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

partition element　unpartitioned　left
partitioned　right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element　unpartitioned　left
partitioned　right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element　unpartitioned　left
partitioned　right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - exchange
  - repeat until pointers cross

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element　unpartitioned　left
partitioned　right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element · unpartitioned · left · partitioned · right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

swap me

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element · unpartitioned · left · partitioned · right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

swap me

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element · unpartitioned · left · partitioned · right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

swap me

C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

partition element · unpartitioned · left · partitioned · right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

swap me

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

pointers cross

swap with partitioning element

partition element | unpartitioned | left | partitioned | right

Partitioning in Quicksort

- How do we partition the array efficiently?
  - choose partition element to be rightmost element
  - scan from left for larger element
  - scan from right for smaller element
  - Exchange and repeat until pointers cross

partition is complete

partition element | unpartitioned | left | partitioned | right

Quicksort Demo

- Quicksort illustrates the operation of the basic algorithm. When the array is partitioned, one element is in place on the diagonal, the left subarray has its upper corner at that element, and the right subarray has its lower corner at that element. The original file is divided into two smaller parts that are sorted independently. The left subarray is always sorted first, so the sorted result emerges as a line of black dots moving right and up the diagonal.

Why study Heapsort?

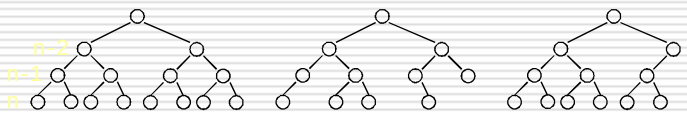
- It is a well-known, traditional sorting algorithm you will be expected to know
- Heapsort is *always* O(n log n)
  - Quicksort is usually O(n log n) but in the worst case slows to $O(n^2)$
  - Quicksort is generally faster, but Heapsort is better in time-critical applications

What is a "heap"?

- Definitions of heap:
  1. A large area of memory from which the programmer can allocate blocks as needed, and deallocate them (or allow them to be garbage collected) when no longer needed
  2. A balanced, left-justified binary tree in which no node has a value greater than the value in its parent
- Heapsort uses the second definition

Balanced binary trees

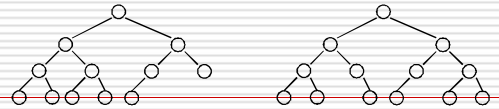
- Recall:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
- A binary tree of depth n is balanced if all the nodes at depths 0 through $n-2$ have two children



Balanced Balanced Not balanced

Left-justified binary trees

- A balanced binary tree is left-justified if:
  - all the leaves are at the same depth, or
  - all the leaves at depth $n+1$ are to the left of all the nodes at depth n

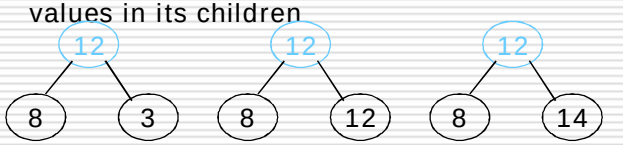


Left-justified Not left-justified

CSE102_Anuradha_G

The heap property

- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property

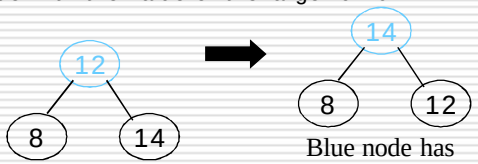
Blue node has heap property

Blue node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

siftUp

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



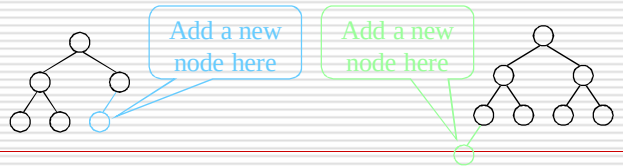
Blue node does not have heap property

Blue node has heap property

- This is sometimes called sifting up
- Notice that the child may have *lost* the heap property

Constructing a heap I

- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:

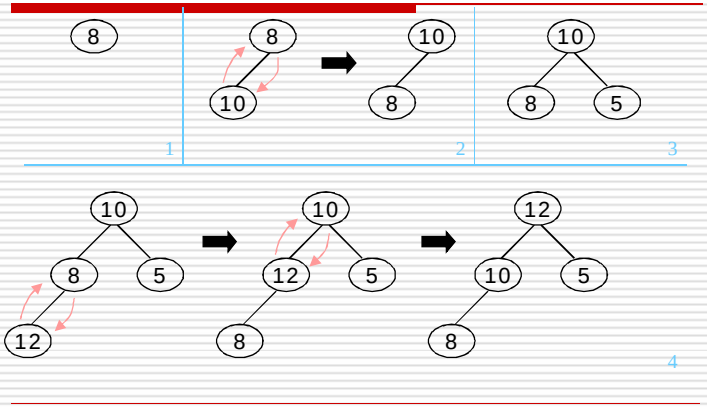


Constructing a heap II

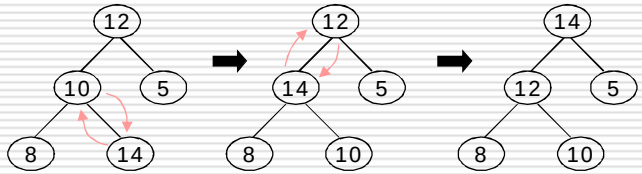
- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We repeat the sifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

CSE102_Anuradha_G

Constructing a heap III



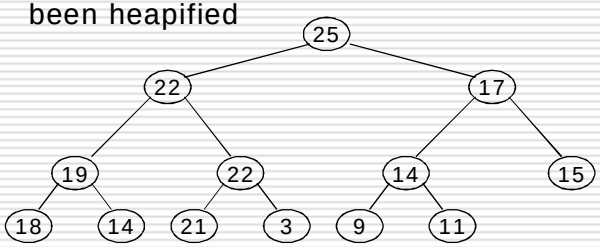
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

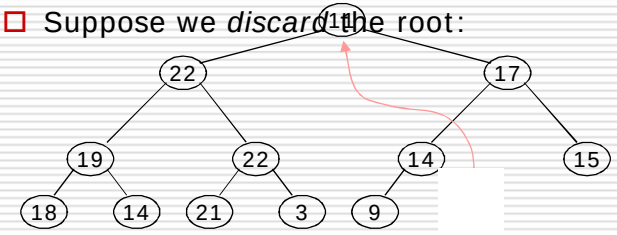
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root

- Notice that the largest number is now in the root
- Suppose we *discard* the root:

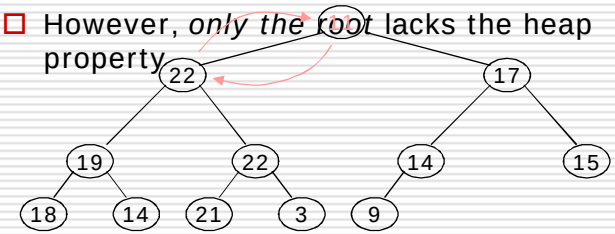


- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

CSE102_Anuradha_G

The reHeap method I

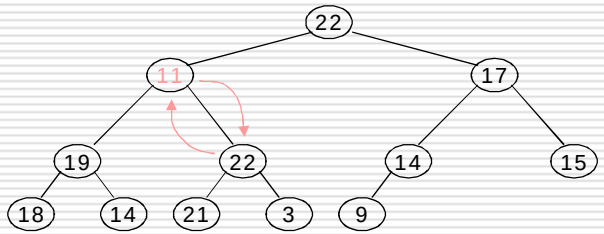
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



- We can siftUp() the root
- After doing this, one and only one of its children may have lost the heap property

The reHeap method II

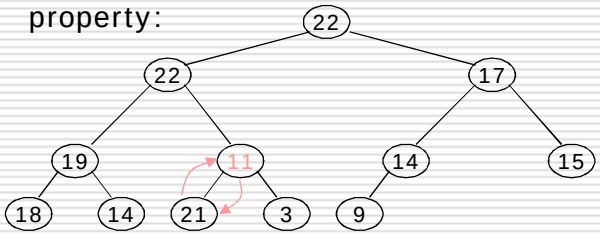
- Now the left child of the root (still the number 11) lacks the heap property



- We can siftUp() this node
- After doing this, one and only one of its children may have lost the heap property

The reHeap method III

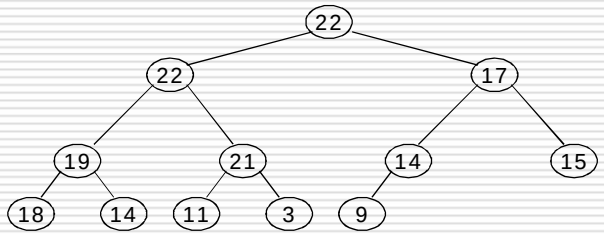
- Now the right child of the left child of the root (still the number 11) lacks the heap property:



- We can siftUp() this node
- After doing this, one and only one of its children may have lost the heap property —but it doesn't, because it's a leaf

The reHeap method IV

- Our tree is once again a heap, because every node in it has the heap property



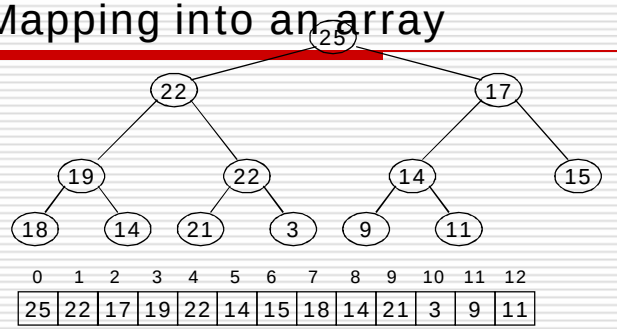
- Once again, the largest (or *a* largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

Sorting

- What do heaps have to do with sorting an array?
- Here's the neat part:
  - Because the binary tree is *balanced* and *left justified,* it can be represented as an array
  - All our operations on binary trees can be represented as operations on *arrays*
  - To sort:

```
heapify the array;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

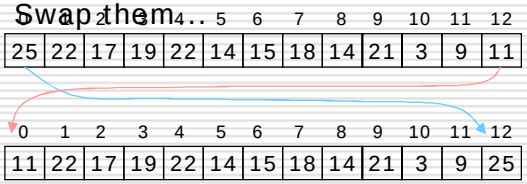
Mapping into an array



- Notice:
  - The left child of index i is at index $2*i+1$
  - The right child of index i is at index $2*i+2$
  - Example: the children of node 3 (19) are 7 (18) and 8 (14)

Removing and replacing the root

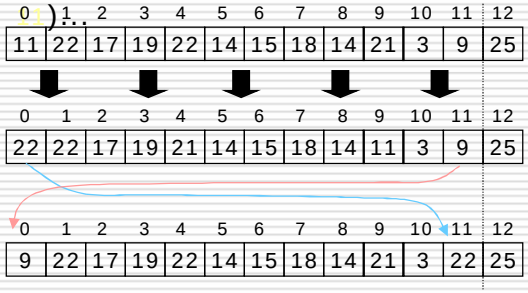
- The "root" is the first element in the array
- The "rightmost node at the deepest level" is the last element
- Swap them...



- ...And pretend that the last element in the array no longer exists—that is, the "last index" is 11 (9)

Reheap and repeat

- Reheap the root node (index 0, containing 11)...



- ...And again, remove and replace the root node
- Remember, though, that the "last" array index is changed
- Repeat until the last becomes first, and the array is sorted!

CSE102_Anuradha_G

Analysis I

- Here's how the algorithm starts:

```
heapify the array;
```

- Heapifying the array: we add each of n nodes
  - Each node has to be sifted up, possibly as far as the root
    - Since the binary tree is perfectly balanced, sifting up a single node takes O(log n) time
  - Since we do this n times, heapifying takes n*O(log n) time, that is, O(n log n) time

Analysis II

- Here's the rest of the algorithm:

```
while the array isn't empty {
   remove and replace the root;
   reheap the new root node;
}
```

- We do the while loop n times (actually, n-1 times), because we remove one of the n nodes each time
- Removing and replacing the root takes O(1) time
- Therefore, the total time is n times however long it takes the reheap method

Analysis III

- To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is O(log n) long
  - And we only do O(1) operations at each node
  - Therefore, reheaping takes O(log n) times
- Since we reheap inside a while loop that we do n times, the total time for the while loop is n*O(log n), or O(n log n)

Analysis IV

- Here's the algorithm again:

```
heapify the array;
while the array isn't empty {
   remove and replace the root;
   reheap the new root node;
}
```

- We have seen that heapifying takes O(n log n) time
- The while loop takes O(n log n) time
- The total time is therefore O(n log n) + O(n log n)
- This is the same as O(n log n) time

CSE102_Anuradha_G

The End

Shell Sort: Analysis

N	Insertion Sort	Shellsort		
		Shell's	Odd Gaps Only	Dividing by 2.2
1000	122	11	11	9
2000	483	26	21	23
4000	1936	61	59	54
8000	7950	153	141	114
16000	32560	358	322	269
32000	131911	869	752	575
64000	520000	2091	1705	1249

$O(N^{3/2})$? $O(N^{5/4})$? $O(N^{7/6})$?

So we have 3 nested loops, but Shell Sort is still better than Insertion Sort! Why?

Generic Sort

- So far we have methods to sort integers. What about Strings? Employees? Cookies?

CSE102_Anuradha_G