CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 5: Divide-and-Conquer and Sorting Algorithms 2

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

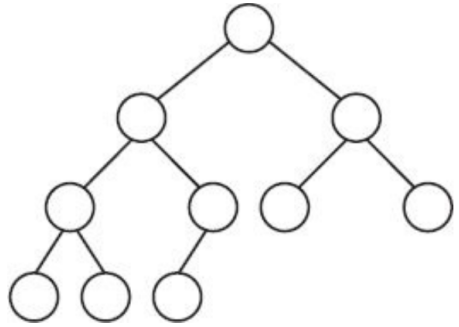
HEAPSORT

Heapsort

- Unlike Mergesort and Quicksort, Heapsort is an in-place $\Theta(n \lg n)$ algorithm.
- It uses the data structure: Heap.

Binary Tree Review

- The *depth* of a node in a tree is the number of edges in the unique path from the root to that node.
- The depth d of a tree is the maximum depth of all nodes in the tree.
- A *leaf* in a tree is any node with no children.
- An *internal node* in a tree is any node that has at least one child.
  - That is, it is any node that is not a leaf.
- A *complete binary tree* is a binary tree that satisfies the following conditions:
  - All internal nodes have two children.
  - All leaves have depth *d*.
- An *essentially complete binary tree* is a binary tree that satisfies the following conditions:
  - It is a complete binary tree down to a depth of *d* – 1.
  - The nodes with depth *d* are as far to the left as possible.

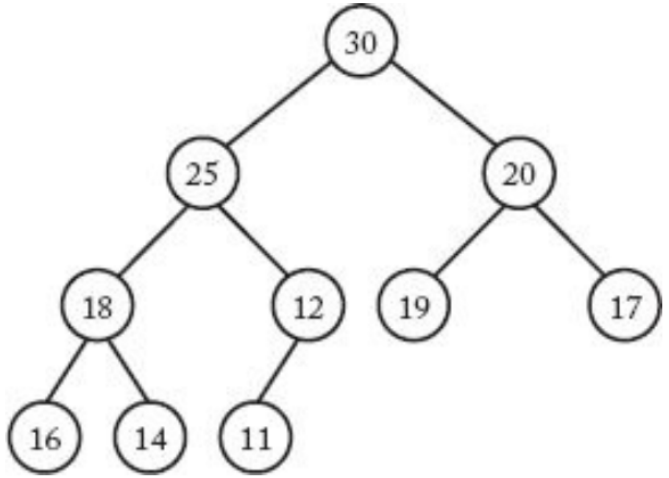


An essentially complete binary tree

Image source: Figure 7.4, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Heap

- A *heap* is an essentially complete binary tree such that
  - The values stored at the nodes come from an ordered set.
  - The value stored at each node is greater than or equal to the values stored at its children. This is called the *heap property*.

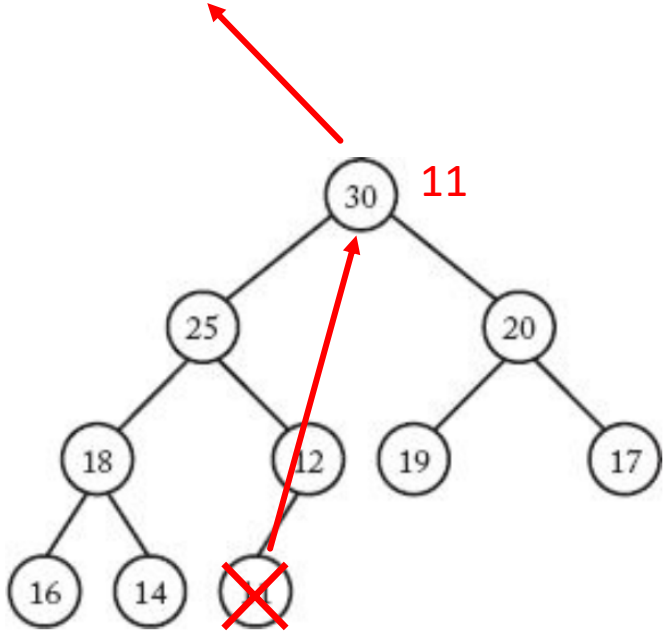


A heap

Image source: Figure 7.5, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Sort by Heap

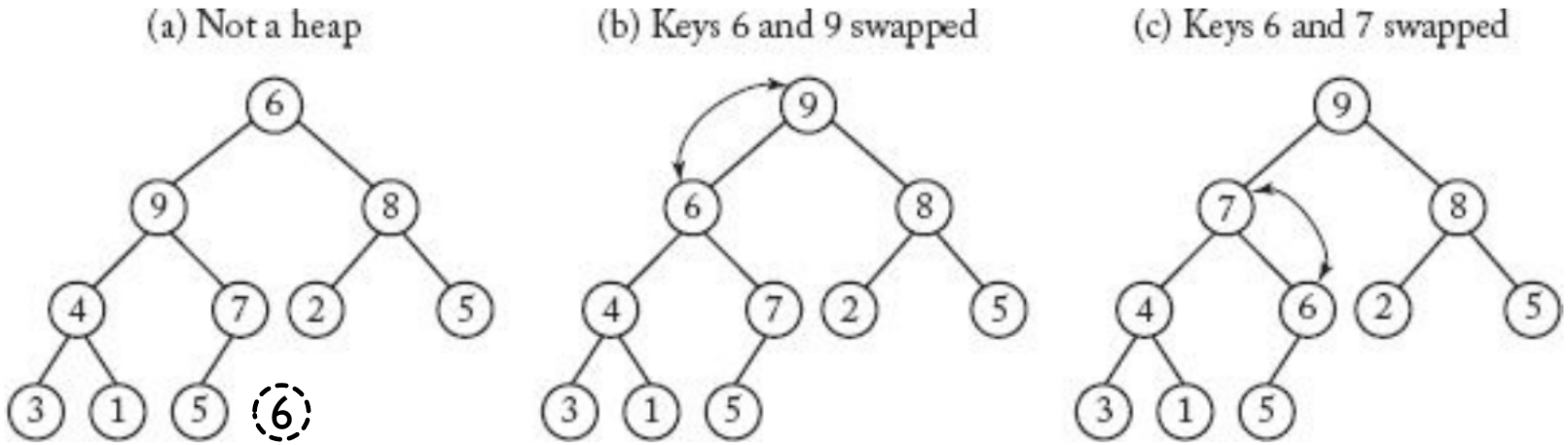
- Because key at the root is always the largest, we can iteratively:
  - Remove the key at the root and store it in an array.
  - Restore the property of the heap.
- The array will be sorted.
  - Placing backwards from the end results a nondecreasing array.
- How to restore?
  - Copy the key at the bottom node (the farthest and rightest node) to the root.
  - Delete the bottom node.
  - Sift the key at the root to a proper position that satisfy the property of a heap.



A heap

Image source: Figure 7.5, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Restore a Heap by Siftdown



Procedure siftdown sifts 6 down until the heap property is restored

- Now we know how to sort by heap and how to restore a heap.
- The remaining problem is how to construct a heap.

Image source: Figure 7.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

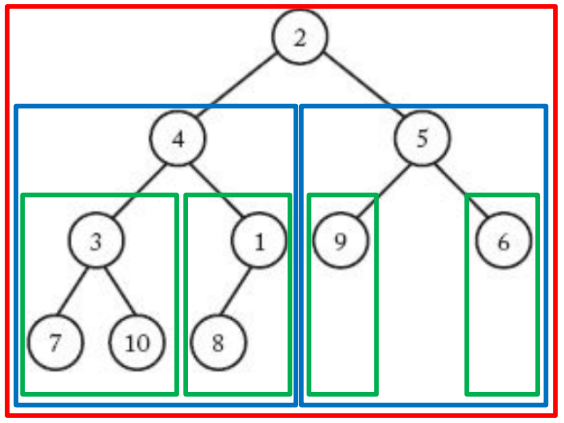
Construct a Heap

- We transform the tree into a heap by repreatedly calling siftdown from bottom.
- Start from all subtrees whose roots have depth $d-1, d-2, \ldots, 0$.

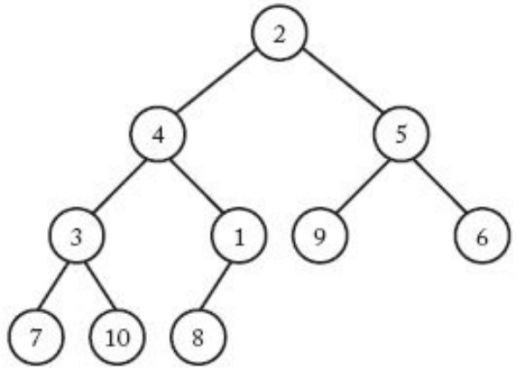
Depth 0

Depth 1

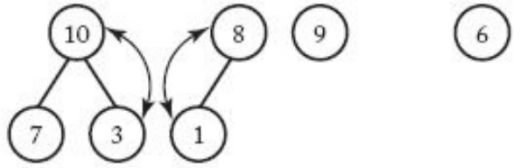
Depth 2



(a) The initial structure



(b) The subtrees, whose roots have depth $d-1$, are made into heaps.



(c) The left subtree, whose root has depth $d-2$, are made into a heap.

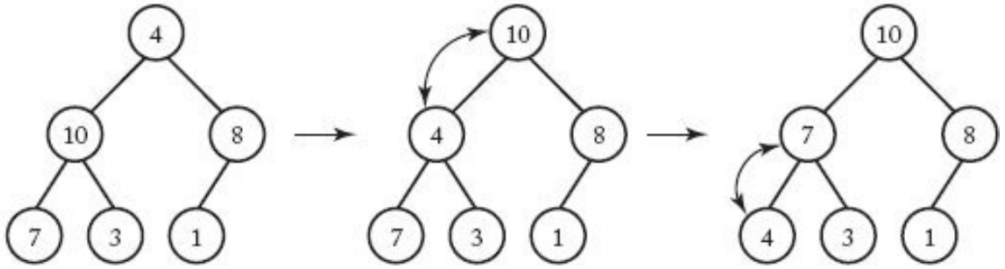
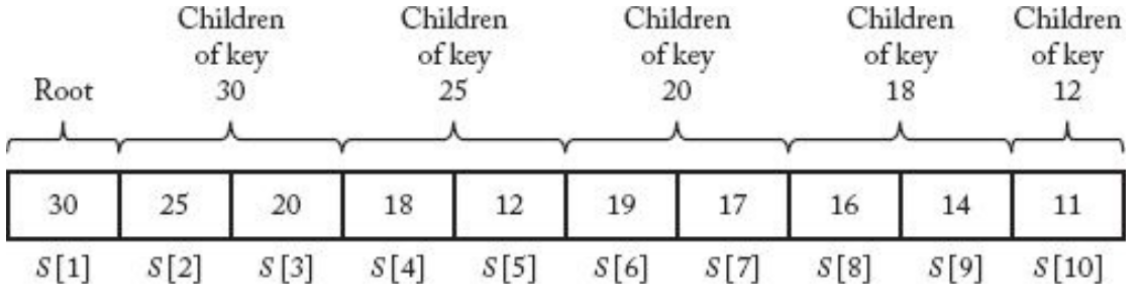


Image source: Figure 7.7, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Implementation of Heapsort

- We directly use array and its index to do tree operations rather than use specific tree data structure.
- If the index of a parent node is x, the index of the left child and right child is simply $2x$ and $2x + 1$.
  - Only essentially complete binary tree can be indexed like this.



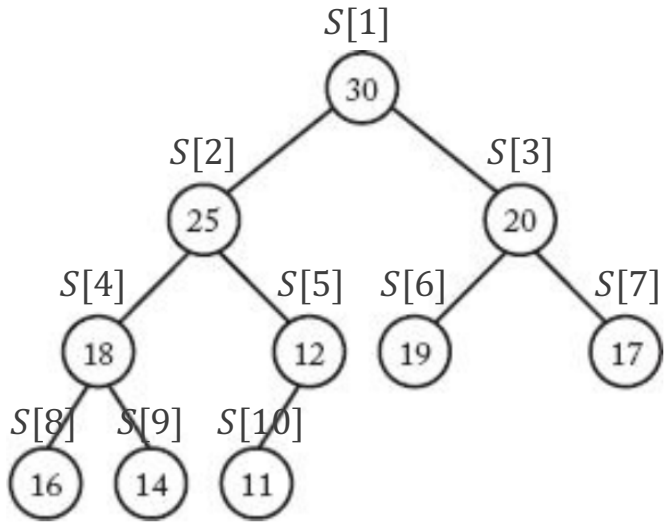


Image source: Figure 7.8, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode of Heapsort

```
struct heap
{
    keytype S[1...n];
    int heapsize;
}
```

```
void removekeys (int n, heap& H, keytype S[])
{
    index i;
    for (i = n; i >= 1; i--)
        S[i] = root(H);
}
```

```
keytype root (heap& H)
{
    keytype keyout;

    keyout = H.S[1]; // get key at the root
    H.S[1] = H.S[heapsize]; // move bottom key to root
    H.heapsize = H.heapsize - 1; // delete bottom node
    siftdown(H, 1); // restore heap property
    return keyout;
}
```

```
void makeheap (int n, heap& H)
{
    index i;
    H.heapsize = n;
    for (i = ⌊n / 2⌋; i >= 1; i--)
        siftdown(H, i);
}
```

```
void heapsort (int n, heap& H)
{
    makeheap(n, H);
    removekeys(n, H, H.S);
}
```

```
void siftdown (heap& H, index i)
{
    index parent, largerchild;
    keytype siftkey;
    bool spotfound;

    siftkey = H.S[i];
    parent = i;
    spotfound = false;
    while (2 * parent <= H.heapsize && !spotfound){
        if (2 * parent < H.heapsize && H.S[2 * parent] < H.S[2 * parent + 1])
            largerchild = 2 * parent + 1;
        else
            largerchild = 2 * parent;
        if (siftkey < H.S[largerchild]){
            H.S[parent] = H.S[largerchild];
            parent = largerchild;
        }
        else
            spotfound = true;
    }
    H.S[parent] = siftkey;
}
```

Space Complexity of Heapsort

- During `removekeys`, we are actually moving the first item to the tail of the array, and then restore the shrunken heap.
- Thus, Heapsort is a true in-place sorting algorithm.

```
void heapsort (int n, heap& H)
{
    makeheap(n, H);
    removekeys(n, H, H.S);
}
```

```
void removekeys (int n, heap& H, keytype S[])
{
    index i;
    for (i = n; i >= 1; i--)
        S[i] = root(H);
}
```

Worst-Case Time Complexity Analysis of Makeheap

- Worst-case: each item needs to be sifted to the bottom.
- Assume n is a power of 2. We have a heap like the figure when $n = 8$.
- We first ignore the nth item in the heap. Thus, we have 2^j nodes with the jth depth and the greatest number of siftdown operation is $d - j - 1$ for each node.

Depth	Number of Nodes with this Depth	Greatest Number of Siftdown Operation for Each Node
0	2^0	$d - 1$
1	2^1	$d - 2$
2	2^2	$d - 3$
...	...	...
j	2^j	$d - j - 1$
...	...	...
$d - 1$	2^{d-1}	0

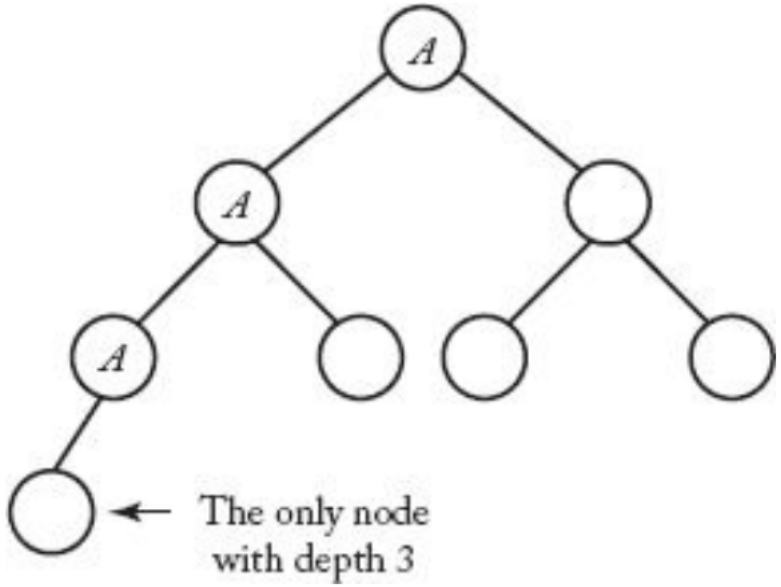


Image source: Figure 7.9, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Worst-Case Time Complexity Analysis of Makeheap

- Therefore, if upper bound of the number of siftdown operations in makeheap is:

$$\sum_{j=0}^{d-1} 2^j(d-j-1)$$

$$= (d-1)\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j\left(2^j\right)$$

$$= (d-1)\left(2^d-1\right) - \left((d-2)2^d+2\right)$$

$$= 2^d - d - 1.$$

- The ignored nth node has d ancestors, so at most d more siftdown operations will be conducted. Therefore, the actual upper bound is

$$2^d - d - 1 + d = 2^d - 1 = n - 1.$$

Worst-Case Time Complexity Analysis of Makeheap

- Because there are two comparisons of keys for each siftdown opearation, the number of comparisons of keys done by `makeheap` is at most $2(n-1)$.
- It is actually superising that heap can be constructed in linear time. If `removekeys` is also linear time, then we may obtain a linear-time sorting algorithm!
  - Is it possible?

Worst-Case Time Complexity Analysis of RemoveKeys

- The examples shows the case of $n = 8$ and $d = \lg 8 = 3$.
- When the first four keys are removed, the new key in the root sifts through at most 2 nodes.
- Then, when the next two keys are removed, the new key in the root sifts through at most 1 nodes.
- Finally, when there's only two keys, the new key in the root doesn't need to be sifted.

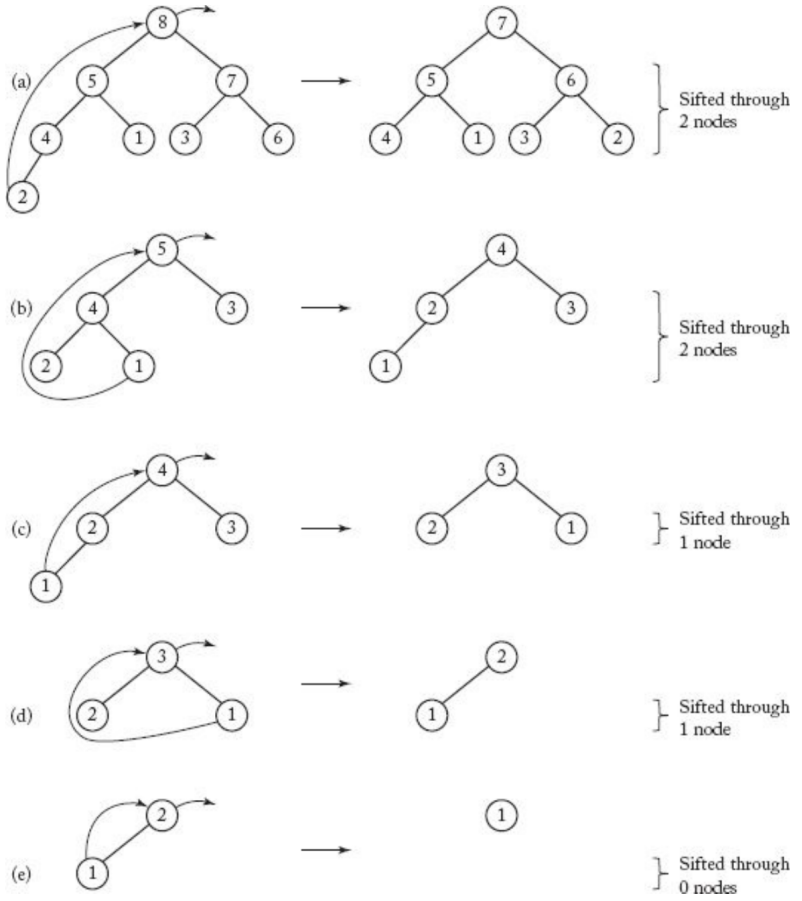


Image source: Figure 7.10, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Worst-Case Time Complexity Analysis of RemoveKeys

- Totally, the number of comparisons of keys done by `removekeys` is at most

$$2\sum_{j=1}^{d-1} j2^j = 2\left(d2^d - 2^{d+1} + 1\right) = 2n\lg n - 4n + 4.$$

- Although `makeheap` is linear-time, `removekeys` still needs $\Theta(n \lg n)$.
- Totally, the worst-case time complexity for Heapsort is

$$W(n) \approx 2n\lg n \in \Theta(n\lg n).$$

- It is difficult to analyze Heapsort's average-case time complexity analytically. However, empirical studies have shown that its average case is not much better than its worse case.

Comparison of Mergesort, Quicksort, and Heapsort

Sorting Algrotihm	Comparison of Keys	Extra Space Usage
Mergesort	$W(n) = n \lg n$ $A(n) = n \lg n$	$\Theta(n)$ records
Quicksort	$W(n) = n^2/2$ $A(n) = 1.38n \lg n$	$\Theta(\lg n)$ indices
Heapsort	$W(n) = 2n \lg n$ $A(n) = 2n \lg n$	In-place

COMPUTATIONAL COMPLEXITY ANALYSIS

Sorting Algorithms

- So far we have learned $\Theta(n^2)$ sorting algorithm exchange sort and $\Theta(n \log n)$ sorting algorithms Mergesort, Quicksort and Heapsort.
- Is it possible to further decrease the complexity of sorting algorithm?
  - Is a $\Theta(n)$ sorting algorithm possible?
- There are two approaches for the problem: Prove you can do or prove you can't do.
  - The $\Theta(n)$ sorting algorithm is developed.
  - Prove that the $\Theta(n)$ sorting algorithm is not possible.
- Once we have such a proof, we don't struggle to improve sorting algorithms and do something intereting else.
  - Actually, people have proven that an sorting algorithm better than $\Theta(n \log n)$ is impossible.

Computational Complexity

- Previously, we determine the time complexity of an algorithm.
- We do not analyze the problem that the algorithm solves.
- For example:
  - We have a $\Theta(n^3)$ algorithm to solve the matrix multiplication problem.
  - It doesn't mean that the problem requires a $\Theta(n^3)$ algorithm.
  - We have further propose $\Theta(n^{2.81})$ and $\Theta(n^{2.38})$ algorithm for the problem.
- What is the minimum complexity requirement for a problem to be solved?

Computational Complexity

- A *computational complexity analysis* tries to determine a lower bound on the efficiency of all algorithms for a given problem.
- It has been proved that the computational complexity of matrix multiplication problem is $\Omega(n^2)$.
  - Up to now, nobody finds a $\Theta(n^2)$ algorithm for matrix multiplication.
  - And nobody finds a lower bound higher than $\Omega(n^2)$ either.
- We usually use Ω notation for computational complexity analysis and Θ or *O* notation for time complexity analysis.
  - We are interested in the lower bound of a problem, and the upper bound of an algorithm.

Decision Tree for Sorting Algorithms

- We can associate the sorting process with a binary tree by representing each node as a comparison.
- This tree is called a *decision tree* because at each node a decision must be made as to which node to visit next.
- A decision tree is called *valid* for sorting n keys if it can sort every permutation of array with size n.
- A decision tree is called *pruned* if every leaf can be reached from the root by making a consistent sequence of decision.

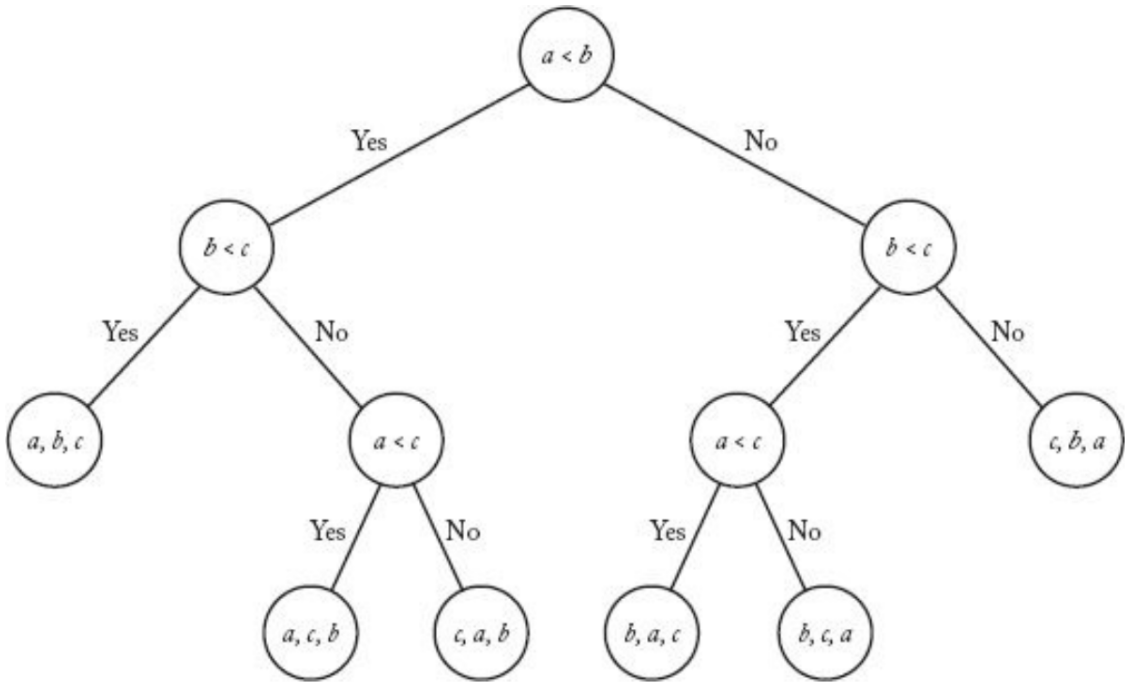


Image source: Figure 7.11, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Lower Bounds for Worst-Case Behavior

Lemma 1

To every deterministic algorithm for sorting n distinct keys there corresponds a pruned, valid, binary decision tree containing exactly $n!$ leaves.

Lemma 2

The worst-case number of comparisons done by a decision tree is equal to its depth.

Lemma 3

Assume that a binary tree with depth d and number of leaves m:

$$d \geq \lceil \lg m \rceil$$

Proof in scratch:

For a complete binary tree, $2^d = m$. Thus, for any binary tree, $2^d \geq m$. Since d is an integer, by taking lg of both side we have $d \geq \lceil \lg m \rceil$. You may refer to the textbook for the more rigorous proof.

Lower Bounds for Worst-Case Behavior

Theorem 1

Any deterministic algorithm that sorts n distinct keys only by comparisons of keys must in the worst case do at least

$$\lceil \lg n! \rceil \text{ comparison of keys.}$$

Proof:

- By Lemma 1, $m = n!$.
- By Lemma 2, worst-case number of comparisons is d.
- By Lemma 3, $d \geq \lceil \lg m \rceil$.

Lower Bounds for Worst-Case Behavior

Lemma 4

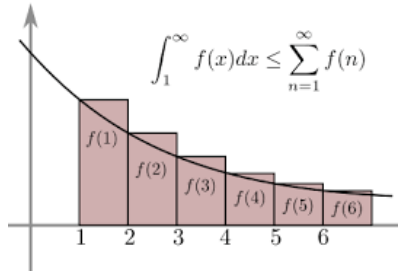
For any positive integer n,

$$\lg(n!) \geq n \lg n - 1.45n.$$

Proof:

$$\lg(n!) = \lg[n(n-1)(n-2)\dots(2)(1)] = \sum_{i=1}^{n} \lg i$$

$$\geq \int_{1}^{n} \lg x \, dx = \frac{1}{\ln 2}(n \ln n - n + 1) \geq n \lg n - 1.45n.$$



Theorem 3

Any deterministic algorithm that sorts n distinct keys only by comparisons of keys must in the worst case do at least

$$\lceil n \lg n - 1.45n \rceil \text{ comparison of keys.}$$

Image source: https://www.quora.com/What-is-the-difference-between-improper-integral-and-infinite-sum

Lower Bounds for Average-Case Behavior

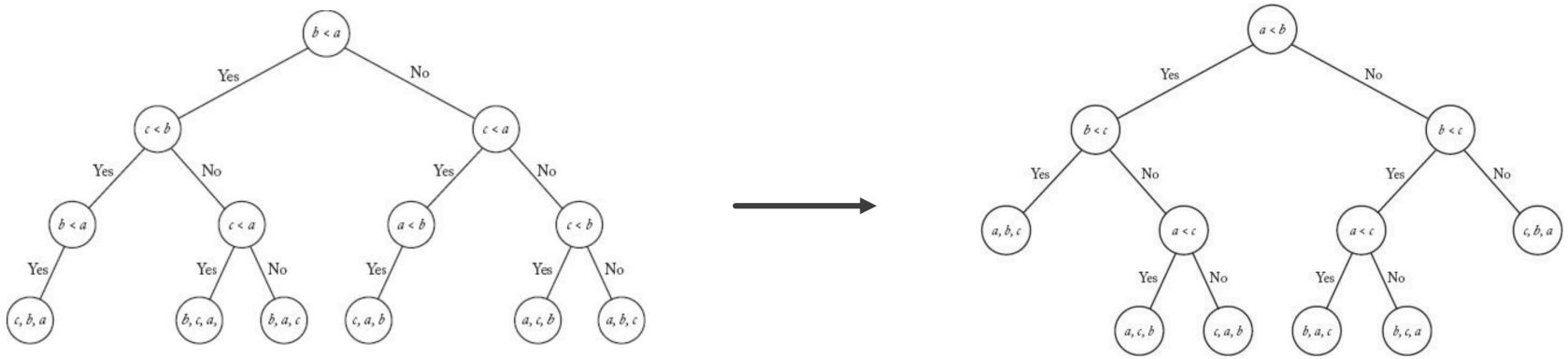
- We see that Mergesort's worst-case performance of $n \lg n - (n - 1)$ is close to optimal.
- Next we show that thia also holds for its average-case performance.

2-Tree

- A binary tree in which every nonleaf contains exactly two children is called a *2-tree*.
- If the decision tree for sorting n distinct keys contains any comparison nodes with only one child, we can change it to a 2-tree.

Lemma 5

To every pruned, valid, binary decision tree for sorting n distinct keys, there corresponds a pruned, valid decision 2-tree that is at least as efficient as the original tree.



A 2-tree

Image source: Figure 7.11-12, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

External Path Length (EPL)

- The *external path length* (*EPL*) of a tree is the total length of all paths from the root to the leaves. For example, for the tree in the figure, $EPL = 2 + 3 + 3 + 3 + 3 + 2 = 16$.
- The *EPL* of a decision tree is the total number of comparisons done by the decision tree to sort all $n!$ possible inputs.
- The average number of comparisons done by a decision tree for sorting n distinct keys is given by

$$\frac{EPL}{n!}.$$

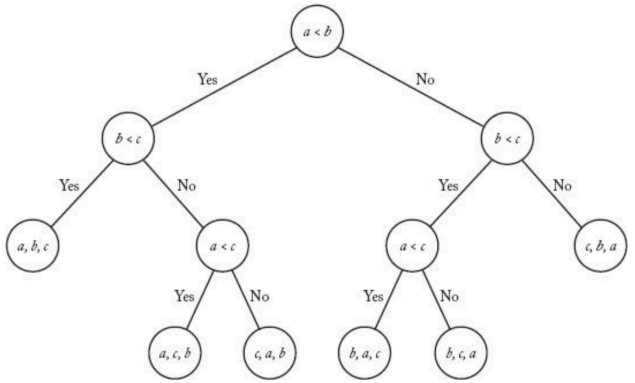


Image source: Figure 7.11, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Minimum EPL

- First we define $minEPL(m)$ as the minimum of the EPL of 2-trees containing m leaves

Lemma 6

Any deterministic algorithm that sorts n distinct keys only by comparisons of keys must on the average do at least

$$\frac{minEPL(n!)}{n!}$$ comparison of keys.

Minimum EPL

Lemma 7

Any 2-tree that has m leaves and whose EPL equals $minEPL(m)$ must have all of its leaves on at most the bottom two levels.

Proof:

- We use proof by contradiction. Suppose that some 2-tree whose EPL equals $minEPL(m)$ does not have all of its leaves on the bottom two levels.
- Let d be the depth of the tree, let A be a leaf in the tree that is *not* on one of the bottom two levels, and let k be the depth of A.
- Because nodes at the bottom level have depth d, we have $k \leq d - 2$.

Minimum EPL

Proof of Lemma 7 (cont'd):

- We can choose a nonleaf B at level $d-1$ in the 2-tree, remove its two children, and give two children to A.
- We lose two children of B and A as the leaf, so that EPL is decrease by $2d+k$.
- We add two children to A and B as the new leaf, so that EPL is increased by $2(k+1)+d-1$.
- The difference of EPL is -1 after moving children. It means that the EPL of original 2-tree is not $minEPL(m)$.

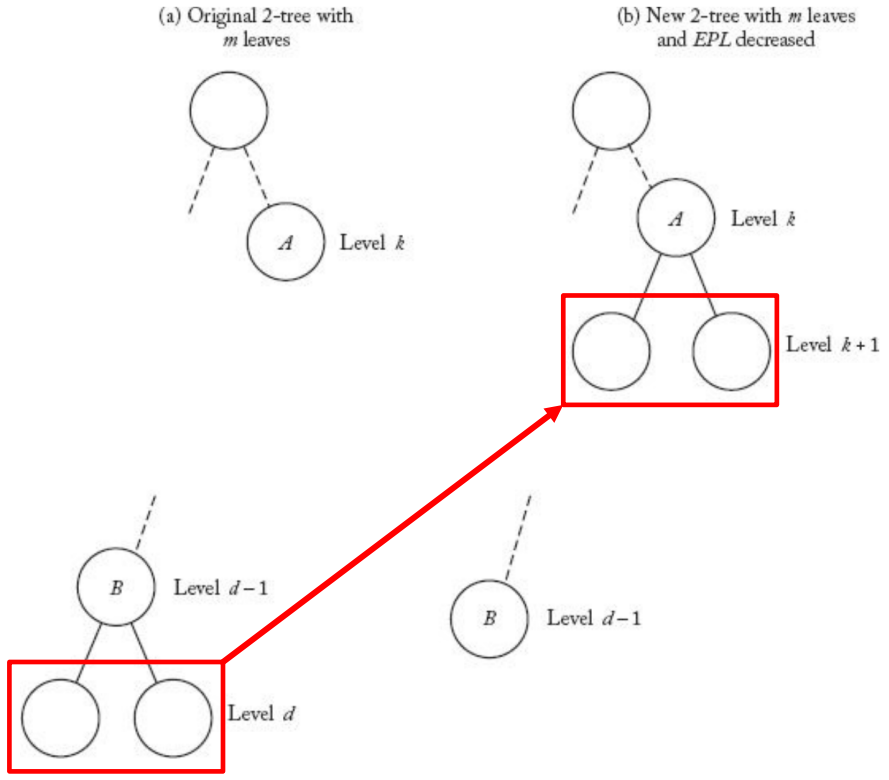


Image source: Figure 7.13, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Minimum EPL

Lemma 8

Any 2-tree that has m leaves and whose EPL equals $minEPL(m)$ must have

$$2^d - m \text{ leaves at level } d-1 \quad and \quad 2m - 2^d \text{ leaves at level } d,$$

And have no other leaves, where d is the depth of the tree.

Proof:

- Because Lemma 7 says that all leaves are at the bottom two levels and because nonleaves in a 2-tree must have two children, there must by 2^{d-1} nodes at level $d-1$ (including leaves and nonleaves).
- If we set r as the number of leaves at level $d-1$, the number of nonleaves at that level is $2^{d-1} - r$.
- Because nonleaves in a 2-tree have exactly two children, there are $2(2^{d-1} - r)$ on level d.
- Totally, there are $m = r + 2(2^{d-1} - r)$ leaves. We get $r = 2^d - m$.

Minimum EPL

Lemma 9

For any 2-tree that has m leaves and whose EPL equals $minEPL(m)$, the depth d is given by

$$d = \lceil \lg m \rceil.$$

Lower Bound of Minimum EPL

Lemma 10

For all integers $m \geq 1$

$$minEPL(m) \geq m\lceil \lg m \rceil.$$

Proof:

- By Lemma 8, we know that
  - There are $2^d - m$ leaves at level $d - 1$.
  - There are $2m - 2^d$ leaves at level d.
- Therefore

$$minEPL(m) = \left(2^d - m\right)(d-1) + \left(2m - 2^d\right)d = md + m - 2^d.$$

- Replace $d = \lceil \lg m \rceil$ by Lemma 9:

$$minEPL(m) = m(\lceil \lg m \rceil) + m - 2^{\lceil \lg m \rceil}.$$

Lower Bound of Minimum EPL

Proof of Lemma 10 (cont'd):

- If m is a power of 2,

$$m(\lceil \lg m \rceil) + m - 2^{\lceil \lg m \rceil} = m \lg m = m \lfloor \lg m \rfloor.$$

- If m is not a power of 2, then $\lceil \lg m \rceil = \lfloor \lg m \rfloor + 1$. So, in this case,

$$\begin{aligned} minEPL(m) &= m(\lfloor \lg m \rfloor + 1) + \mathrm{m} - 2^{\lceil \lg m \rceil} \\ &= m \lfloor \lg m \rfloor + 2m - 2^{\lceil \lg m \rceil} > m \lfloor \lg m \rfloor. \end{aligned}$$

because in general $2m > 2^{\lceil \lg m \rceil}$.

Lower Bounds for Average-Case Behavior

Theorem 4

Any deterministic algorithm that sorts n distinct keys only by comparisons of keys must on the average do at least

$$\lfloor n \lg n - 1.45n \rfloor \text{ comparisons of keys.}$$

Proof:

- By Lemma 6, any such algorithm must on the average do at least

$$\frac{minEPL(n!)}{n!} \text{ comparison of keys.}$$

- By Lemma 10, $minEPL(m) \geq m\lfloor \lg m \rfloor$, this expression is greater than or equal to

$$\frac{n!\,\lfloor \lg(n!) \rfloor}{n!} = \lfloor \lg(n!) \rfloor.$$

- Follow Lemma 4, $\lg(n!) \geq n \lg n - 1.45n$, the theorem is proved.

Lower Bounds for Average-Case Behavior

- Now you can see, for the sorting problem:
  - The worst-case needs at least $\lceil n \lg n - 1.45n \rceil$ comparison of keys.
  - The average-case needs at least $\lfloor n \lg n - 1.45n \rfloor$ comparison of keys.
- You're not required to remember all the details of the proof.
- But you should have mastered the idea of how to prove the computational complexity of sorting algorithm by using decision tree and EPL.
  - Now, you know better of the essence of a sorting problem.

RADIX SORT

More Than Comparing by Keys

- We showed that any algorithm that sorts *only by comparisons of keys* can be no better than $\Theta(n \lg n)$.
  - The computational analysis is only for the case of only by comparisons of keys.
- If we know nothing about the keys, we have no choice but to sort by comparing the keys.
- However, when we have *more knowledge* we can consider other sorting algorithms.
- By using additional information about the keys, we next develop one such algorithm.

Divide into Piles

- Suppose that the keys are all nonnegative integers represented in base 10.
- We can first distribute them into distinct piles based on the values of the leftmost digits.
  - Keys with the same leftmost digit are placed in the same pile.
- Then, the second and third digits from the left.
- After we have inspected all the digits, the keys will be sorted.
- A difficulty with this procedure is that we need a variable number of piles.
  - You need lots of piles and most of them are empty.

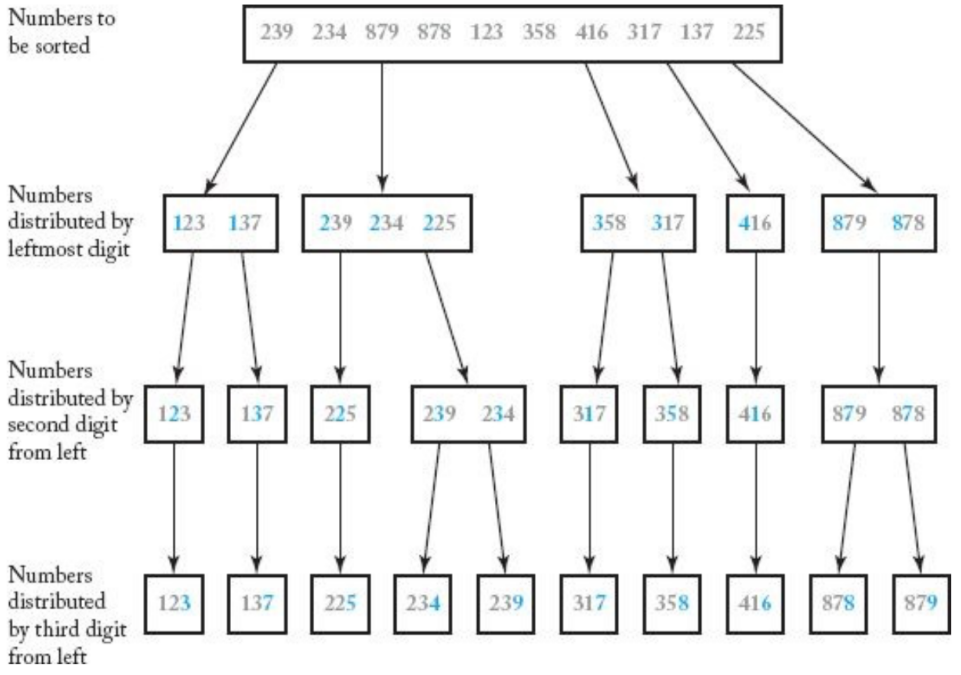
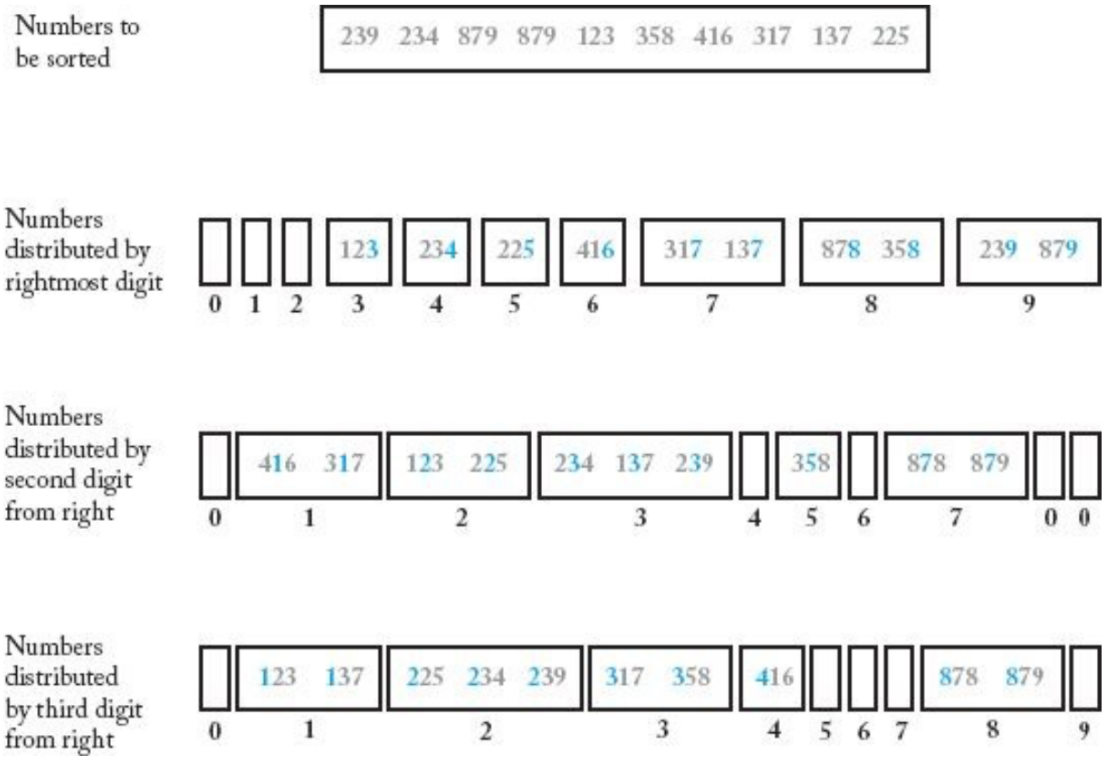


Image source: Figure 7.14, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Divide into Piles

- Solution:
  - We inspect digits from right to left, and we always place a key in the pile corresponding to the digit currently being inspected.
  - On each pass, if two keys are to be placed in the same pile, they follow the order in the previous pass.



Radix Sort

- This sorting method is called *radix sort* because the information used to sort the keys is a particular radix (base).
- The radix could be any number base, or we could use the letters of the alphabet.
- The number of piles is the same as the radix.
  - If we were sorting numbers represented in hexadecimal, the number of piles would be 16.
  - If we were sorting alpha keys represented in the English alphabet, the number of piles would be 26.

Pseudocode of Radix Sort

- Because the number of keys in a particular pile changes with each pass, a good way to implement the algorithm is to use linked lists.
- Each pile is represented by a linked list.
- After each pass, the keys are removed from the lists (piles) by coalescing them into one master linked list.

```
struct nodetype
{
    keytype key;
    nodetype* link;
};
typedef nodetype* node_pointer;
```

```
void radixsort (node_pointer& masterlist, int numdigits)
{
    index i;
    node_pointer list[0...9];

    for (i = 1; i <= numdigits; i++){
        distribute(masterlist, list, i);
        coalesce(masterlist, list);
    }
}
```

```
void distribute (node_pointer& masterlist,
                 node_pointer& list,
                 index i)
{
    index j;
    node_pointer p;

    for (j = 0; j <= 9; j++)
        list[j] = NULL;
    p = masterlist;
    while (p != NULL){
        j = value of ith digit (from the right) in p -> key;
        link p to the end of list[j];
        p = p -> link;
    }
}
```

```
void coalesce (node_pointer& masterlist,
               node_pointer& list);
{
    index j;

    masterlist = NULL;
    for (j = 0; j <= 9; j++)
        link the nodes in list[j] to the end of masterlist;
}
```

Every-Case Time Complexity of Radix Sort

- `numdigits` passes of `distribute` and `coalesce`.
- n passes through the **while** loop in `distribute`.
- 10 passes through the **for** loop in `coalesce`.
- Each of these procedures is called `numdigits` times from `radixsort`. Therefore,

$$T(numdigits, n) = numdigits(n + 10) \in \Theta(numdigits \times n).$$

- It looks like a $\Theta(n)$ algorithm, but not.
  - E.g. sorting 10 numbers with 10 digits takes $\Theta(n^2)$.
- It is suitable for sorting records like social secrity numbers (9 digits integer).

Conclusion

After this lecture, you should know:

- How does Heapsort work?
  - Makeheap, siftdown, removekeys…
- What is the computational complexity?
- What is the computational complexity of sorting problem on worst- and average-case?
- How does Radix Sort work?

Assignment 2

- Assignment 2 is released. The deadline is **18:00, 18th May**.

Thank you!

- Any question?
- Don’t hesitate to send email to me for asking questions and discussion. ☺

Acknowledgement: Thankfully acknowledge slide contents shared by