Lecture 07: Sorting I

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: October 24, 2024

Announcements

1. Third Quiz, due Friday
   - Similar format to before
   - Covers fundamental data structures (Lectures 4–6)
   - Quiz is **closed resource**
     - No books, notes, internet, etc.
     - Do not discuss until after submission deadline (Friday night, after midnight)
2. Programming Assignment (Draft) Posted
   - Due Wednesday, 13 November
3. Attendance Code:

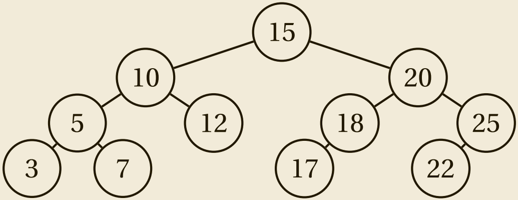
Meeting Goals

- Finish up balanced binary trees
- Discuss the sorting task
- Introduce HEAPSORT
- Discuss Divide and Conquer approaches to sorting
  - MERGESORT
  - QUICKSORT

AVL Trees

From Last Time

Binary Search Trees



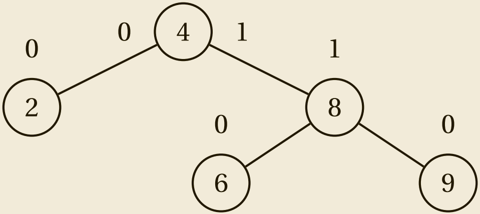
Height and Balance

- **height** of v = max distance to a descendent leaf
- T is **height balanced** if for every v, the heights of v's children differ by at most 1
- Properties of height balanced trees
  - height h satisfies $h \leq 2 \log n$
  - CONTAINS, ADD, REMOVE run in $O(\log n)$ time

Question. How can we efficiently maintain height balance for any sequence of operations?

Creating Imbalance

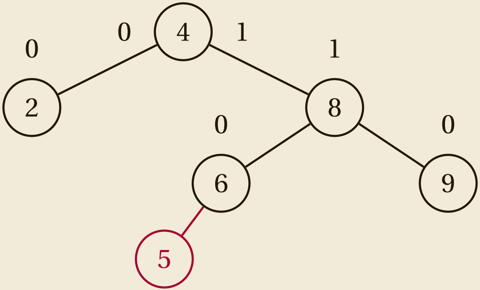
A Minimal Working Example (MWE) **balanced**



Question. What happens when we ADD(5)?

Creating Imbalance

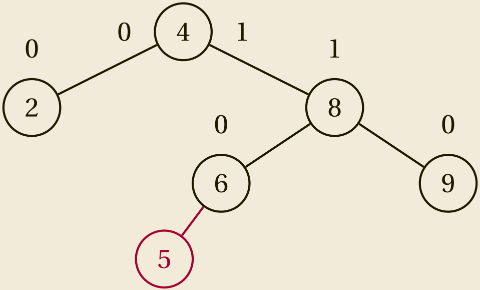
A Minimal Working Example (MWE) **unbalanced**



Question. What happens when we ADD(5)?

Creating Imbalance

A Minimal Working Example (MWE)



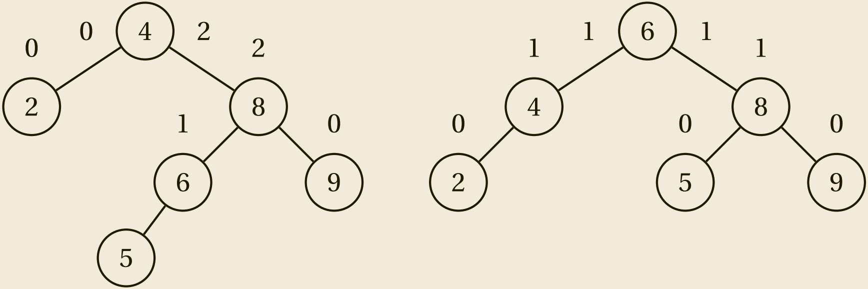
Question. What happens when we ADD(5)?

PollEverywhere

Which vertices are unbalanced?

pollev.com/comp526

Fixing Imbalance



General Strategy. Find the “lowest” unbalanced vertex, and “pull up” its lower child.

Unbalanced Observations

Suppose T was balanced before $\text{ADD}(x)$ and unbalanced after $\text{ADD}(x)$. Then:

1. $\text{ADD}(x)$ can only change the height/balance of x's **ancestors**.
2. The height of any vertex can can only increase by one as the result of $\text{ADD}(x)$.

Unbalanced Observations

Suppose T was balanced before ADD(x) and unbalanced after ADD(x). Then:

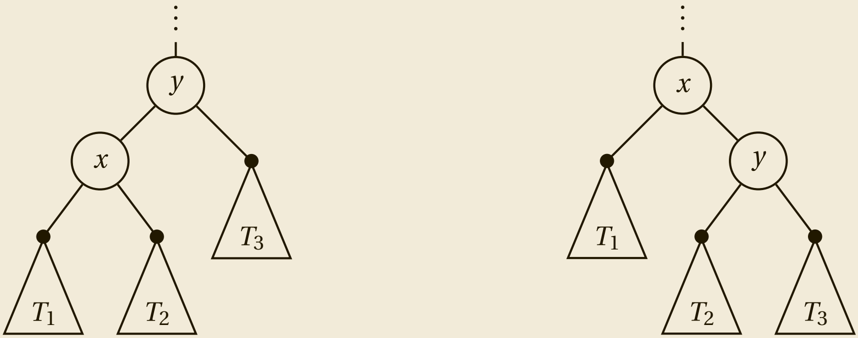
1. ADD(x) can only change the height/balance of x's **ancestors**.
2. The height of any vertex can can only increase by one as the result of ADD(x).

This means:

- We only need to check x's ancestors for imbalance after ADD(x).
- We only need to correct an imbalance of 2 to restore balance in the tree after ADD(x).

Rotations

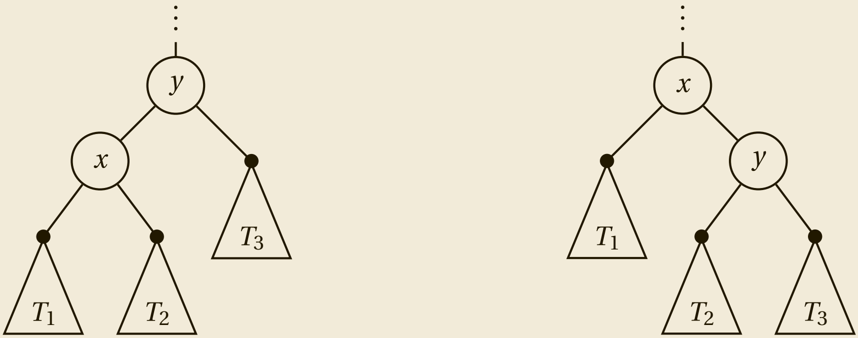
$\Longrightarrow$ right rotation at y $\Longrightarrow$



$\Longleftarrow$ left rotation at x $\Longleftarrow$

Rotations

$\Longrightarrow$ right rotation at y $\Longrightarrow$



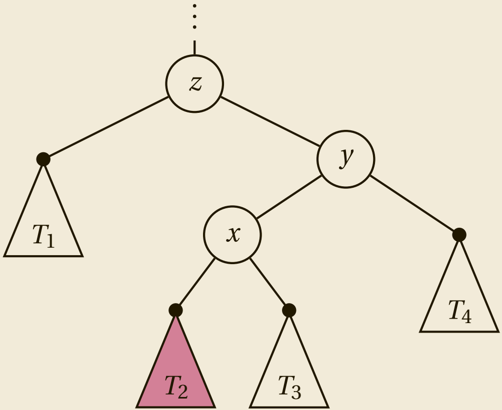
$\Longleftarrow$ left rotation at x $\Longleftarrow$

Main Observation. If T is a BST, then it remains a BST after any rotation.

Restoring Balance After Add

Suppose T was balanced before $\text{ADD}(w)$ and is unbalanced after the operation. Then define

- z is w's closest unbalanced **ancestor**
- y is z's child towards w
- x is y's child towards w
    - Why do these vertices exist?

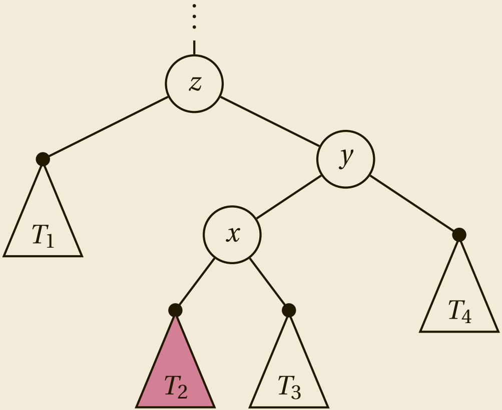


Heights After Add

PollEverywhere

If z had height h before $\textsc{Add}(w)$, what are the heights of z, T_1, y, and x afterward?

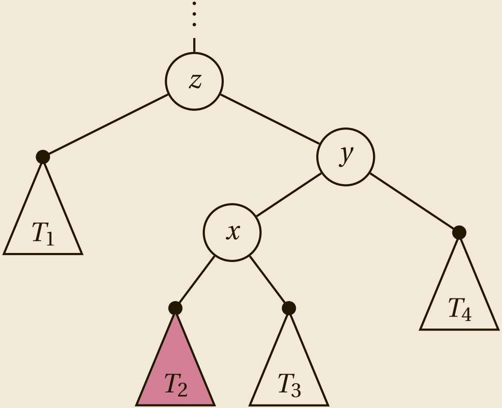
pollev.com/comp526



Heights After Add

Heights after $\textsc{Add}(w)$

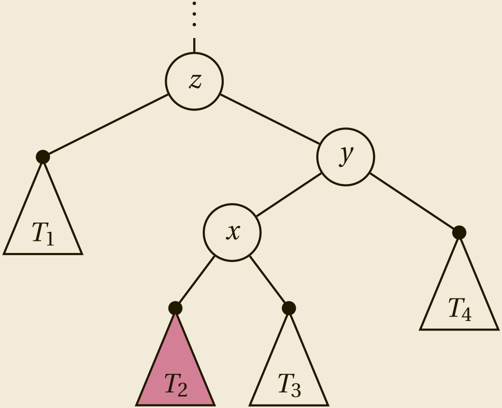
- $z : h+1$
- $y : h$
- $x : h-1$
- $T_1 : h-2$
- $T_2 : h-2$
    - why not $h-3$?
- $T_3 : h-3$
    - why not $h-4$?
- $T_4 : h-2$



Heights After Add

Heights after ADD(w)

- $z : h+1$
- $y : h$
- $x : h-1$
- $T_1 : h-2$
- $T_2 : h-2$
    - why not $h-3$?
- $T_3 : h-3$
    - why not $h-4$?
- $T_4 : h-2$



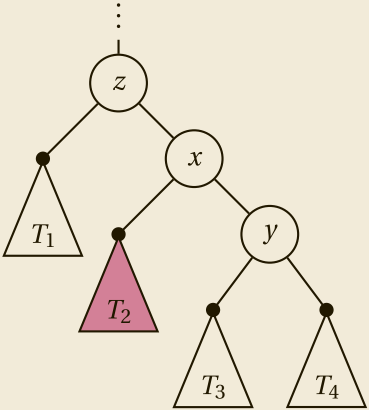
Question. How to “pull” T_2 up?

Heights After Right Rotation at y

PollEverywhere

What is the new height of z's right child?

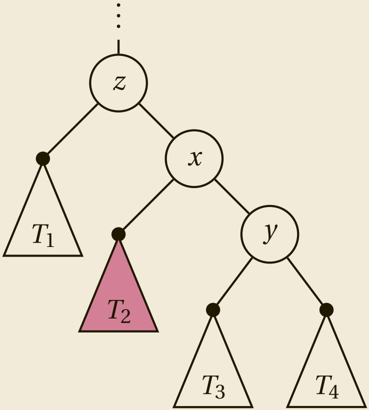
pollev.com/comp526



Heights After Right Rotation at y

Heights after Right Rotation at y

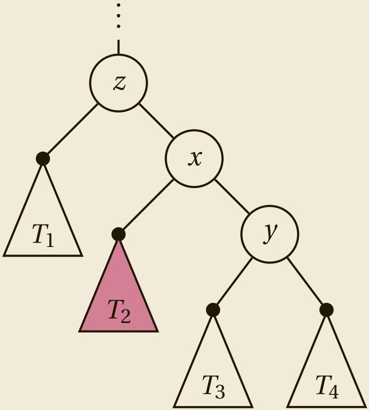
- $z : h+1$
- $y : h-1$
- $x : h$
- $T_1 : h-2$
- $T_2 : h-2$
- $T_3 : h-3$
- $T_4 : h-2$



Heights After Right Rotation at y

Heights after Right Rotation at y

- $z : h+1$
- $y : h-1$
- $x : h$
- $T_1 : h-2$
- $T_2 : h-2$
- $T_3 : h-3$
- $T_4 : h-2$

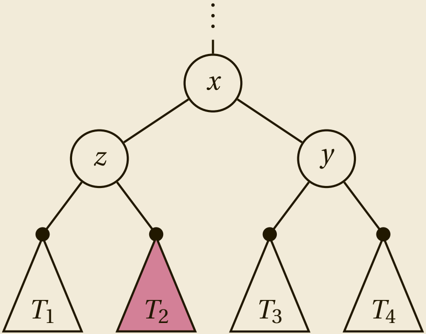


Damn! What if we try again?

Heights After Left Rotation at z

Heights after Right Rotation at y

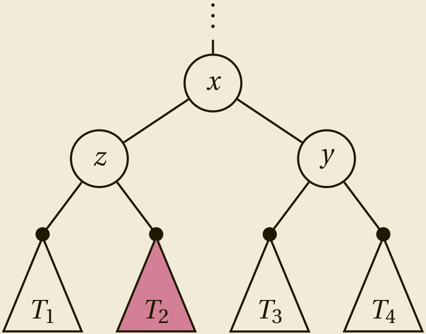
- z :
- y :
- x :
- $T_1 : h-2$
- $T_2 : h-2$
- $T_3 : h-3$
- $T_4 : h-2$



Heights After Left Rotation at z

Heights after Right Rotation at y

- z :
- y :
- x :
- $T_1 : h-2$
- $T_2 : h-2$
- $T_3 : h-3$
- $T_4 : h-2$

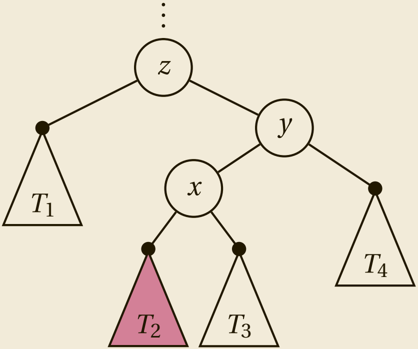


Hooray! We restored balance!!

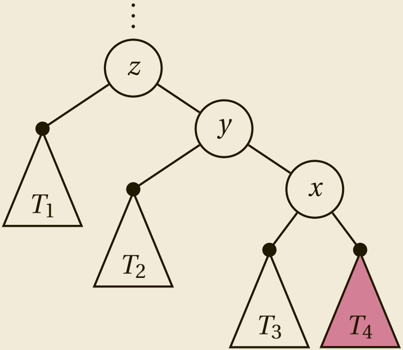
- …Not just at in our subtree, but on the whole tree?

Other Cases

Example we considered



Another Possibility



Only one rotation needed!

Also to consider: mirror images.

- These are the only 4 possibilities for z, y, and x.

Implementation Details

Unfortunately to pull this off, we need more overhead.

- More storage:
  - maintain **height** of each vertex (in addition to references to children, parent)
- More **work** on each ADD/REMOVE:
  - update the heights of vertices
  - check for imbalance
  - restore balance as above

Implementation Details

Unfortunately to pull this off, we need more overhead.

- More storage:
  - maintain **height** of each vertex (in addition to references to children, parent)
- More **work** on each ADD/REMOVE:
  - update the heights of vertices
  - check for imbalance
  - restore balance as above

PollEverywhere

What is the add'l cost of checking/restoring balance for ADD?

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(\sqrt{n})$
4. $\Theta(n)$

pollev.com/comp526

Implementation Details

Unfortunately to pull this off, we need more overhead.

- More storage:
  - maintain **height** of each vertex (in addition to references to children, parent)
- More **work** on each ADD/REMOVE:
  - update the heights of vertices
    - **Only need to update ancestors of added vertex**
  - check for imbalance
    - **Only need to check ancestors of added vertex**
  - restore balance as above
    - **Only takes $O(1)$ time!**

PollEverywhere

What is the add'l cost of checking/restoring balance for ADD?

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(\sqrt{n})$
4. $\Theta(n)$

`pollev.com/comp526`

They Payoff

This scheme for balancing BST is called **AVL trees**

- Named for **A**delson-**V**elsky and **L**andis (1962)

Similar re-balancing technique also works for REMOVE method

- Re-balancing removal also takes worst case $\Theta(\log n)$ time.

Big Deal: We can now implement ORDEREDSETS and MAPS where **all** operations are performed in worst case $O(\log n)$ time!

They Payoff

This scheme for balancing BST is called **AVL trees**

- Named for **A**delson-**V**elsky and **L**andis (1962)

Similar re-balancing technique also works for REMOVE method

- Re-balancing removal also takes worst case $\Theta(\log n)$ time.

Big Deal: We can now implement ORDEREDSETS and MAPS where **all** operations are performed in worst case $O(\log n)$ time!

Other balanced (binary) tree implementations also exist:

- Red-Black trees
- Scapegoat trees
- 2-3 trees
- …

All have similar *worst case, asymptotic* running time

- different implementations suited for different applications

ADT & Data Structure Recap

Simple ADTs

- STACK
- QUEUE
- DEQUE

Efficient implementation with linear data structures:

- arrays
- linked lists

All operations performed in (amortized) $\Theta(1)$ time.

ADT & Data Structure Recap

Simple ADTs

- Stack
- Queue
- Deque

Efficient implementation with linear data structures:

- arrays
- linked lists

All operations performed in (amortized) $\Theta(1)$ time.

Sophisticated ADTs

- PriorityQueue
- Map (associative array, dictionary, symbol table)

Efficient implementation with tree-like data structures

- heaps
- (balanced) binary search trees

All operations in (amortized) $O(\log n)$ time.

Sorting

The Sorting Task

Fundamental Task: sorting a list of elements from smallest to largest

7	1	2	5	3	4	8	6

$\longmapsto$

1	2	3	4	5	6	7	8

Typical basic (unit cost) operations:

- **compare** two elements to see which is larger
- **swap** two elements in the array

The Sorting Task

Fundamental Task: sorting a list of elements from smallest to largest

7	1	2	5	3	4	8	6

$\longmapsto$

1	2	3	4	5	6	7	8

Typical basic (unit cost) operations:

- **compare** two elements to see which is larger
- **swap** two elements in the array

(Perhaps) surprisingly sorting is still an active area of study/research!

- practical and theoretical improvements still being found
- algorithms for different contexts
  - e.g., non-standard sorting models

Elementary Sorting

Iterative sorting:

- Sort in **phases** where each phase accomplishes some global task.

Three Basic Strategies

1. SELECTIONSORT
    - Each phase i finds the smallest element in $a[i \ldots n-1]$ and swaps it into position i
    - Uses (asymptotically) fewest SWAPS possible

Elementary Sorting

Iterative sorting:

- Sort in **phases** where each phase accomplishes some global task.

Three Basic Strategies

1. SELECTIONSORT
   - Each phase i finds the smallest element in $a[i \dots n-1]$ and swaps it into position i
   - Uses (asymptotically) fewest SWAPS possible
2. BUBBLESORT
   - Each phase iterates over adjacent *pairs* and swaps those which are out of order
     - after phase i, $a[n-i-1 \dots n-1]$ contains the i largest elements sorted
   - Used mostly for illustrative purposes.

Elementary Sorting

Iterative sorting:

- Sort in **phases** where each phase accomplishes some global task.

Three Basic Strategies

1. SELECTIONSORT
   - Each phase i finds the smallest element in $a[i \dots n-1]$ and swaps it into position i
   - Uses (asymptotically) fewest SWAPS possible
2. BUBBLESORT
   - Each phase iterates over adjacent *pairs* and swaps those which are out of order
     - after phase i, $a[n-i-1 \dots n-1]$ contains the i largest elements sorted
   - Used mostly for illustrative purposes.
3. INSERTIONSORT
   - Each phase i inserts $x = a[i]$ into sorted order in $a[0 \dots i]$
   - Typically fast for *small* sequences and "almost sorted" sequences

InsertionSort in Detail

Phases $i = 1, 2, \ldots, n-1$:

- Phase i moves $x = a[i]$ into sorted position in $a[0 \ldots i]$.
- Performed via adjacent comparisons:
  - if x is smaller than left neighbor, swap x with left neighbor

procedure INSERTIONSORT(a, n)
    for $i = 1, 2, \ldots, n-1$ **do**
        $j \leftarrow i$
        while $j > 0$ and $a[j] < a[j-1]$ **do**
            SWAP$(a, j, j-1)$
            $j \leftarrow j-1$
        end while
    end for
end procedure

InsertionSort in Detail

PollEverywhere

What is the *worst case* running time of InsertionSort?

1. $\Theta(n)$
2. $\Theta(n \log n)$
3. $\Theta(n^2)$
4. $\Theta(2^n)$

pollev.com/comp526

```
procedure InsertionSort(a, n)
    for i = 1, 2, ..., n − 1 do
        j ← i
        while j > 0 and a[j] < a[j − 1] do
            Swap(a, j, j − 1)
            j ← j − 1
        end while
    end for
end procedure
```

InsertionSort in Detail

State after each phase:

0	1	2	3	4
4	3	5	1	2
3	4	5	1	2
3	4	5	1	2
1	3	4	5	2
1	2	3	4	5

procedure INSERTIONSORT(a, n)
  for $i = 1, 2, \ldots, n-1$ **do**
    $j \leftarrow i$
    while $j > 0$ and $a[j] < a[j-1]$ **do**
      SWAP$(a, j, j-1)$
      $j \leftarrow j-1$
    end while
  end for
end procedure

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

Sorting Using Heaps

Recall the (array backed) heap data structure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	13	10	6	66	39	42	17	96	70	89	95	98	63

Heap Operations in $O(\log n)$ time:

- $\textsc{Insert}(x)$
- $\textsc{RemoveMin}()$.

Question. How to use heaps to sort *efficiently* ($o(n^2)$ time)?

Sorting Using Heaps

Recall the (array backed) heap data structure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	13	10	6	66	39	42	17	96	70	89	95	98	63

Heap Operations in $O(\log n)$ time:

- $\textsc{Insert}(x)$
- $\textsc{RemoveMin}()$.

Question. How to use heaps to sort *efficiently* ($o(n^2)$ time)?

- Add all elements to a heap.
- Repeatedly $\textsc{RemoveMin}$ and add elements back to sorted array

What is the running time of this procedure?

Sorting Using Heaps

Recall the (array backed) heap data structure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	13	10	6	66	39	42	17	96	70	89	95	98	63

Heap Operations in $O(\log n)$ time:

- $\textsc{Insert}(x)$
- $\textsc{RemoveMin}()$.

Question. How to use heaps to sort *efficiently* ($o(n^2)$ time)?

- Add all elements to a heap.
- Repeatedly $\textsc{RemoveMin}$ and add elements back to sorted array

What is the running time of this procedure?

- $\Theta(n \log n)$ This is much better than $\Theta(n^2)$!

Another Question. Do we need a separate heap?

Sorting In-Place

Heap Modification: **MaxHeap**

- Same as MinHeap, but all inequalities reversed
  - **Largest** value at root
  - Children store **smaller** values

HEAPSORT outline:

1. Make array a MaxHeap
   - HEAPIFY by calling BUBBLEUP at each index
2. Sort from right side of array
   - swap $a[0]$ with $a[n-i-1]$
   - TRICKLEDOWN from $a[0]$ to $a[n-i-1]$

Sorting In-Place

Heap Modification: **MaxHeap**

- Same as MinHeap, but all inequalities reversed
  - **Largest** value at root
  - Children store **smaller** values

HEAPSORT outline:

1. Make array a MaxHeap
   - HEAPIFY by calling BUBBLEUP at each index
2. Sort from right side of array
   - swap $a[0]$ with $a[n-i-1]$
   - TRICKLEDOWN from $a[0]$ to $a[n-i-1]$

```
procedure HEAPSORT(a, n)
    for i = 1, 2, ..., n − 1 do
        BUBBLEUP(a, i)
                    ▷ Start from index i
    end for
    for i = n − 1, n − 2, ..., 1 do
        SWAP(a, 0, i)
        TRICKLEDOWN(a, i − 1)
                    ▷ Stop at index i − 1
    end for
end procedure
```

Sorting In-Place

Heap Modification: **MaxHeap**

- Same as MinHeap, but all inequalities reversed
  - **Largest** value at root
  - Children store **smaller** values

HEAPSORT outline:

1. Make array a MaxHeap
   - HEAPIFY by calling BUBBLEUP at each index
2. Sort from right side of array
   - swap $a[0]$ with $a[n-i-1]$
   - TRICKLEDOWN from $a[0]$ to $a[n-i-1]$

```
procedure HEAPSORT(a, n)
    for i = 1, 2, ..., n−1 do
        BUBBLEUP(a, i)
                ▷ Start from index i
    end for
    for i = n−1, n−2, ..., 1 do
        SWAP(a, 0, i)
        TRICKLEDOWN(a, i−1)
                ▷ Stop at index i−1
    end for
end procedure
```

Question. What is the running time of HEAPSORT?

HeapSort Example

Step 1: HEAPIFY!

0	1	2	3	4
4	3	5	1	2
4	3	5	1	2
5	3	4	1	2
5	3	4	1	2
5	3	4	1	2

procedure HEAPSORT(a, n)
    for $i = 1, 2, \ldots, n-1$ **do**
        BUBBLEUP(a, i)
                ▷ Start from index i
    end for
    for $i = n-1, n-2, \ldots, 1$ **do**
        SWAP($a, 0, i$)
        TRICKLEDOWN($a, i-1$)
                ▷ Stop at index $i-1$
    end for
end procedure

HeapSort Example

Step 2: Remove maximum values!

0	1	2	3	4
5	4	3	1	2
4	2	3	1	5
3	2	1	4	5
2	1	3	4	5
1	2	3	4	5

procedure HEAPSORT(a, n)
    for $i = 1, 2, \ldots, n-1$ **do**
        BUBBLEUP(a, i)
                ▷ Start from index i
    end for
    for $i = n-1, n-2, \ldots, 1$ **do**
        SWAP($a, 0, i$)
        TRICKLEDOWN($a, i-1$)
                ▷ Stop at index $i-1$
    end for
end procedure

Worst case running time is $\Theta(\log n)$, but HEAPSORT doesn't perform great in practice (for large arrays)

- poor locality of reference

Sorting by Divide & Conquer

The Divide & Conquer Strategy

Generic Strategy

Given an algorithmic task:

1. Break the input into smaller instances of the task
2. Solve the smaller instances
   - this is typically recursive!
3. Combine smaller solutions to a solution to the whole task

The Divide & Conquer Strategy

Generic Strategy

Given an algorithmic task:

1. Break the input into smaller instances of the task
2. Solve the smaller instances
   - this is typically recursive!
3. Combine smaller solutions to a solution to the whole task

Divide & Conquer Sorting

MergeSort: Divide by *index*

- divide array into left and right halves
- recursively sort halves
- merge halves

The Divide & Conquer Strategy

Generic Strategy

Given an algorithmic task:

1. Break the input into smaller instances of the task
2. Solve the smaller instances
   - this is typically recursive!
3. Combine smaller solutions to a solution to the whole task

Divide & Conquer Sorting

MERGESORT: Divide by *index*

- divide array into left and right halves
- recursively sort halves
- merge halves

QUICKSORT: Divide by *value*

- pick a *pivot value* p
- split array according to p
  - $\leq p$ on left, $> p$ on right
- recursively sort sub-arrays

Merging Sorted Arrays

Question

Suppose we are given two **sorted arrays**, a and b. How can we merge them into a single sorted array that contains all the values from both arrays?

0	1	2	3	4
2	3	6	7	8

0	1	2	3
1	4	5	9

0	1	2	3	4	5	6	7	8

Merging Code

Merging *sorted* arrays a (size m) and b (size n) into array c starting at index s

procedure MERGE(a, b, c, s, m, n) ▷ Merge arrays a and b into array c starting at index s. a has size m and b has size n
    $i, j \leftarrow 0$, $k \leftarrow s$
    while $k < s + m + n$ **do**
        if $j = n$ or $a[i] < b[j]$ **then**
            $c[k] \leftarrow a[i]$
            $i \leftarrow i + 1$
        else
            $c[k] \leftarrow b[j]$
            $j \leftarrow j + 1$
        end if
        $k \leftarrow k + 1$
    end while
end procedure

Merging Code

PollEverywhere

What is the running time of MERGE?

1. $\Theta(m+n)$
2. $\Theta(m \cdot n)$
3. $\Theta(\log(m+n))$
4. $\Theta(\log mn)$

pollev.com/comp526

procedure MERGE(a, b, c, s, m, n) ▷ Merge arrays a and b into array c starting at index s. a has size m and b has size n
    $i, j \leftarrow 0,\ k \leftarrow s$
    while $k < s + m + n$ **do**
        if $j = n$ or $a[i] < b[j]$ **then**
            $c[k] \leftarrow a[i]$
            $i \leftarrow i + 1$
        else
            $c[k] \leftarrow b[j]$
            $j \leftarrow j + 1$
        end if
        $k \leftarrow k + 1$
    end while
end procedure

Sorting by Merging

MERGESORTStrategy:

- To sort $a[i \ldots k]$:
  - If $i = k$, then we're done
  - Othewise split (sub)interval in half
  - Recursively sort halves
  - Merge sorted halves
    - copy values to new arrays for this

Sorting by Merging

MERGESORTStrategy:

- To sort $a[i \ldots k]$:
  - If $i = k$, then we're done
  - Othewise split (sub)interval in half
  - Recursively sort halves
  - Merge sorted halves
    - copy values to new arrays for this

procedure MERGESORT(a, i, k)
    if $i < k$ **then**
        $j \leftarrow \lfloor (i + k)/2 \rfloor$
        MERGESORT(a, i, j)
        MERGESORT$(a, j + 1, k)$
        $b \leftarrow$ COPY(a, i, j)
        $c \leftarrow$ COPY$(a, j + 1, k)$
        MERGE(b, c, a, i)
    end if
end procedure

Sorting by Merging

PollEverywhere

Consider an execution of MergeSort$(a, 0, 3)$ where $a = [4, 2, 1, 3]$. How many total calls to MergeSort are executed (including the initial call)?

pollev.com/comp526

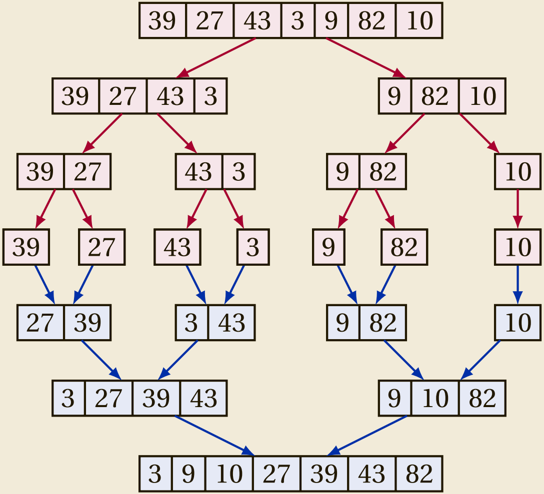
```
procedure MergeSort(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MergeSort(a, i, j)
        MergeSort(a, j + 1, k)
        b ← Copy(a, i, j)
        c ← Copy(a, j + 1, k)
        Merge(b, c, a, i)
    end if
end procedure
```

Sorting by Merging

Tracing the Recursive Calls

procedure MERGESORT(a, i, k)
    if $i < k$ **then**
        $j \leftarrow \lfloor (i+k)/2 \rfloor$
        MERGESORT(a, i, j)
        MERGESORT$(a, j+1, k)$
        $b \leftarrow$ COPY(a, i, j)
        $c \leftarrow$ COPY$(a, j+1, k)$
        MERGE(b, c, a, i)
    end if
end procedure

A Larger Example



tikz code courtesy of SebGlav on `tex.stackexchange.com`

MergeSort Analysis

Question. What is the running time of MERGESORT?

- How do we analyzing the running time of a recursive function?

MergeSort Analysis

Question. What is the running time of MERGESORT?

- How do we analyzing the running time of a recursive function?

Think about this for next time.

Next Time: More Sorting

- MERGESORT analysis
- QUICKSORT
- Lower Bounds
- Non-comparison Based Methods
- More Sorting Algorithms?

Scratch Notes