Tutorial 05 - More Binary Heap, Midtest Review

CS2040C Semester 2 2018/2019

By Jin Zhe, adapted from slides by Ranald+Si Jie, AY1819 S2 Tutor

More Binary Heap Stuffs

Recap: Binary Max-Heap

Main Idea

- Each vertex has at most 2 children (i.e. *binary*)
- The value of each vertex is **≥** than its children, and therefore also all its descendents by inductive reasoning
    - All our operations are to maintain this property
    - Also known as the **heap property**

Recap: Binary Max-Heap

- Height of a complete binary tree is $O(\log N)$
- Root (i.e. topmost) vertex/node is always the max value
- `insert(val)`
  - "Bubble up" from leaf: $O(\log N)$ height
- `ExtractMax()`
  - "Bubble down" from root: $O(\log N)$ height
- `create(arr)`
  - $O(N \log N)$ version
  - $O(N)$ version
- `HeapSort(arr)`
  - $O(N)$ `create(arr)` then $O(\log N)$ `ExtractMax()` N times: $O(N \log N)$

Question 1

Problem statement

Give an algorithm to count all vertices that have value `>x` in a max-heap of size n.

Your algorithm must run in $O(k)$ time where k is the number of vertices in the output.

Key lesson: This is a new algorithm analysis type for most of you as the time complexity of the algorithm does not depend on the input size n but rather the output size k :O...

Note that this question has also been integrated in VisuAlgo Online Quiz, so it may appear in future Online Quizzes :)

Observation

“A binary heap is made up of many binary subheaps”.

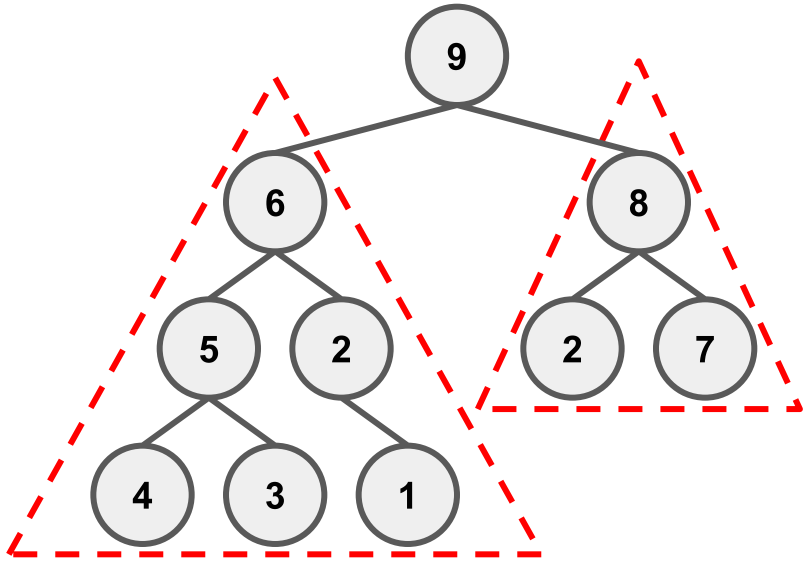
More specifically, a binary max-heap is a complete binary tree where the root vertex has value greater than both its children and each of its children is also a binary heap.

This recursive definition should give you a hint that the solution is likely recursive in nature too…

In fact, most tree-related algorithms are recursive because trees are defined recursively!

Observation

These two are subtrees that are by themselves binary heaps!



Observation

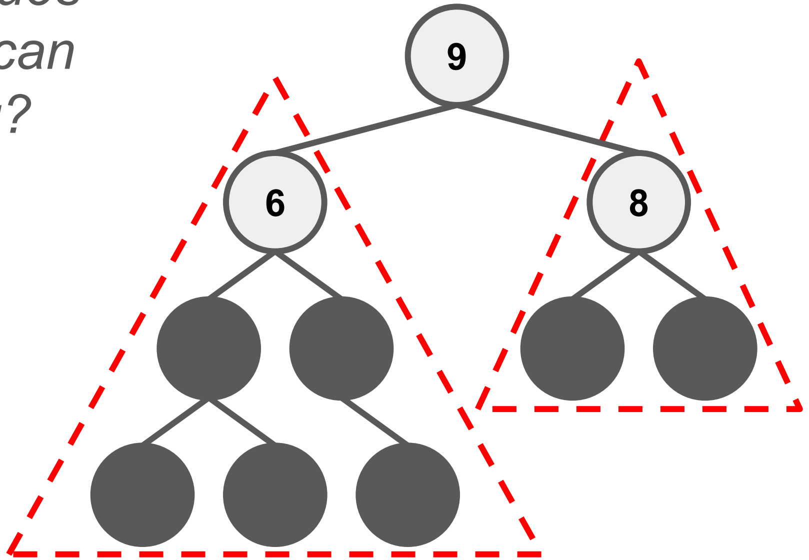
Recall for a binary heap, the root vertex has the greatest value in the entire heap.

i.e. All other vertices (i.e. non-root) `v` in the binary heap, have `v.value <= root.value`

Example

Let’s say I want to find all vertices with value `>7`.

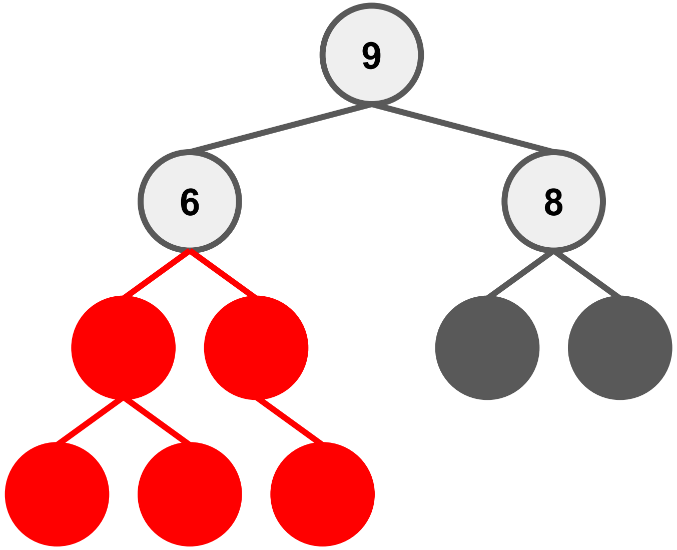
If I hide all the values except the roots, can you infer anything?



Example

Let's say I want to find all vertices with value >7.

You can infer that all these nodes are ≤6 and therefore less than what we want!



Approach

Traverse the tree, for every vertex `v` that we encounter:

- If the value of the vertex `v` is lesser than or equal to `x`, then we can stop the search
- Else we output `v` and continue searching down its children

Solution

We can implement the solution recursively

```
void findVerticesBiggerThanX(vertex v, int x) {
    if (v.value > x) {
        cout << v.value << endl;              // Output
        findVerticesBiggerThanX(v.left, x);   // Recurse left
        findVerticesBiggerThanX(v.right, x); // Recurse right
    }
}
```

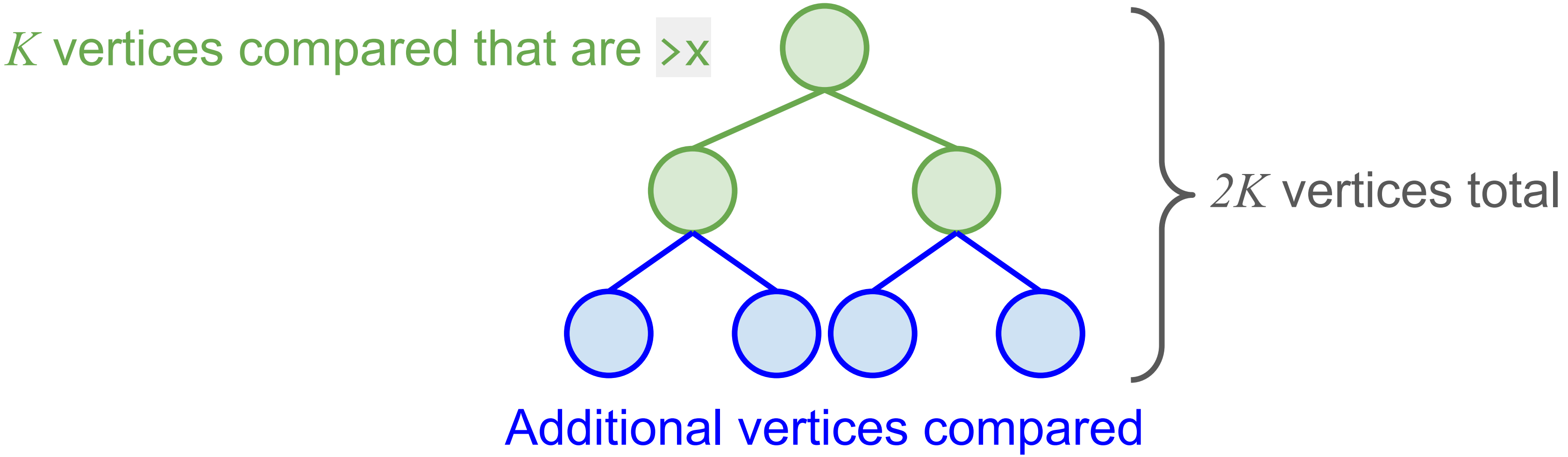
Test yourself!

If the answer has K outputs, what is the total number comparisons done?

Test yourself!

If the answer has K outputs, what is the total number comparisons done?

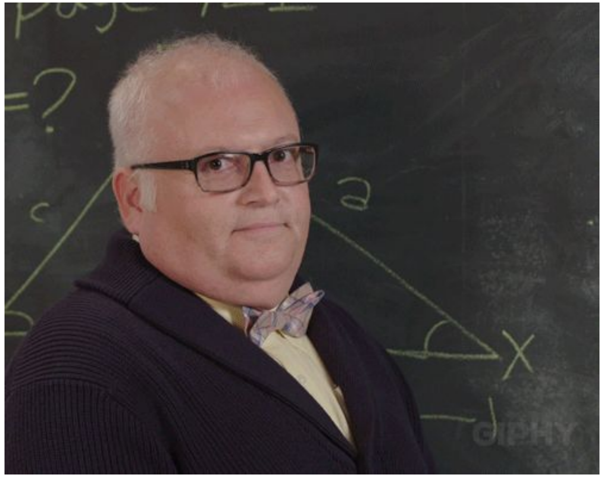
Answer: $2K$ (i.e. 2 comparisons for every vertex with value >x)


K vertices compared that are >x
$2K$ vertices total
Additional vertices compared

Time complexity

Total time complexity is the number of output operations plus the number of vertices compared so,

$$O(K + 2K) = O(3K) = O(K)$$



Tree Traversals

- Actually, this simple recursive algorithm is what we call a depth first search (DFS), a classic tree traversal algorithm!
- More specifically, it is a (pruned) pre-order traversal which is a tail recursion (no deferred operations)
- The different types of tree traversals will become more important in *later weeks* of CS2040C.
- For now, just try to appreciate and understand what we mean. (Self-read the next few slides!)

Tree Traversals (self-read)

Tree traversals comes in 3 (no-pun intended) flavours:

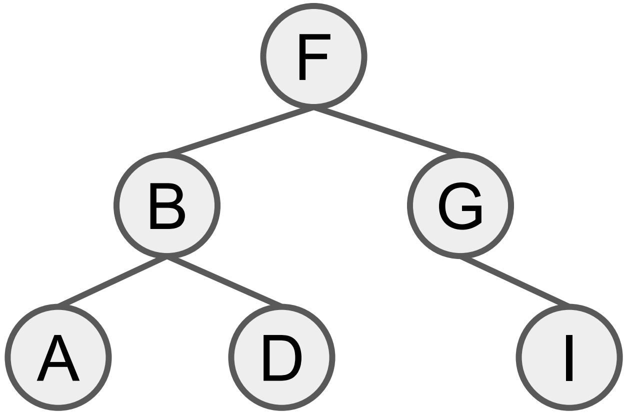
1. *Pre-order traversal*
2. *In-order traversal*
3. *Post-order traversal*

Their name comes from where the “current operation” in each level of the recursion is done in relation to the recursive calls of that level. “Current operation” can simply be printing out the value of the current vertex being visited.

Pre-order Traversal (self-read)

1. Current operation
2. Recruse left
3. Recruse right

```
void pre_order(vertex v) {
    if (v) {
        cout << v.value << " ";
        pre_order(v.left);
        pre_order(v.right);
    }
}
```



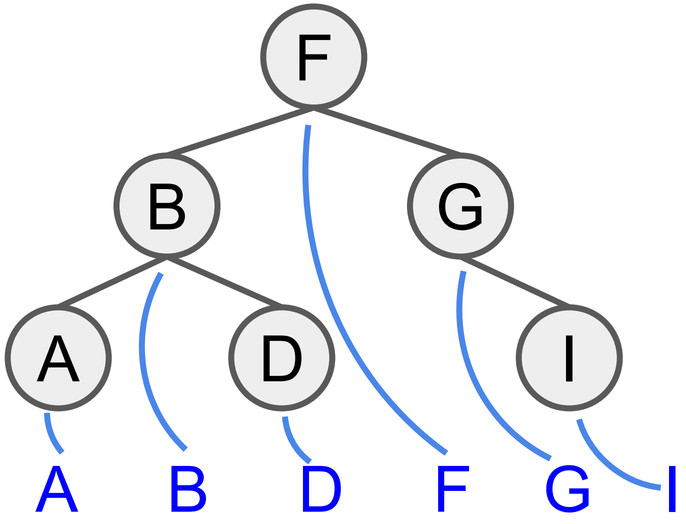
F B A D G I

Print order in blue

In-order Traversal (self-read)

1. Recurse left
2. Current operation
3. Recruse right

```
void in_order(vertex v) {
    if (v) {
        in_order(v.left);
        cout << v.value << " ";
        in_order(v.right);
    }
}
```

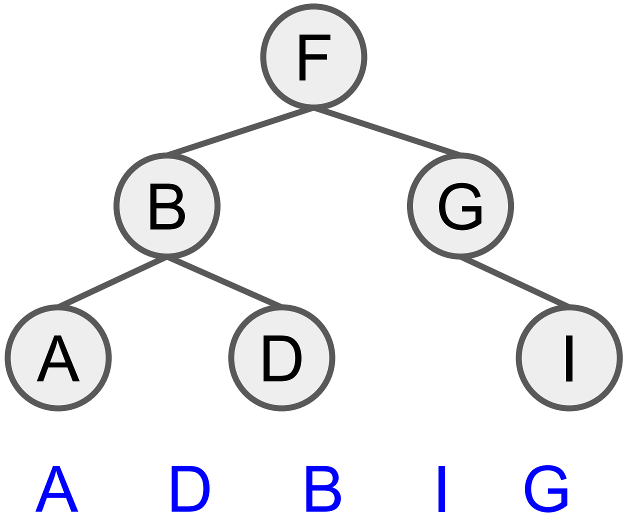


Print order in blue
Resembles “falling leaves” of a tree where lines don’t cross each other!

Post-order Traversal (self-read)

1. Recurse left
2. Recurse right
3. Current operation

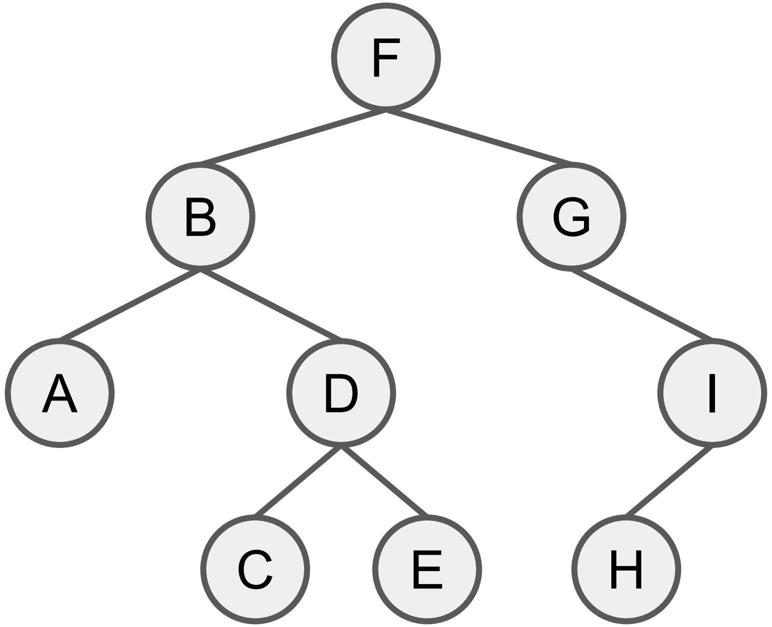
```
void post_order(vertex v) {
    if (v) {
        post_order(v.left);
        post_order(v.right);
        cout << v.value << " ";
    }
}
```



Print order in blue

Test yourself!

For your own practice, try out the 3 types of traversals on this tree!



Question 2

Problem statement

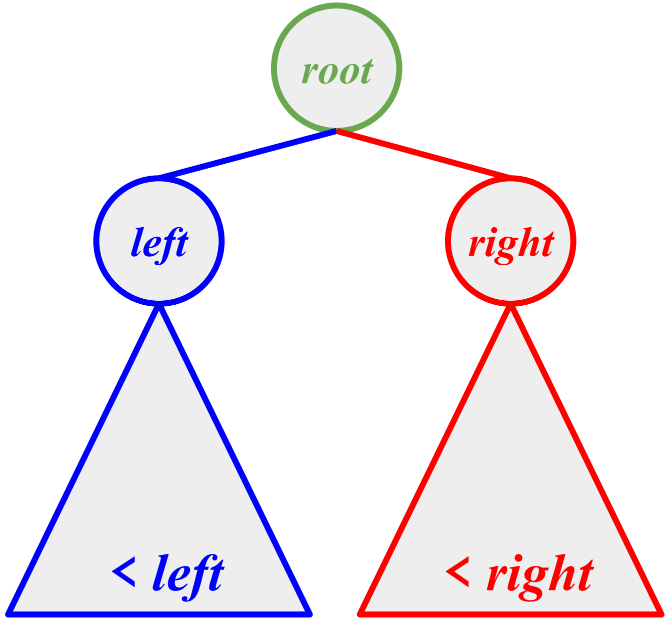
Claim: The second largest element in a max heap with more than two elements is always one of the children of the root. Is this true? If yes, show a simple proof. Otherwise, show a counter example.

For simplicity please assume that all values are unique!

Note that this kind of (simple) proof may appear in future CS2040/C written tests, so please refresh your CS1231 (if you have taken that module) or just concentrate on how the tutor will answer this kind of question.

Proof by contradiction

1. Assume that 2nd largest is neither ***left*** nor ***right***
2. Realize both ***left*** and ***right*** must be lesser than 2nd largest
3. That means 2nd largest is a descendant of either ***left*** or ***right***
4. But descendants cannot be greater than the ancestors in a max-heap! So this is not possible!
5. We reached a contradiction and therefore 2nd largest **must be** either ***left*** or ***right***!



Question 3

Did you know?

There are two interesting features of Binary Heap data structure that are not available in C++ STL `priority_queue` and Java `PriorityQueue` yet:

1. `Increase/Decrease/UpdateKey(old_v, new_v)`
2. `DeleteKey(v)`

These two operations are not yet included in VisuAlgo (the hidden slide https://visualgo.net/en/heap?slide=3-1).

Question 3 a)

Given Steven's Binary Heap Demo code (that is a Binary Max Heap), what should we modify/add so that we can implement `DecreaseKey(old_v, new_lower_v)`?

This discussion shall also suffice for analysis of `Increase` as well as generalised `UpdateKey(old_v, new_v)` since they are analogous.

Questions to ask

- How do we locate `old_v` and how fast can we do it?
- Will it affect Complete Binary Tree (compact array) property? If so how do we handle?
- Will it affect heap property? If so how do we handle?
- What’s the time complexity?

Questions to ask

- How do we locate `old_v` and how fast can we do it?
    - $O(N)$ DFS. Worst case when it is the smallest value in heap
    - $O(1)$ using Hash Table! *Key* is vertex value, *Value* is vertex index in compact array
- Will it affect Complete Binary Tree (compact array) property? If so how do we handle?
    - No because no gaps created
- Will it affect heap property? If so how do we handle?
    - Yes it will. After decreasing value, “bubble down” $O(\log N)$
- What’s the time complexity?
    - $O(\log N)$

Question 3 b)

Given Steven's Binary Heap Demo code (that is a Binary Max Heap), what should we modify/add so that we can implement `DeleteKey(v)` where `v` is not necessarily the max element?

Solution 1: Building upon `UpdateKey`

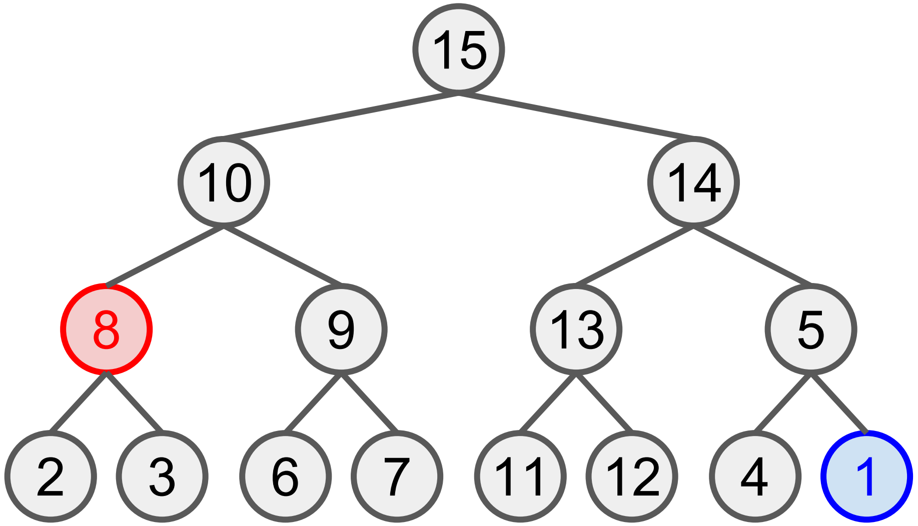
1. `UpdateKey(v, +∞)`
2. `ExtractMax()`

Solution 2: Generalising `ExtractMax()`

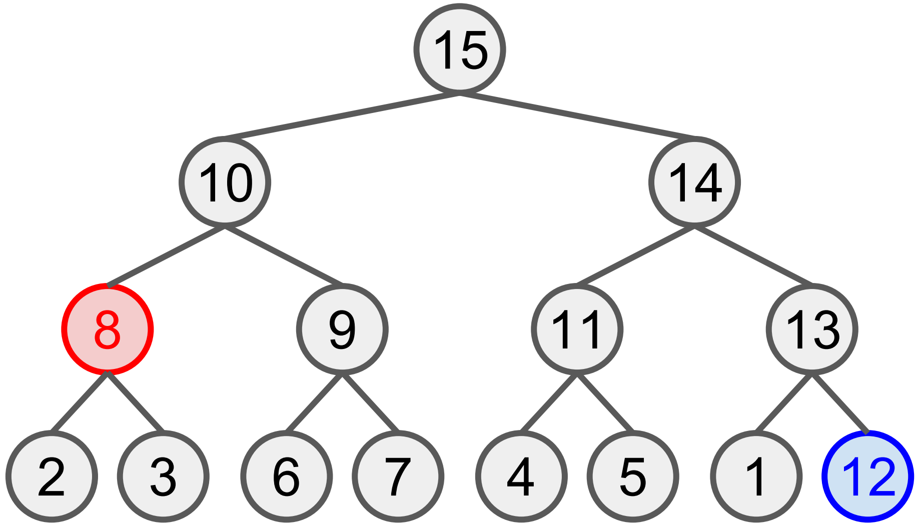
1. Find index of `i` vertex with value `v` in compact array
2. Move value at index `N` into value at index `i`
3. Then `shift_down` or `shift_up`?

Solution 2: Generalising `ExtractMax()`

Consider `DeleteKey(8)` for the two heaps below



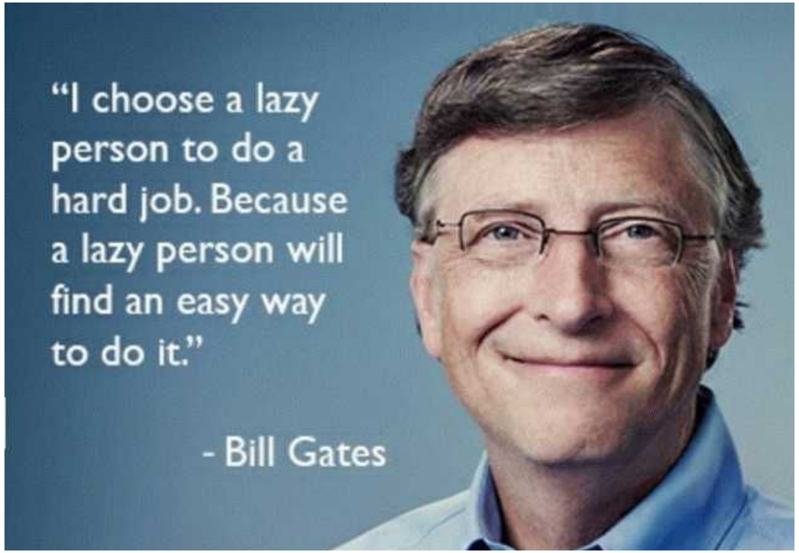
This will need `shift_down`



This will need `shift_up`

Extra stuff—A primer for lazy algorithms

We can actually use a “lazy approach” for `deleteKey(v)` `increaseKey(old_v, new_v)` and `decreaseKey(old_v, new_v)`!


“I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it.”
- Bill Gates

Lazy `DeleteKey(v)`

1. Find vertex. Can be done in $O(1)$ using a Hash Table
2. Flag vertex as “invalid”
3. Done! :O

Complexity: $O(1)$

We will discard “invalid” vertices as we encounter them during during `extractMax()` calls! If we’re lucky to not end up encountering invalidated vertices in our overall routine, we circumvent the $O(\log N)$ deletion time.

Lazy `UpdateKey(old_v, new_v)`

1. Find vertex. Can be done in $O(1)$ using a Hash Table
2. Flag vertex as “invalid”
3. `insert(new_v)`
4. Done!

Complexity: $O(\log N)$

Same as lazy `DeleteKey`, we will discard “invalid” vertices as we encounter them during during `extractMax()` calls

Midterm post-mortem

Question A

Marked by Jin Zhe

Question A一Comments

- Aim: to test your familiarity with string operations and methods in C++
- Therefore we are more strict on correct usage of string methods. i.e. Calling the wrong function or passing in the wrong arguments is given one-off penalizations of 1 mark per mistake (i.e. repeated offences of same mistake ignored)
- Full pseudocode solutions are given 4 marks maximum. 1 mark to describe tokenization process, 1 mark for each of the 3 cases correctly described

Question A—Comments

- There are too many ways to solve this problem!
- `stringstream` method given in model solution achieves one of the most elegant solutions
- Short solutions can also be achieved with a combination of `count`, `find`, `find_last_of`, `substr`
- Many students went ahead with manually tokenizing by iterating through each character. Very error prone! And a nightmare to mark...

Question A—Common mistakes

- Did not read question carefully
  - using `cin` when `Singaporean_name` is given by function
  - `cout`/`printf` instead of returning a `string` (heavy penalization)
- Forgetting to include space between words of call name
- Wrong usage of `stringstream`
- Wrong arguments passed into `count`, `find`, `find_last_of`, `substr`. e.g. Many thought that the second argument of `substr` is the exclusive ending index. It isn’t!
- Calling c-string functions on C++ `std::string`
- Not converting `char` array/vector to `std::string` before returning
- State $O(N)$ for complexity when question stated `Singaporean_name` not more than 30 characters

Question A—Common mistakes

- When conducting manual tokenization:
  - forgot to empty/flush temporary string
  - forgot to add last word in name because condition only checks for space
  - Mistook space with `0`, `'0'`, `'\0'`, `NULL`, `string::npos`
  - not calling `push_back` when appending character to string. Confusing between `append` (for strings) with `push_back` (for characters)
  - Off-by-one errors: included extra space/character or lacking a character

Question B

Marked by Yohanes Yudhi Adikusuma
In case you haven’t see him,
his photo is shown here

Question B.1ー Common mistakes (1)

- Using for loop to find the element in naive $O(n)$ instead of $O(\log n)$ binary search, taking advantage that roster list is always sorted
- Only say “use binary search” without coding it or even explain how it works (unclear on what you want to binary search and what happens if not found)

Question B.2 — Common mistakes (2)

- Simply putting the element at the back of the roster list before calling default C++ sorting function (Steven knows many will do this instead of the faster *$O(n)$* insertion sort - as we keep re-inserting new element to already sorted roster, hence partial marks)

Question B.3 — Common mistakes (2)

- Calling remove function in C++ with `roster.remove()`, actually `std::vector` does not have such function; the `erase` method requires an iterator, not the value to be erased (so we need to find it first using B.1's `inClassRoster`)
- We will soon learn a better data structure to do Question B in *$O(\log n)$* for `inClassRoster`, `addStudent`, and `dropStudent` using a Binary Search Tree (BST). i.e. `std::set<string>`

Question C

Marked by Jin Zhe

Question C.1 — Comments

- Aim: to test your familiarity with custom comparators in sorting
- If you are still not familiar with them, please review the solutions presented for this question in tutorial 4 slides!
- Many don’t realize that only a single sort is required with one loaded custom comparator function!

Question C.1 — Comments

A handful of you came up with one of these 2 solutions (and their variations):

1. Initialize 2 vectors: `odd` and `even`
2. Scan through `A` and respectively populate `odd` and `even` with odd and even numbers encountered
3. Sort `odd` ascending
4. sort `even` descending
5. Concatenate `even` with `odd` and assign that to be `A`

1. Partition `A` into 2 partitions such that left partitions hold all even numbers and right partition holds all odd numbers
2. Count the size of even partition. Call it `E`
3. `sort(A.begin(), A.begin() + E, greater<int>());`
4. `sort(A.begin() + E, A.end());`

Question C.1 — Comments

Other crazy approaches:

- Counting sort
  - Signed 32-bit integer can be anywhere in [-2,147,483,647, 2,147,483,647] so that’s a range of about 4 billion
  - Each integer requires 4 bytes, so total RAM requirement for counting sort is therefore 4×4 billion bytes = 16 GB (Which is crazy! :O)
  - At least 2 students attempted this but their solution was wrong
- Radix sort
  - OK if you can implement, but highly unlikely given the space of answer box and time limitation of the exam!
  - Thankfully haven’t encountered anyone doing this yet

Question C.1—Common mistakes

- Defining a custom (sometimes wrong) comparator for sorting descending instead of using `std::greater<int>()`
- Thinking there is `push_front()`, `remove_front()` for vectors
- Chaining multiple sort routines without using `stable_sort`, thereby losing previous sorted ordering
- Logical errors in comparator functions due to unfamiliarity
- Thinking that comparator logic with constant steps will impact time complexity

Question C.1 — Common mistakes

One method devised by some of you is worthy of mention:

1. Go through vector and multiply even numbers with -1
2. Sort the vector in ascending order
3. Go through vector again and multiple negative even numbers with -1

This appears to work with the given example which only listed positive numbers. However the question stated that `A` is a vector of 32-bit signed integers. That means odd numbers can be negative too! In such cases the proposed method will not work. e.g. for `{-1,-2,2,1}`

Question C.2ー Comments

This question is basically testing about

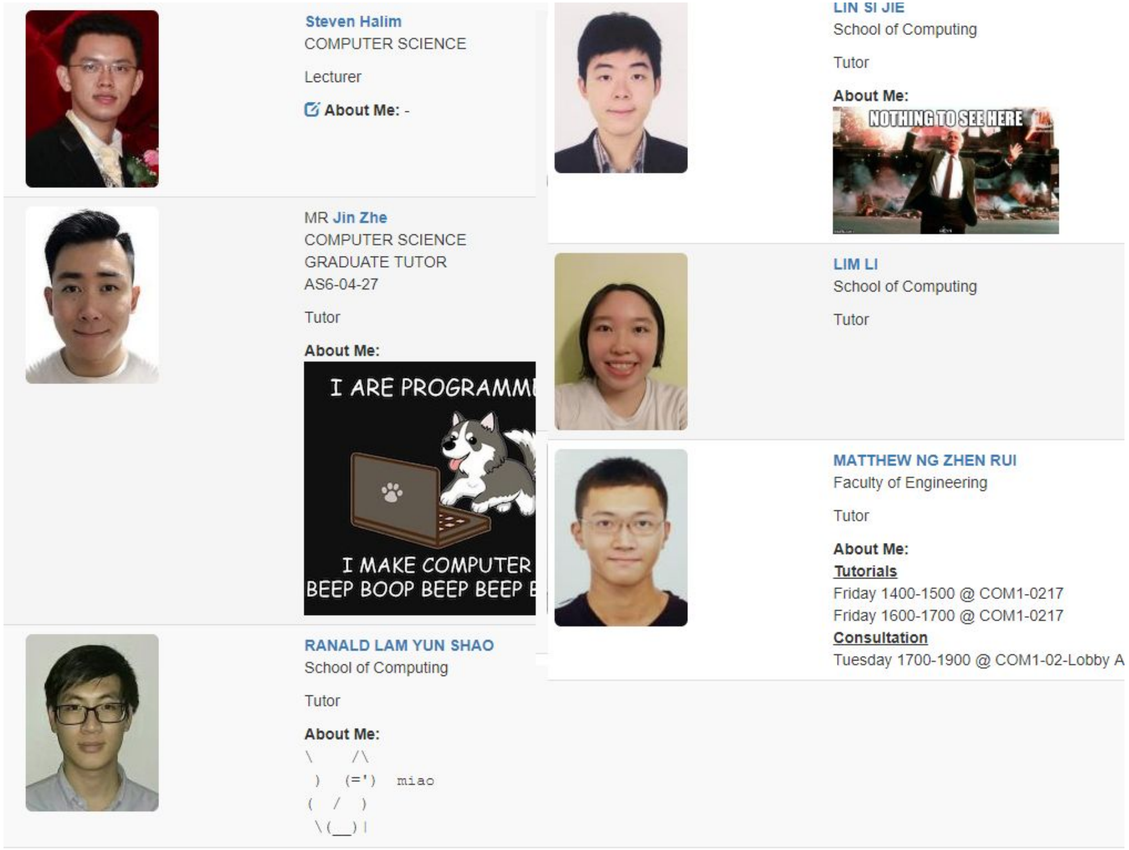
- Basic input: Specifically how to read in an entire line as string. Steven correctly predicted that many will do `cin >> TA[i]` (only first word of TA) instead of `getline(cin, TA[i])` (full TA name)
- Your understanding of `stable_sort` and when it is needed (heavy penalization if not used)
- Custom comparator (again) and how you may call other functions within it

Question D

Marked by Steven

PLEASE REMEMBER OUR (FULL) NAMES… (1)

Lecturer + 5 tutorial TAs



Steven Halim
COMPUTER SCIENCE
Lecturer
About Me: -

MR Jin Zhe
COMPUTER SCIENCE
GRADUATE TUTOR
AS6-04-27
Tutor
About Me:

RANALD LAM YUN SHAO
School of Computing
Tutor
About Me:

```
\    /\
 )  (=')  miao
(  /  )
 \(__)|
```

LIN SI JIE
School of Computing
Tutor
About Me:

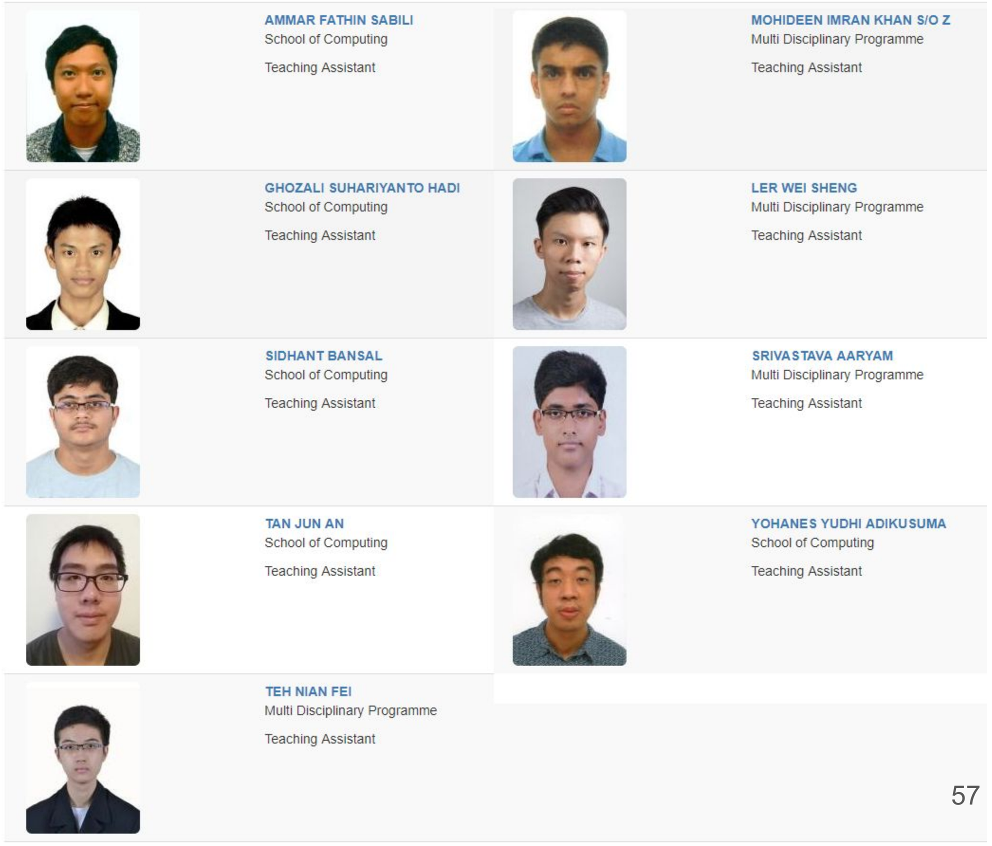
LIM LI
School of Computing
Tutor

MATTHEW NG ZHEN RUI
Faculty of Engineering
Tutor
About Me:
Tutorials
Friday 1400-1500 @ COM1-0217
Friday 1600-1700 @ COM1-0217
Consultation
Tuesday 1700-1900 @ COM1-02-Lobby A

PLEASE REMEMBER OUR (FULL) NAMES... (2)

8 Lab TAs + 1 grader (Yohanes)

BTW before you can complain,
all our names are already INSIDE
the question paper...



AMMAR FATHIN SABILI
School of Computing
Teaching Assistant

MOHIDEEN IMRAN KHAN S/O Z
Multi Disciplinary Programme
Teaching Assistant

GHOZALI SUHARIYANTO HADI
School of Computing
Teaching Assistant

LER WEI SHENG
Multi Disciplinary Programme
Teaching Assistant

SIDHANT BANSAL
School of Computing
Teaching Assistant

SRIVASTAVA AARYAM
Multi Disciplinary Programme
Teaching Assistant

TAN JUN AN
School of Computing
Teaching Assistant

YOHANES YUDHI ADIKUSUMA
School of Computing
Teaching Assistant

TEH NIAN FEI
Multi Disciplinary Programme
Teaching Assistant

The rest during lecture

All updated remarks and actionable changes in the second half will be mentioned by Steven during lecture on Week 07

Question E

Marked by Steven

Question E — Comments

- Nobody (out of 269) got full marks 10/10...
- The highest got 9/10 (a few students), very close to making RL-enqueue $O(1)$
    - But still $O(T)$...
    - Already spot the $O(1)$ mapping between `person_ID` to `team_ID`
        - Most use slow mapping in this part...
- About ⅓ leave this section totally blank,
    - Unfortunately for you but fortunately for me, this is… very easy to mark…

Question E — Common mistakes

- Wrong assumption: You **cannot** find `team_ID` by doing `person_ID/100` :O…
    - The sample test case so happens to show that for simplicity in illustration, but you cannot make such an assumption!
- The ⅔ who attempted mostly optimize RL-dequeue to $O(1)$ prematurely and have hard time making RL-enqueue respectable
    - Some are doing $O(T \log k)$, $O(Tk)$, $O(Ck)$, and any other high time complexity :O
        - T up to 1000, k up to 1000, C up to $200{,}000$

Questions?