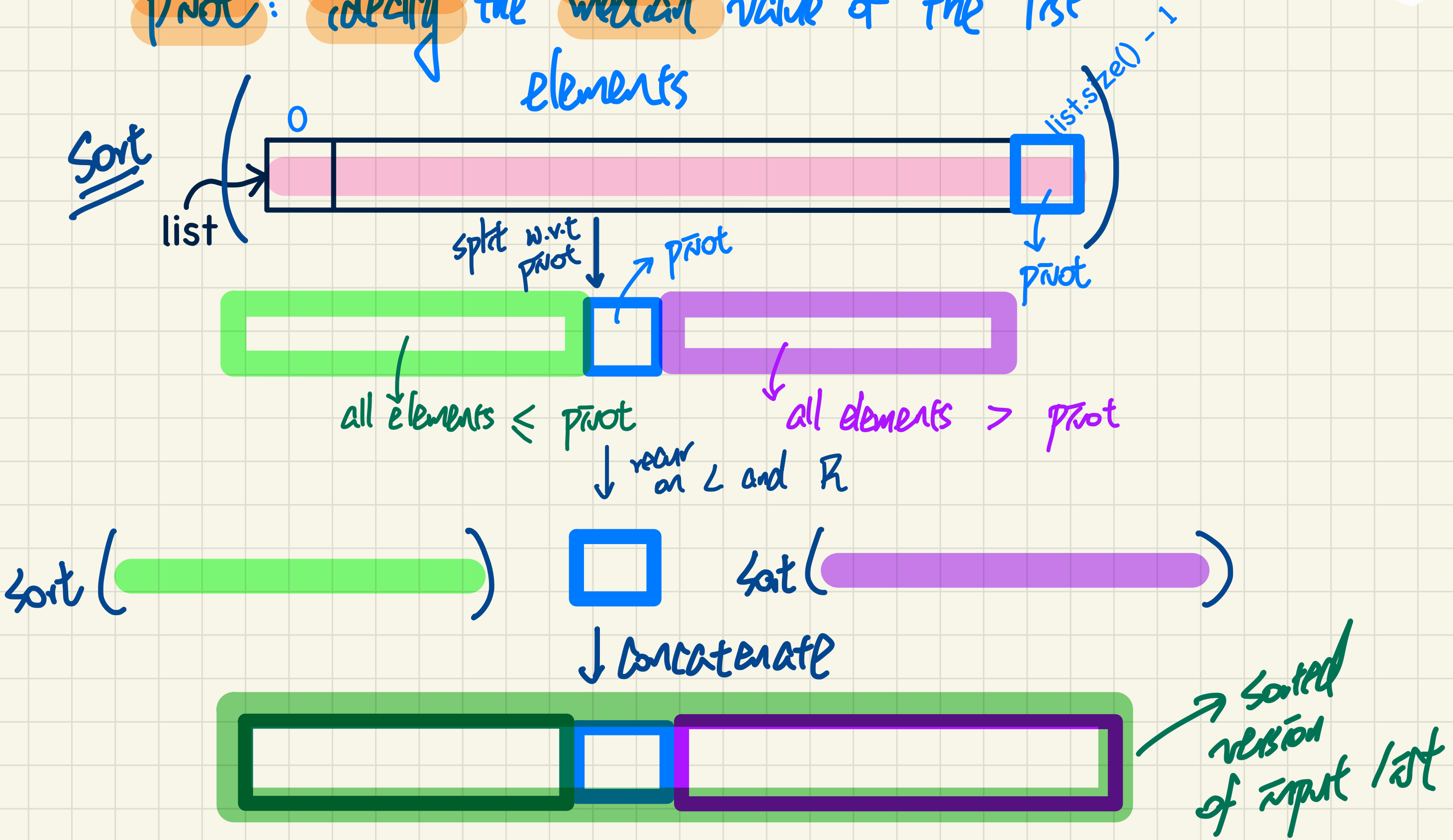
<u>Lecture 4</u>

<u>Part D</u>

Examples on Recursion
Quick Sort

Quick Sort: Ideas

pivot: ideally the median value of the list elements



Quick Sort in Java

```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue = list.get(pivotIndex);
    List<Integer> left = allLessThanOrEqualTo(pivotIndex, list);
    List<Integer> right = allLargerThan(pivotIndex, list);
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = new ArrayList<>();
    sortedList.addAll(sortedLeft);
    sortedList.add(pivotValue);
    sortedList.addAll(sortedRight);
  }
  return sortedList;
}
```

base cases $O(1)$

$O(1)$ (pivotIndex, pivotValue)

$O(n)$ (left, right)

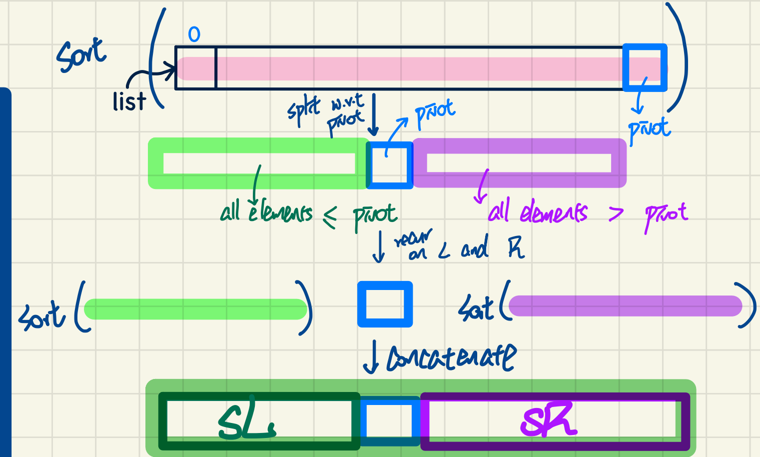
$O(1)$ (sortedList construction)

1. Best case: pivot is s.t. $|L| \approx |R|$

2. Worst case: e.g. $|L| = n-1$, $|R| = 0$

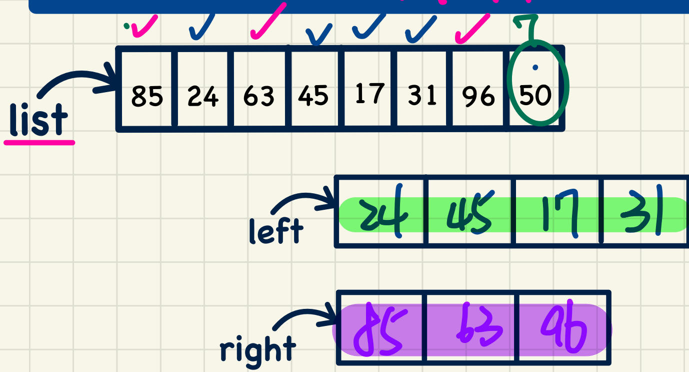
$|L| \ll |R|$ or $|R| \ll |L|$

$O(n)$




```
List<Integer> allLessThanOrEqualTo(int pivotIndex, List<Integer> list) {
  List<Integer> sublist = new ArrayList<>();
  int pivotValue = list.get(pivotIndex);
  for(int i = 0; i < list.size(); i ++) {
    int v = list.get(i);
    if(i != pivotIndex && v <= pivotValue) { sublist.add(v); }
  }
  return sublist;
}
List<Integer> allLargerThan(int pivotIndex, List<Integer> list) {
  List<Integer> sublist = new ArrayList<>();
  int pivotValue = list.get(pivotIndex);
  for(int i = 0; i < list.size(); i ++) {
    int v = list.get(i);
    if(i != pivotIndex && v > pivotValue) { sublist.add(v); }
  }
  return sublist;
}
```

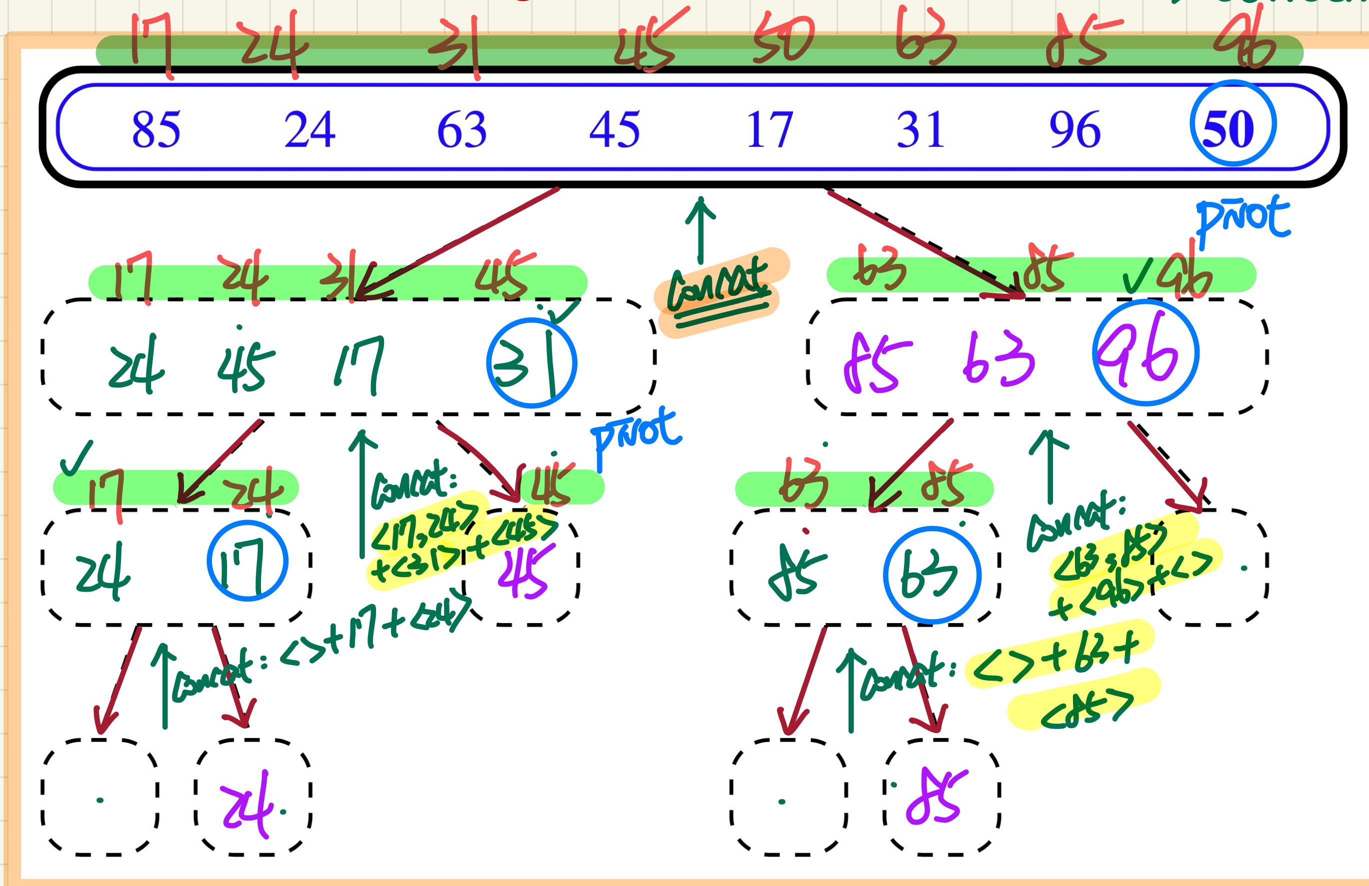
$O(n)$



Quick Sort: Tracing

split

concatenate



Quick Sort: Worst-Case Running Time

```java
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue = list.get(pivotIndex);
    List<Integer> left = allLessThanOrEqualTo(pivotIndex, list);
    List<Integer> right = allLargerThan(pivotIndex, list);
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = new ArrayList<>();
    sortedList.addAll(sortedLeft);
    sortedList.add(pivotValue);
    sortedList.addAll(sortedRight);
  }
  return sortedList;
}
```

$O(n)$

| 9 | 7 | 5 | 3 | 1 |

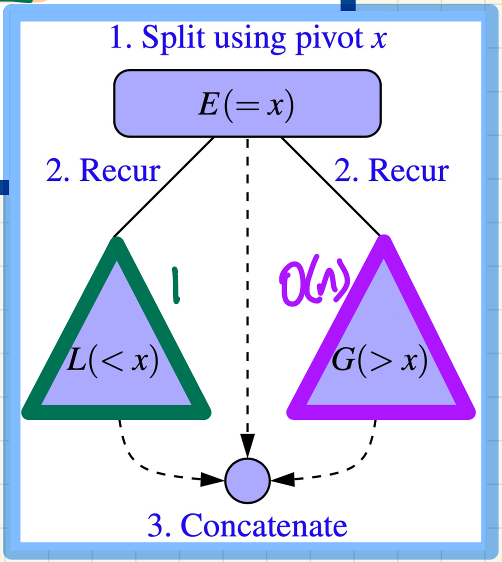
1 | 9 | 7 | 5 | 3 |

1 | 9 | 7 | 5 |

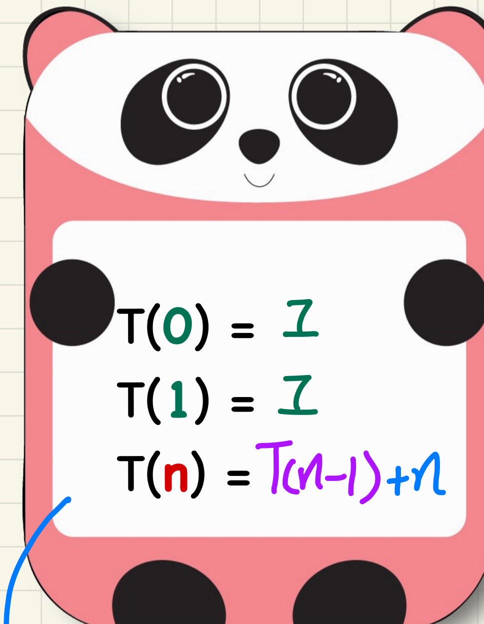
1 | 9 | 7 |

1 | 9 |

\# splits: 4 $O(n)$



Running Time as a Recurrence Relation



$T(0) = 1$

$T(1) = 1$

$T(n) = T(n-1) + n$

Exercise: Solve by unfolding

Quick Sort: Best-Case Running Time

```java
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue = list.get(pivotIndex);
    List<Integer> left = allLessThanOrEqualTo (pivotIndex, list);
    List<Integer> right = allLargerThan (pivotIndex, list);
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = new ArrayList<>();
    sortedList.addAll(sortedLeft);
    sortedList.add(pivotValue);
    sortedList.addAll(sortedRight);
  }
  return sortedList;
}
```

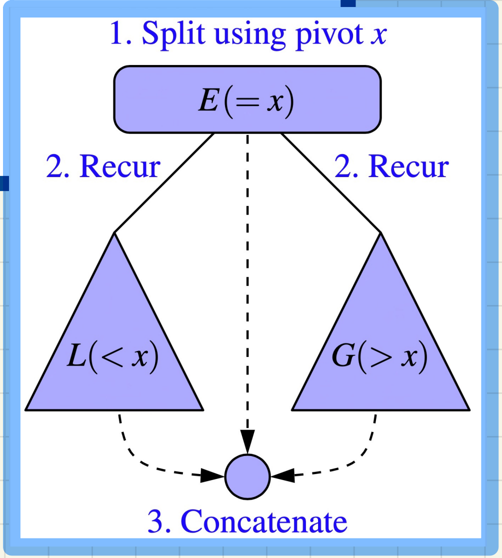
$O(1)$

$O(n)$

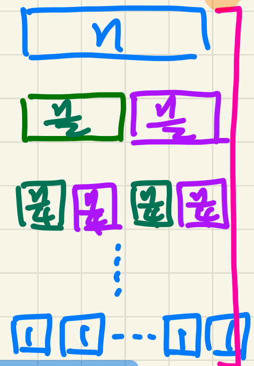
median

$\leq$ $>$

sizes equal



$\log_2 n$



Running Time as a Recurrence Relation

$T(0) = 1$

$T(1) = 1$

$T(n) = 2\,T(\frac{n}{2}) + n$

Ex. 2

Exercise 1: Solve by Unfolding.

$T(0) = 1$

$T(1) = 1$

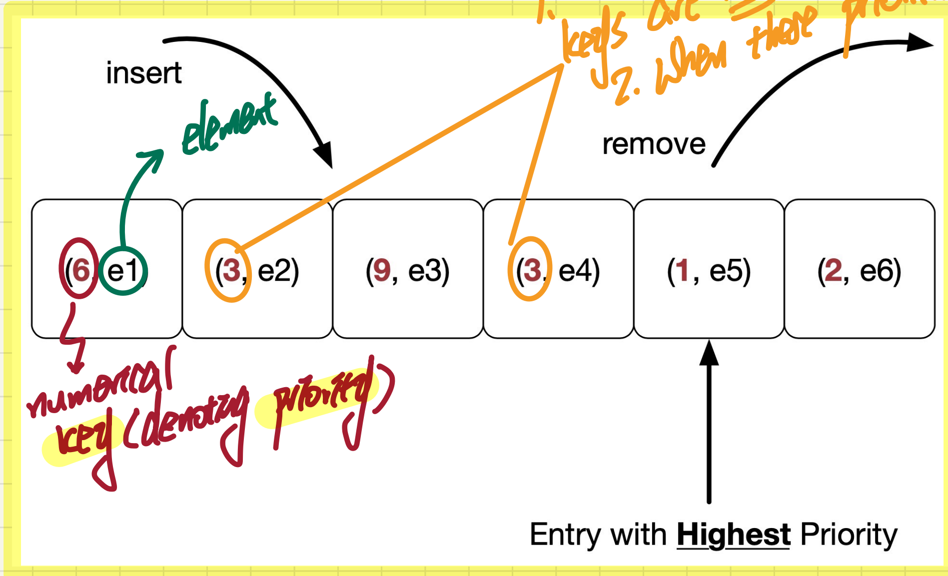
$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + n$

Lecture 5d

Part A

Priority Queue -
Intro & List-Based Implementations

What is a Priority Queue (PQ)



1. keys are not for searching purpose
2. When there priorities become the highest, it does not matter which entry is chosen.

Compare:

1. FIFO q. remove element based on chronological order of insertion.
2. PQ. remove element based on priorities

1. In PQ, no notions of "front" or "end" of the q.
2. The lower the key value of an entry, the higher its priority is.
   ↳ the entry with the minimum key value has the highest priority.

List-Based Implementations of Priority Queue (PQ)

PQ Method	List Method — A1: SORTED LIST		A2: UNSORTED LIST	
size	list.size $O(1)$			
isEmpty	list.isEmpty $O(1)$			
min	list.first	$O(1)$	search min	$O(n)$
insert	insert to "right" spot	$O(n)$	insert to front	$O(1)$
removeMin	list.removeFirst	$O(1)$	search min and remove	$O(n)$

≈ insertion sort

≈ selection sort

e.g. if app has more frequent calls to min/insert

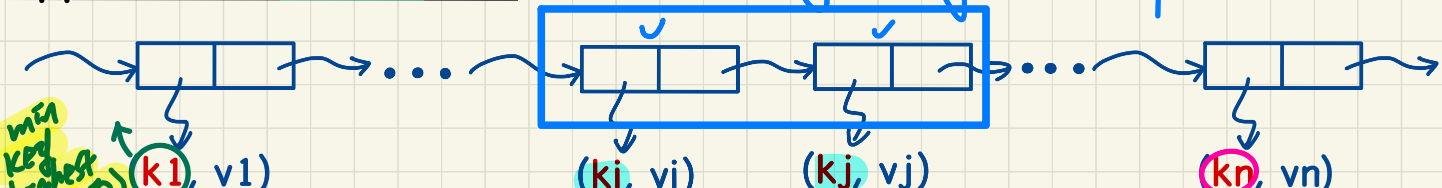
"min" ⇒ choose A1

Exercise

Use DLL for A1 & A2. Does it make a difference to RT?

$k_i \le k_j$ (entry i "more important" than entry j)

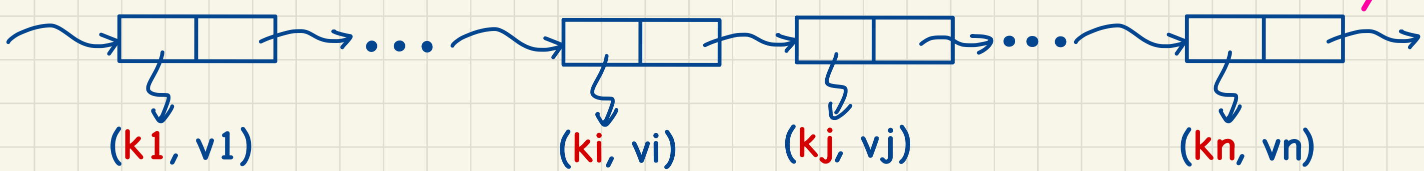
Approach 1: Sorted List



min key (highest prio.)

max key (lowest priority)

Approach 2: Unsorted List



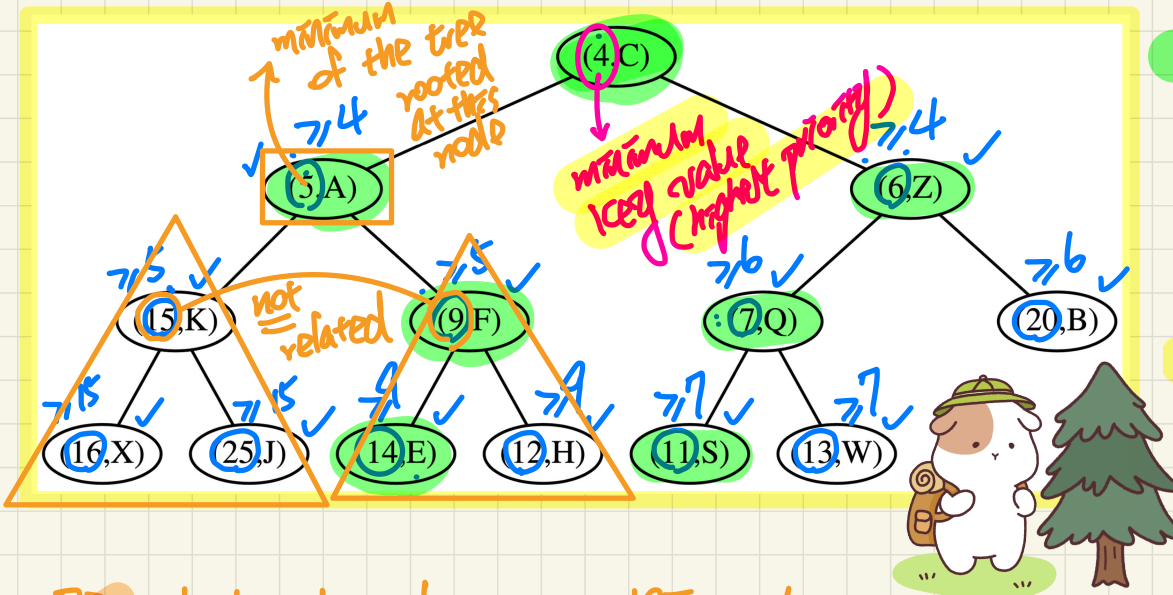
Lecture 5d

Part B

Heaps -
Examples and Properties

Heaps: Relational Properties of Keys

Property: Each non-root node **n** is s.t. **key**(**n**) ≥ **key**(parent(**n**))



P1. Any leaf-to-root path has a sorted seq of keys.

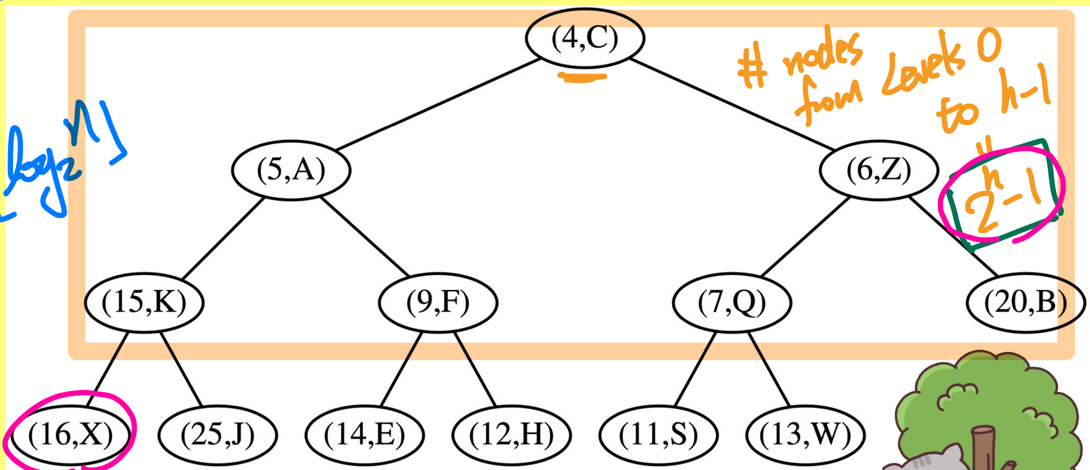
P2. the minimum key exists in the root entry.

P3. key values between LST and RST are <u>not</u> related.

Heaps: Structural Properties of Nodes

Property: The tree is a complete Binary Tree

$h = \lfloor \log_2 n \rfloor$



$n = 13$

$\lfloor \log_2 n \rfloor = \lfloor 3.\cdots \rfloor = 3$

Min # of nodes : $(2^h - 1) + 1$

Max # of nodes : $(2^h - 1) + 2^h = 2^{h+1} - 1$

# of nodes at level $h = n - (2^h - 1)$

Example Heaps

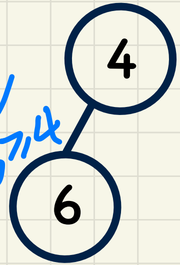
relational
structural

Example 1

✓

smallest possible one-node heap

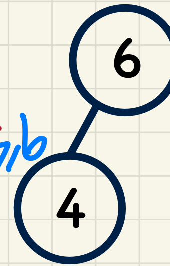
Example 2



✓

✓ $6 \geq 4$

Example 3

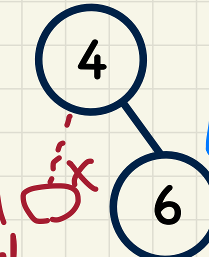


✗

✗ $4 \geq 6$

violating the relational property

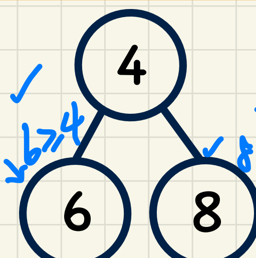
Example 4



✓ $6 \geq 4$

violating structural property

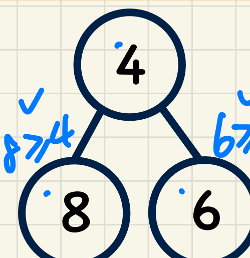
Example 5



✓ $6 \geq 4$

✓ $8 \geq 4$

Example 6



✓ $8 \geq 4$

✓ $6 \geq 4$

✓

full BTs

$\Rightarrow$ complete BTs

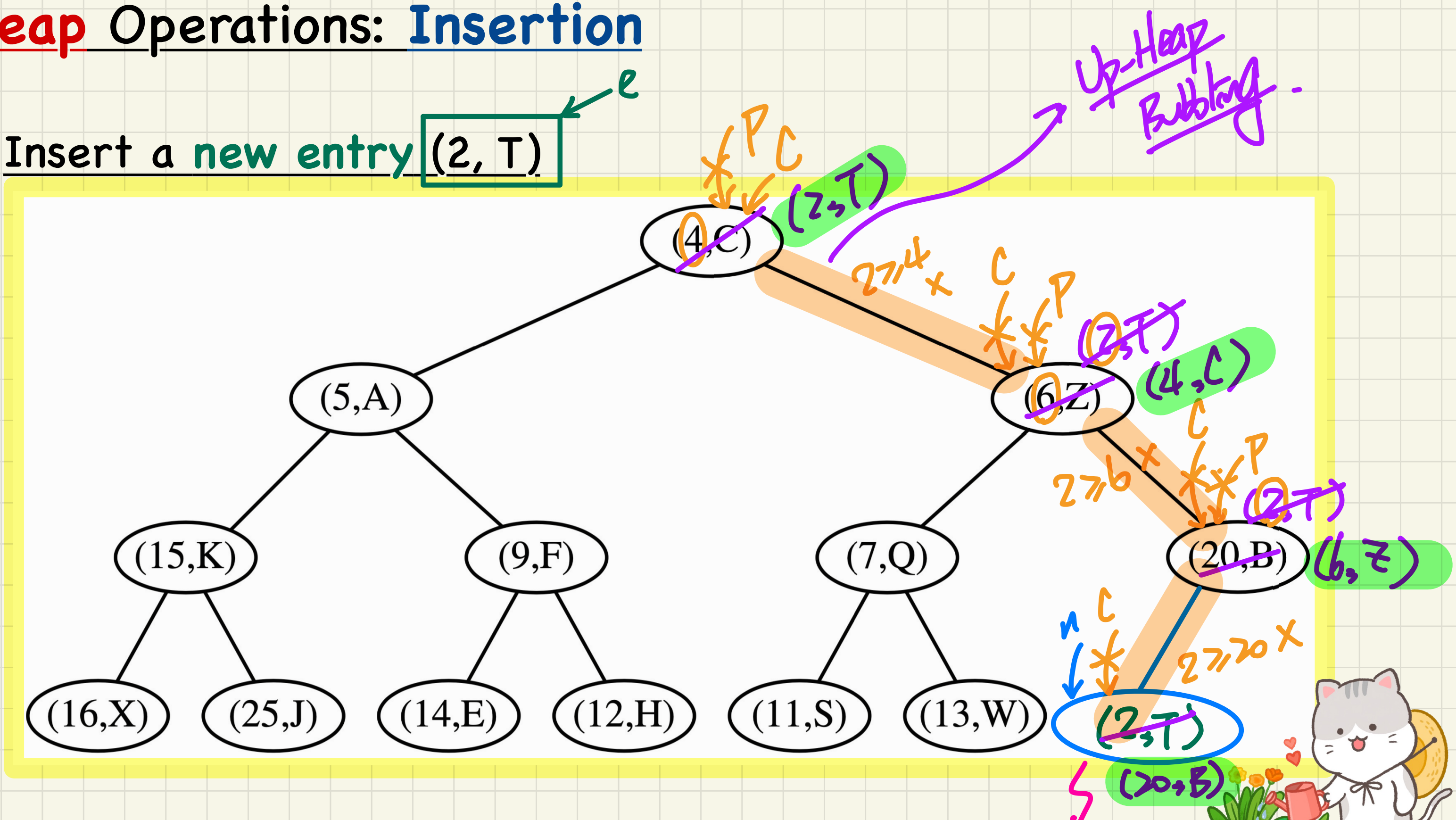
Lecture 5d

Part C

Heaps - Insertions

Heap Operations: Insertion

Insert a new entry $\boxed{(2, T)}$ e



must be right-most at Level h in order to preserve structural property

Lecture 5d

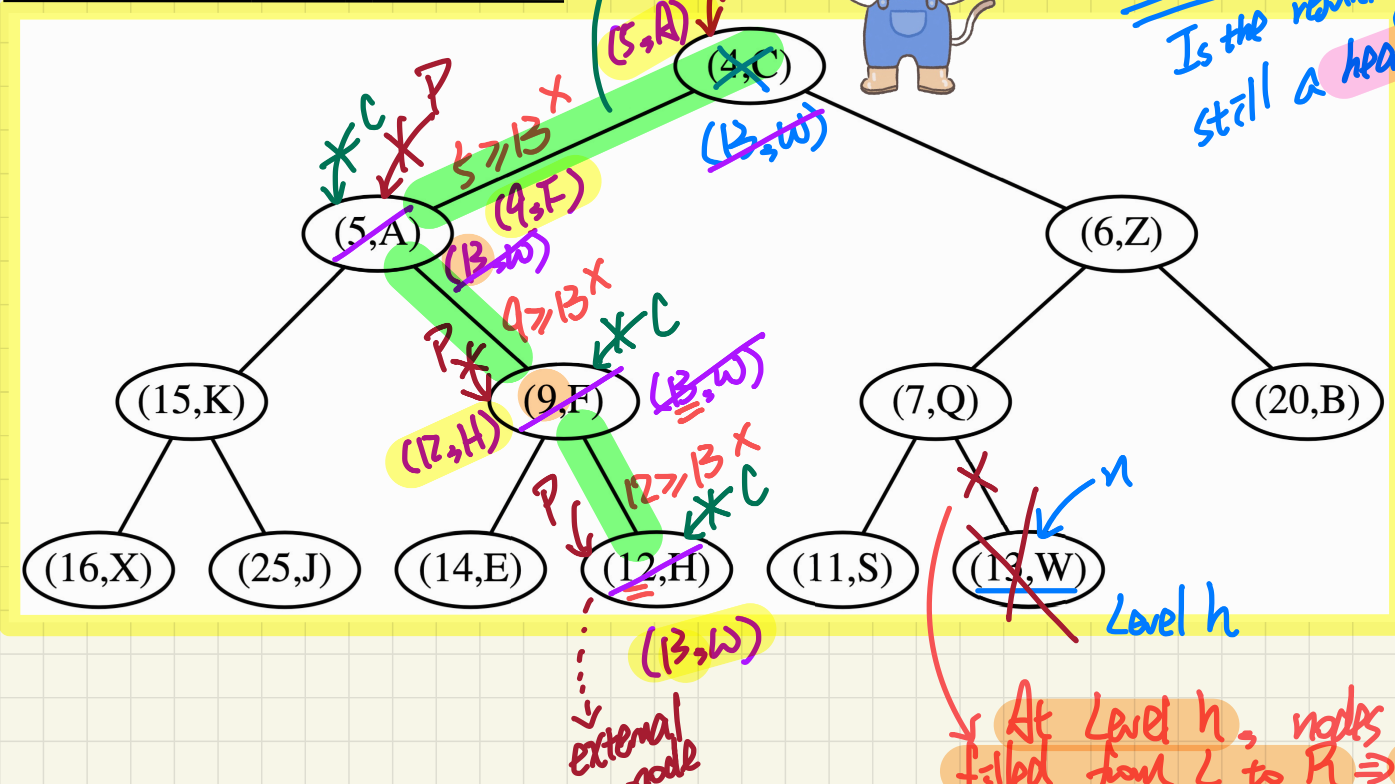
Part D

Heaps - Deletions

Heap Operations: Deletion

root-to-leaf path (down-heap bubbling)

Delete the root/minimum



Exercise
Is the resulting tree still a heap?

At Level h, nodes are still filled from L to R ⇒ complete BT

Lecture 5d

Part E

Heaps -
Top-Down Heap Construction

Top-Down Heap Construction

Problem: Build a heap out of ***N*** entires, supplied one at a time.

- Initialize an ***empty heap h***.
- As each new entry **e** = (k, v) is supplied, **insert e** into ***h***.

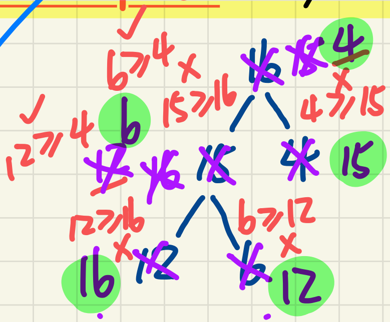
first inserted to level 2

Exercise: Build a heap out of the following 15 keys:

<16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14>

Assumption: Key values supplied one at a time.

first inserted to level 1



RI: # nodes at level · # upheap bubbling steps

$$* \quad \underbrace{1}_{\text{root}} + 2^1 \cdot 1 \leq \log n$$

$$+ 2^2 \cdot 2 \leq \log n$$

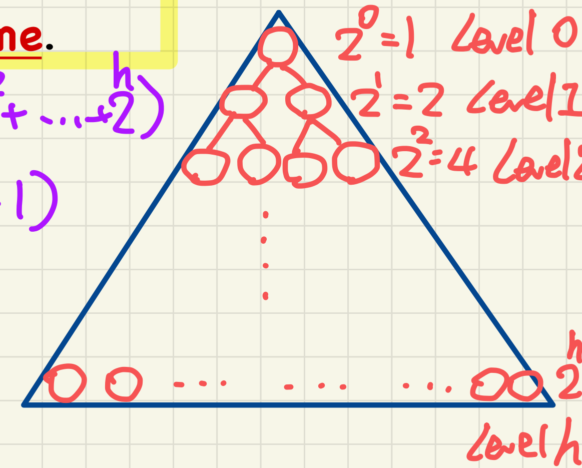
$$+ \cdots$$

$$+ 2^h \cdot h \quad \log n$$

$$* \leq 1 + \log_2 n \cdot (2^1 + 2^2 + \cdots + 2^h)$$

$$= 1 + \log_2 n \,(n - 1)$$

$$O(n \cdot \log n)$$



Exercise: Complete inserting the remaining keys to the heap.

Lecture 5d

Part F

Heaps -
Bottom-Up Heap Construction

Bottom-Up Heap Construction

Problem: Build a heap out of N entires, supplied all at once.

- **Assume**: The resulting heap will be *completely filled* at **all** levels.
  $\Rightarrow N = 2^{h+1} - 1$ for some *height* $h \geq 1$ [$h = (log\ (N + 1)) - 1$]
  15. 3.
- Perform the following steps called *Bottom-Up Heap Construction*:

  Step 1: Treat the first $\boxed{\frac{N+1}{2^1}}$ list entries as heap roots. 8

  $\therefore \frac{N+1}{2^1}$ heaps with height 0 and size $\boxed{2^1 - 1}$ constructed.

  Step 2: Treat the next $\boxed{\frac{N+1}{2^2}}$ list entries as heap roots. 4

  ◇ Each *root* sets two heaps from **Step 1** as its *LST* and *RST*.

  ◇ Perform *down-heap bubbling* to restore **HOP** if necessary.

  $\therefore \frac{N+1}{2^2}$ heaps, each with height 1 and size $\boxed{2^2 - 1}$, constructed. 3

  ...

  $\frac{16}{2^{3+1}} = 1$

  Step $h + 1$: Treat next $\boxed{\frac{N+1}{2^{h+1}}} = \frac{(2^{h+1}-1)+1}{2^{h+1}} = 1$ list entry as heap root.

  ◇ Each *root* sets two heaps from **Step h** as its *LST* and *RST*.

  ◇ Perform *down-heap bubbling* to restore **HOP** if necessary.

  $\therefore \boxed{\frac{N+1}{2^{h+1}} = 1}$ heap, each with height h and size $\boxed{2^{h+1} - 1}$ constructed. $2^{3+1} - 1 = 15.$

Step 3: # heaps: ① $\frac{N+1}{2^3}$

② size of each heap: $2^3 - 1 = 7$

③ height of each heap: 2

Exercise: Build a **heap** out of the following 15 keys:

<16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14>

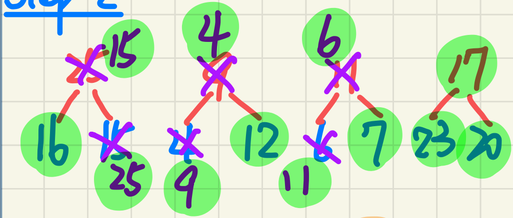
Assumption: Key values supplied all at once.

50% Step 1

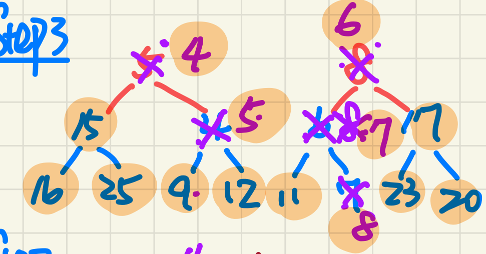
8 heaps, size 1, height 0

16 15 4 12 6 7 23 20

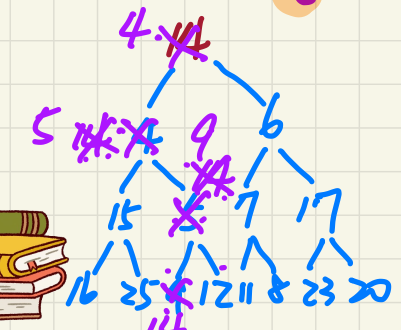
Step 2 25%



Step 3 12.5%



Step



Lecture 5d

Part G

Heaps -
Heap Sort Algorithm

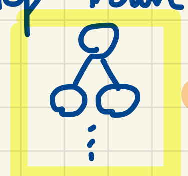
Heap Sort: Ideas

$O(N \cdot \log N)$

N entries

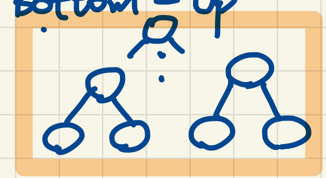
Construct a heap out of N entries

(A) Top-Down



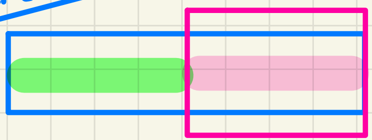
$O(N \cdot \log N)$

(B) Bottom-Up



$O(N)$

Selection Sort

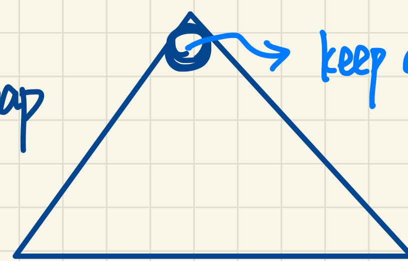


select the min from unsorted portion & put it to the front/end of the list

≈ Selection Sort

exploit the HOP (relational property): root stores entry with min key

heap



keep deleting the root until the heap is empty.

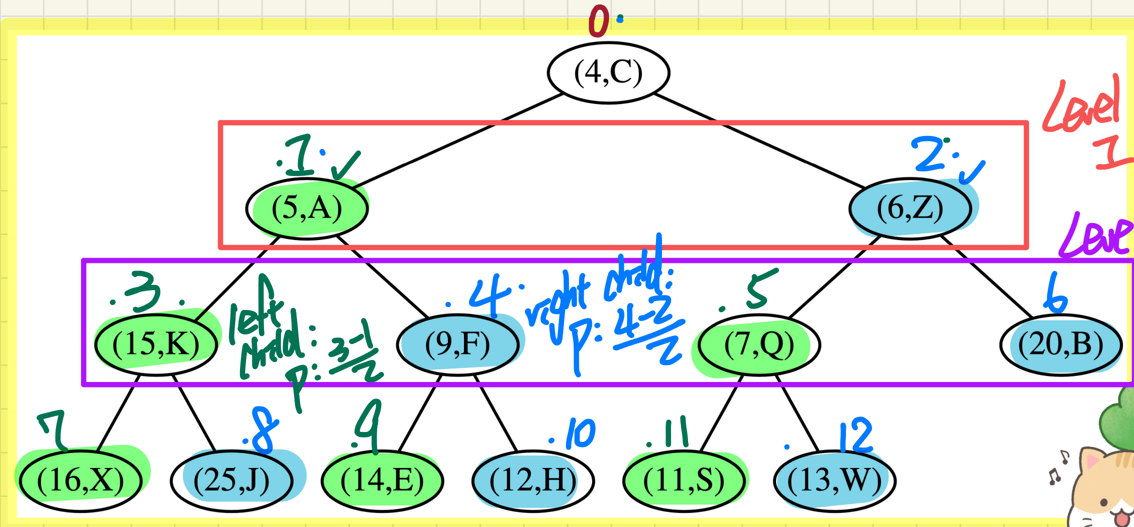
N deletions, each $O(\log N) \Rightarrow O(N \cdot \log N)$

Lecture 5d

Part H

Heaps -
Array-Based Implementation

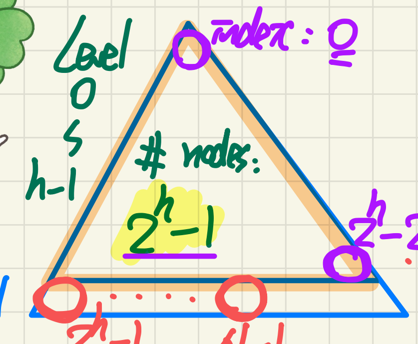
Array-Based Representation of a Complete BT



Exercise

What if the BT is not complete?

(bad for space util.)



$$
index(x) = \begin{cases} \underline{0} & \text{if } x \text{ is the root} \\ \underline{2 \cdot index(\ parent(x)\ ) + 1} & \text{if } x \text{ is a left child} \\ \underline{2 \cdot index(\ parent(x)\ ) + 2} & \text{if } x \text{ is a right child} \end{cases}
$$