4DS3: Data Structures and Algorithms Spring/Summer 2023
Challenge Project

1. Suppose an application uses only three operations: insert(), find(), and remove(). Under what circumstances would you use a binary heap instead of an unordered array?

With a normal queue, it is used first in first out. To insert(), find(), or delete() an element from a normal queue, the **worst case** for them can be Θ**(n)**.
The binary heap can be efficient in Priority Queue for insert(), find(), and remove() operations.
A priority can be a smaller number of higher priorities or a higher number of high priorities. Applying **min heap to solve a higher number of high priorities**, and applying **max heap to solve a smaller number of higher priorities**.

insert(): It is hard to insert elements into the unordered array because of the limited space. Even for insertion using a link list, it still needs to sort the array before the element insert.
To perform the insertion of a binary tree, the element is added to the last empty space of the array, which is the child node (left child of A[i]=A[2i+1], right child of A[i]=A[2i+2], if n=0). If it is running Max heap and the inserted element is larger than the element at the parent node, the inserted element will be swaped to the parent node and repeat the heapfiy until sorted. Therefore, **the maximum run time** for an element insertion is the height of the tree**(**Θ**(logn)**). If there is nothing to swap, the **min run time** is Θ**(1)**.

Remove(): The remove operation is to remove the root element by replacing the last element of the heap. Then, sorted downward by heapify. The **max run time** is the height of the tree**(**Θ**(lgn)** and the **min runtime** is Θ**(1)**. In addition, the heap size decreased, and keep it as free space.
However, the unordered array will still remain the same space size.

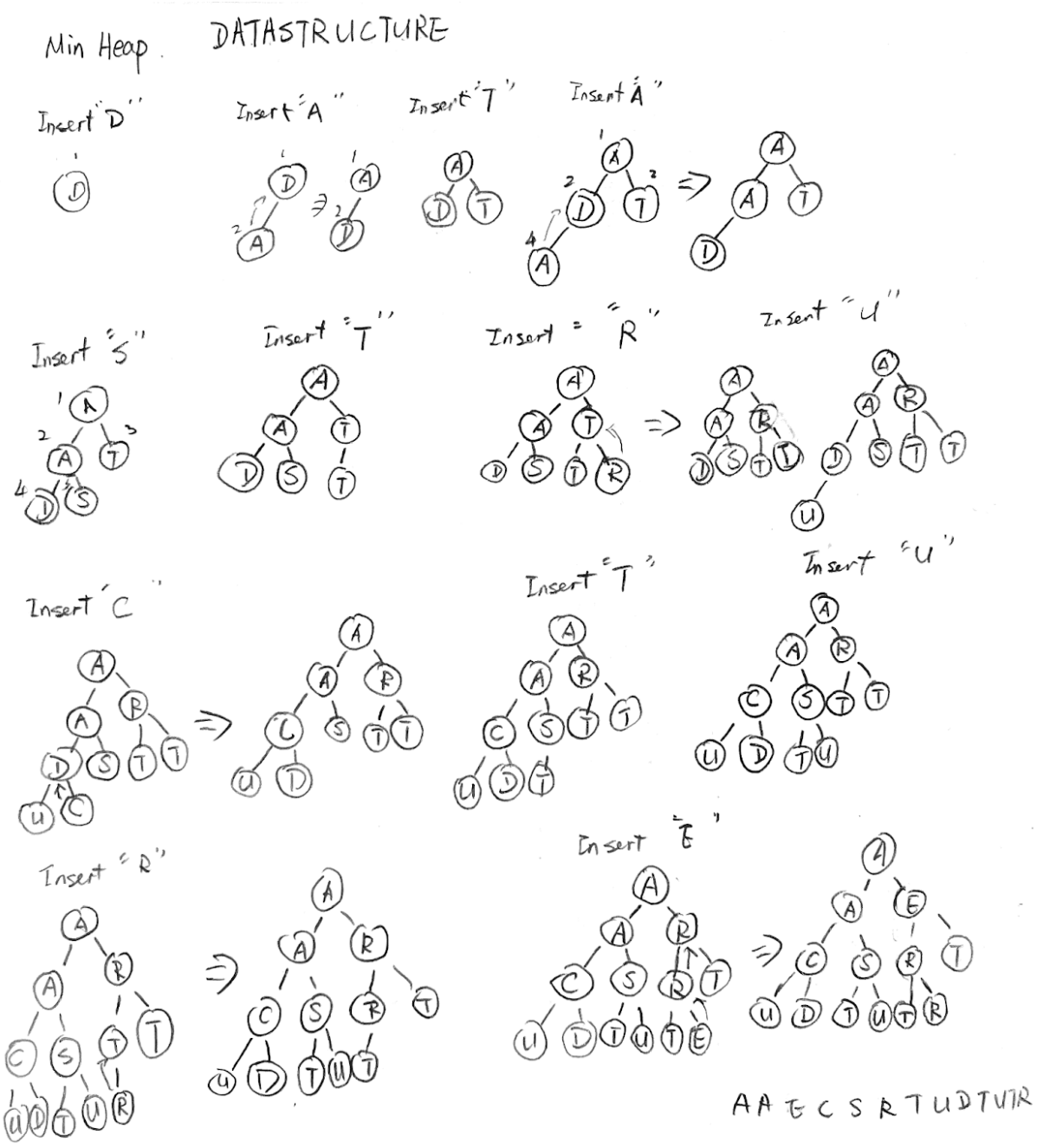
Find(): The **best case** of the find() operation is Θ**(1)**. The **worst case** will be finding all the elements from the whole heap, which is Θ**(n)**.

	Best Case	Average Case	Wrost Case
Insert()	$\Theta(1)$	$\Theta(lgn)$	$\Theta(lgn)$

find()	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Delete()	$\Theta(1)$	$\Theta(nlgn)$	$\Theta(nlgn)$

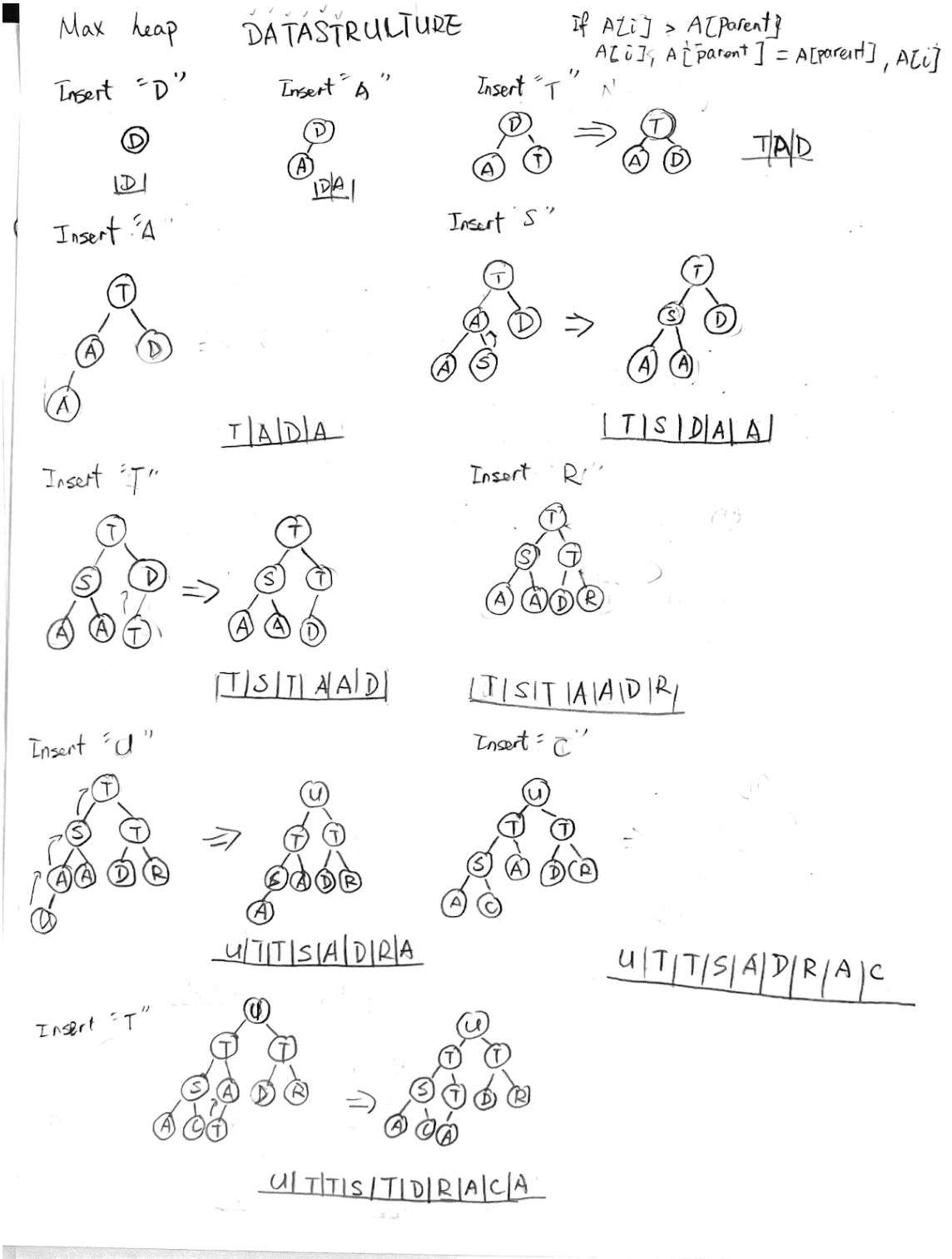
2. Suppose we wish to create a binary heap containing the keys D A T A S T R U C T U R E. (All comparisons use alphabetical order.)

a. Show the resulting min-heap if we build it using successive insert() operations (starting from D).

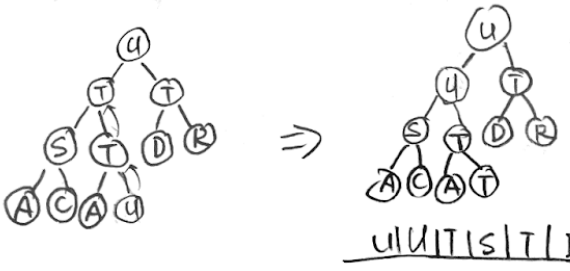


['D']
['A', 'D']
['A', 'D', 'T']
['A', 'A', 'T', 'D']
['A', 'A', 'T', 'D', 'S']
['A', 'A', 'T', 'D', 'S', 'T']
['A', 'A', 'R', 'D', 'S', 'T', 'T']
['A', 'A', 'R', 'D', 'S', 'T', 'T', 'U']
['A', 'A', 'R', 'C', 'S', 'T', 'T', 'U', 'D']
['A', 'A', 'R', 'C', 'S', 'T', 'T', 'U', 'D', 'T']
['A', 'A', 'R', 'C', 'S', 'T', 'T', 'U', 'D', 'T', 'U']
['A', 'A', 'R', 'C', 'S', 'R', 'T', 'U', 'D', 'T', 'U', 'T']
['A', 'A', 'E', 'C', 'S', 'R', 'T', 'U', 'D', 'T', 'U', 'T', 'R']

b. Show the resulting max-heap if we build it using successive insert() operations (starting from D).

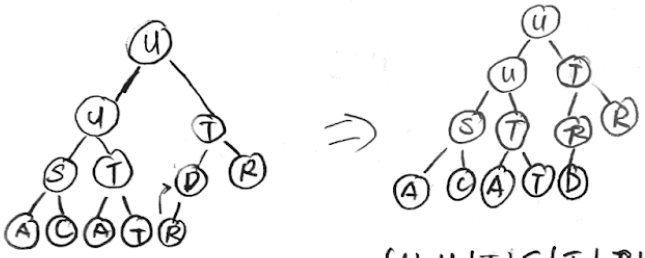


Insert "U"



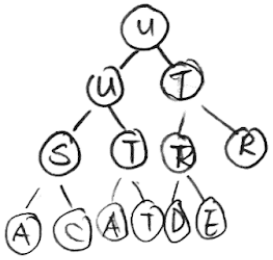
U|U|T|S|T|D|R|A|C|A|T

Insert "R"



U|U|T|S|T|R|R|A|C|A|T|D

Insert "E"

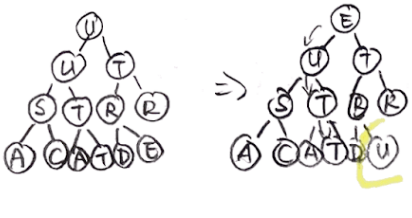


U|U|T|S|T|R|R|A|C|A|T|D|E

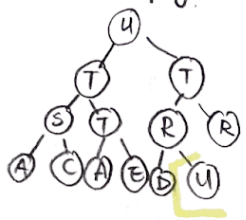
['D']
['D', 'A']
['T', 'A', 'D']
['T', 'A', 'D', 'A']
['T', 'S', 'D', 'A', 'A']
['T', 'S', 'T', 'A', 'A', 'D']
['T', 'S', 'T', 'A', 'A', 'D', 'R']
['U', 'T', 'T', 'S', 'A', 'D', 'R', 'A']
['U', 'T', 'T', 'S', 'A', 'D', 'R', 'A', 'C']
['U', 'T', 'T', 'S', 'T', 'D', 'R', 'A', 'C', 'A']
['U', 'U', 'T', 'S', 'T', 'D', 'R', 'A', 'C', 'A', 'T']
['U', 'U', 'T', 'S', 'T', 'R', 'R', 'A', 'C', 'A', 'T', 'D']
['U', 'U', 'T', 'S', 'T', 'R', 'R', 'A', 'C', 'A', 'T', 'D', 'E']

c. Run the algorithm for HeapSort on your max-heap. Show your working, the heap structure at each iteration as well as the final output array.

Heap sort
A[1] ⟷ A[13]



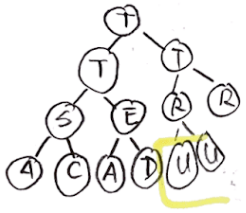
Max Heapify[A, 12]



UTTSTRRACAED | U

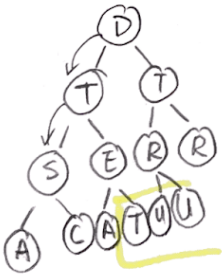
A[1] ⟷ A[12]

Max Heapify (A, 11)

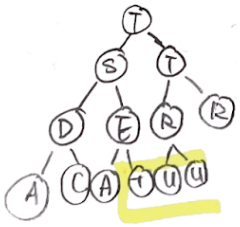


TTTSERRACAD | UU

A[1] ⟷ A[11]

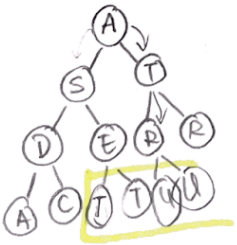


Max Heapify [A, 10]

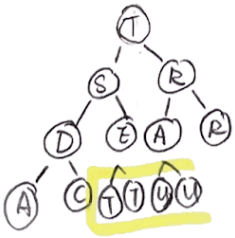


TSTDERRACA | TUU

A[1] ⟷ A[10]



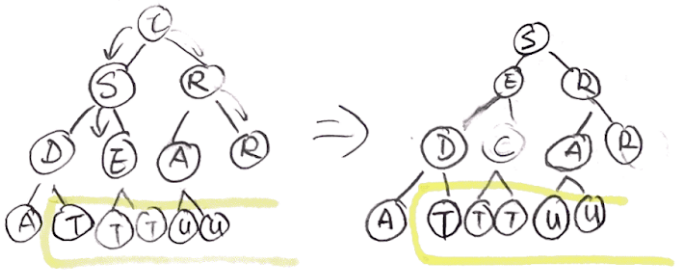
MaxHeapify (A, 9)



TSRDEARAC | TTUU

A[1] ⟷ A[9]

Max heapify (A, 8)



SERDCARA | TTTUU

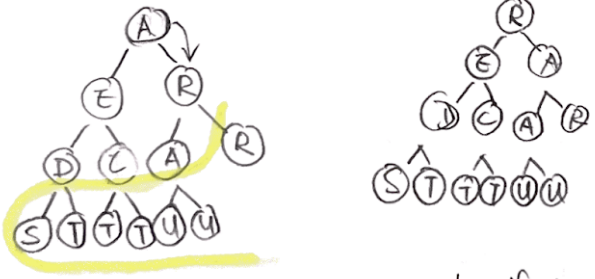
A[1] ⟷ A[8]

Max heapify (A, 7)

RERDCAA | STTTUU

A[1] ⟷ A[7]

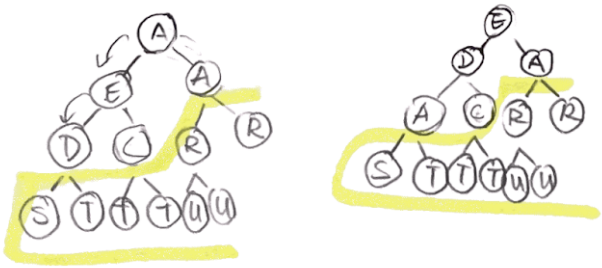
Max heapify (A, 6)



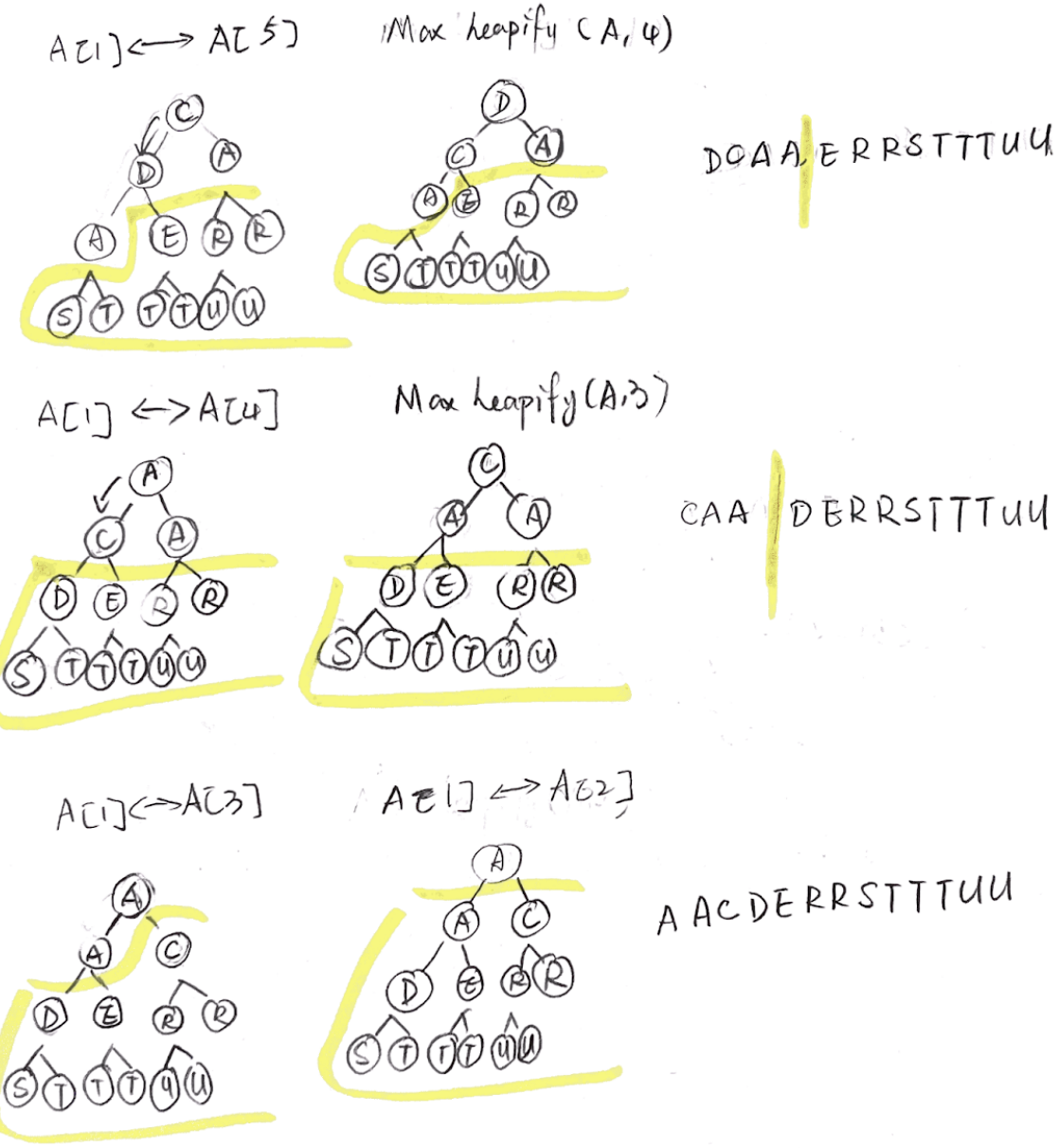
READCA | RSTTTUU

A[1] ⟷ A[6]

Max heapify (A, 5)



EDAAC | RRSTTTUU



Max Heap array:
UURSTTRACATDE
Heap Sort:
['U', 'T', 'T', 'S', 'T', 'R', 'R', 'A', 'C', 'E', 'A', 'D', 'U']
['T', 'T', 'T', 'S', 'E', 'R', 'R', 'A', 'C', 'A', 'D', 'U', 'U']
['T', 'S', 'T', 'D', 'E', 'R', 'R', 'A', 'C', 'A', 'T', 'U', 'U']
['T', 'S', 'R', 'D', 'E', 'A', 'R', 'A', 'C', 'T', 'T', 'U', 'U']
['S', 'E', 'R', 'D', 'C', 'A', 'R', 'A', 'T', 'T', 'T', 'U', 'U']
['R', 'E', 'R', 'D', 'C', 'A', 'A', 'S', 'T', 'T', 'T', 'U', 'U']
['R', 'E', 'A', 'D', 'C', 'A', 'R', 'S', 'T', 'T', 'T', 'U', 'U']

['E', 'D', 'A', 'A', 'C', 'R', 'R', 'S', 'T', 'T', 'T', 'U', 'U']
['D', 'C', 'A', 'A', 'E', 'R', 'R', 'S', 'T', 'T', 'T', 'U', 'U']
['C', 'A', 'A', 'D', 'E', 'R', 'R', 'S', 'T', 'T', 'T', 'U', 'U']
['A', 'A', 'C', 'D', 'E', 'R', 'R', 'S', 'T', 'T', 'T', 'U', 'U']
['A', 'A', 'C', 'D', 'E', 'R', 'R', 'S', 'T', 'T', 'T', 'U', 'U']

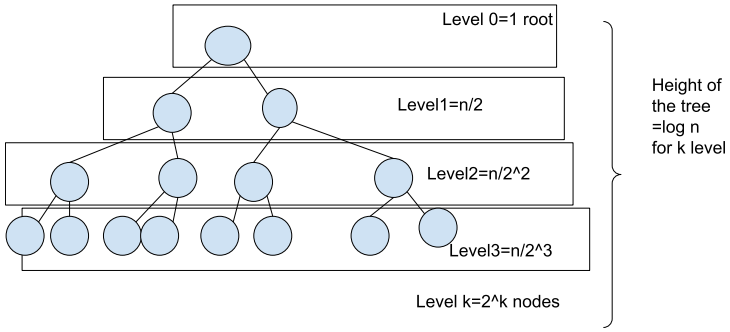
3. Analyse the various heap and heapsort algorithms and come up with:
i. an equation for the running time T(n), and,
ii. the order of growth for each
b. Insert() and find() for a heap. If you need to, design pseudocode for these.
c. MAX-HEAPIFY
d. BUILD-MAX-HEAP
e. HEAPSORT

<table>
<tr><td>Cost</td><td>Code</td><td>Calculation</td></tr>
<tr><td>C1</td><td>insert(arr, element)</td><td rowspan="9">T(n)=C1+C2+C3+C4+C5+ C6+C7+C8+T(n/2)
=C+T(n/2)

T(n)=T(n/2^k)+k

When n and k is big enough, n/2^k=1
k=logn
T(n)=T(1)+log n
Thus, order of growth= $\Theta(lgn)$</td></tr>
<tr><td>C2</td><td>arr.append(element)</td></tr>
<tr><td>C3</td><td>n=n+1</td></tr>
<tr><td>C4</td><td>Def heapify(arr,n, n-1):</td></tr>
<tr><td>C5</td><td>parent_index=(i-1)//2</td></tr>
<tr><td>C6</td><td>If arr[parent_index]>0:</td></tr>
<tr><td>C7</td><td>If arr[i]>arr[parent_index]:</td></tr>
<tr><td>C8</td><td>swap(arr[i],
arr[parent_index]</td></tr>
<tr><td>T(n/2)</td><td>heapify(arr,n,parent)</td></tr>
<tr><td colspan="3">For the wrost case, it needs to compare the root, level2, to the level k, which are the is the height of the binary tree. Thus, order of growth =$\Theta(lgn)$</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>C1</td><td>find(element)</td><td>T(n)=C1+n</td></tr>
</table>

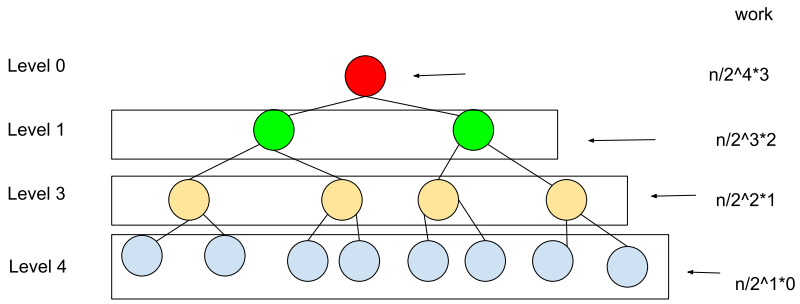
n	for i in range(0,n):	Thus the order of growth= Θ(n)
n	If element==arr[i]:	
n	Return true	
	Max_heapify(Arr, i, n):	
Constant	l=2*i+1	
	r=2*i+1	
	If l<=n and Arr[l]>Arr[i]	
	largest=l	
	Else largest=l	
	If r<=n and Arr[r]>Arr[i]	
	largest=r	
	If largest!=i	
	swap(Arr[i],Arr[largest])	
h=lg n	Max_heapify(Arr,i,n)	



Each parent nodes has two child nodes to compare with.
Thus, the high of the tree=lg n, the algorithm is Θ(lg n)

Build Max Heap	
n/2+1	For i=n/2 down to 1
n/2*Θ(log n)	Max-Heapify(A,i, n)
T(n)=n+n*h=Θ(nlgn)	

Visuallized the calculation	



(*How Can Building a Heap Be O(n) Time Complexity?*, n.d.)

Assume the k level of the calculation is n
Thus, T(n)=Θ(n*h)=Θ(n* log n)

To sum all work, we can get T(n)=n/2^1*0+n/2^2*1+n/2^3*2+n/2^4*3+n/(2^k+1)*k
T(n)=n*(0+¼*1+⅛*2+1/16*3+...+1/(2^k+1)*k)≃1
Thus, if the tree is large enough, T(n)=Θ(n)

Heapsort(A,n)	
Θ(n)/Θ(nlogn)	Build_Max_Heap(A,n)
n	For i=n downto 2
n-1	Exchange A[1] with A[i]
(n-1)*Θ(lgn)=Θ(n*lgn)	Max_Heapify(A,1,i-1)
T(n)=Θ(nlogn)+n+(n-1)+Θ(nlogn)=2Θ(nlgn)+Θ(n)=Θ(nlgn)	

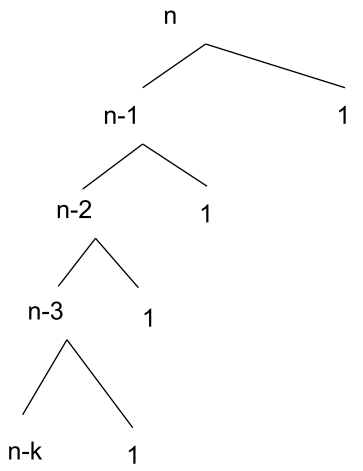
4. Quick Sort

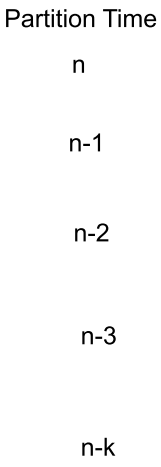
a. Is there any effect on the performance of QuickSort algorithm if there are repeated elements? Use this example to explain your answer: In the extreme case, if the whole list consists of the same element, what would be the approximate running time for QuickSort?

a.

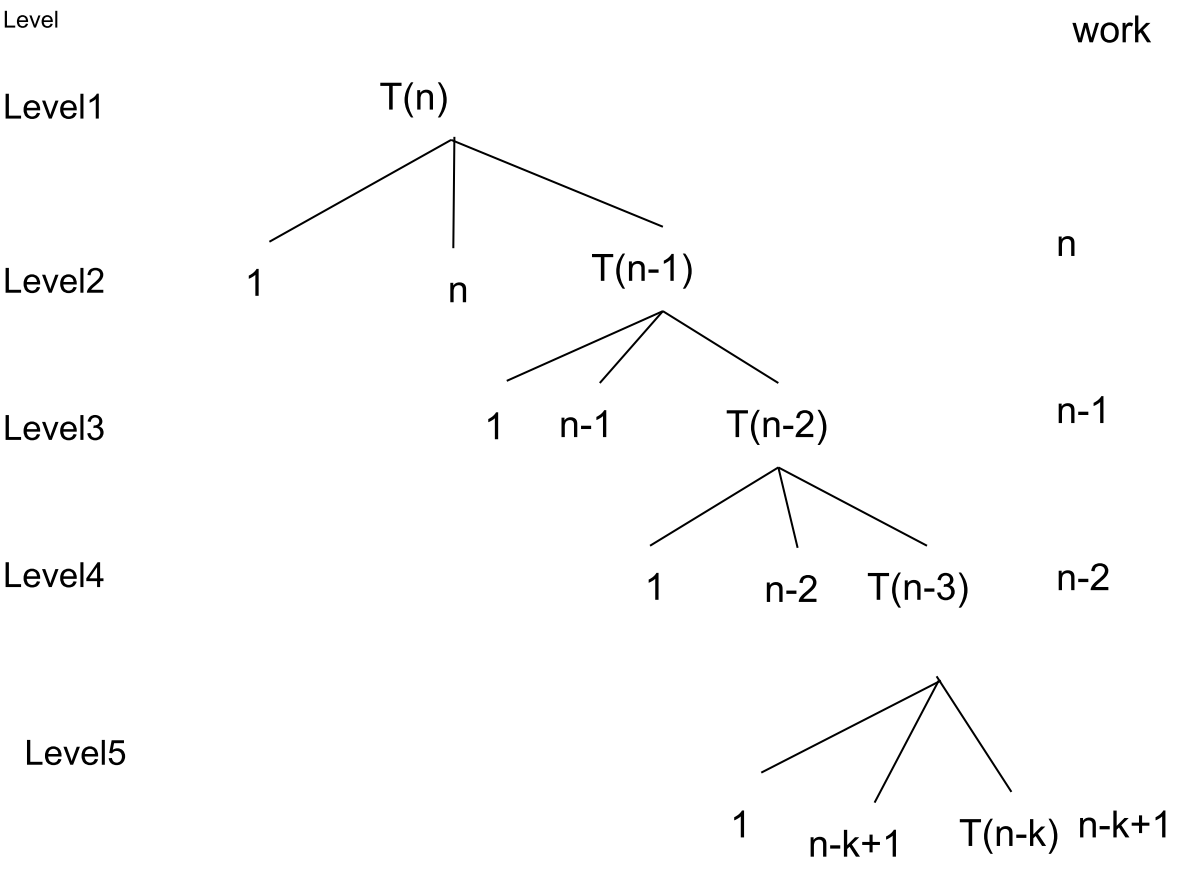
arr_index(i)	1	2	3	4	5	6
arr	1	1	1	1	6	6

partition:1
[1, 1, 1, 1, 1, 6, 6]
partition:2
[1, 1, 1, 1, 1, 6, 6]
partition:3
[1, 1, 1, 1, 1, 6, 6]
partition:4
[1, 1, 1, 1, 1, 6, 6]
patition:5
[1, 1, 1, 1, 1, 6, 6]
patition:6
[1, 1, 1, 1, 1, 6, 6]





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



When n, and k are large enough: T(1)=T(n-k), n=k, n-k+1=2

Sum all work:
n+(n-1)+(n-2)+...+(n-k+1)+2=n*(n+1)/2-1
Thus, T(n)=$\Theta(n^2)$
(Datta, 2022)

b. Let's say you're working on a dataset that you know will have a lot of duplicates. What would be a possible solution to the above issue?

There are two solutions to improve the quicksort:

1. Choose the pivot at the median. Best Case: T(n)=$\Theta(n) + 2T(n/2)$
   However, it is not easy to always pick the median to analyze the algorithm.
2. Pick a random key as a pivot, which is called Randomized-Quicksort:

The runtime is various, but T(n)=$\Theta(nlgn)$ eventually.

5. You are designing a To-Do list for your own personal use. The program's main job will be to track the list of tasks you have to do in your personal and professional life. You should be able to tick them off as you complete them. You should also be able to search them if needed. Which of the following data structures would you use to implement this? Why or why not?
a. Stack
b. Queue
c. Neither, please specify another.

The goal to design a To-Do list are:

1. Allow to add tasks as per the entered dates or due dates
   Queue: Enqueque() to add the tasks as per the entered dates or priorities,

2. Search first or last tasks
   Queue:peek()/front() check the first task
   rear() check the last task

3. Able to add or remove multiple lists
   Stack: push(), add the task list, and stack at the top as a new list
   pop(), if the new tasks, lists, or items are invalid and would like to be removed,
   pop() can remove the newest tasks/lists/items.

4. Able to insert the priority tasks to the To-Do list
   Use Heap to insert priority tasks to the queue.

5. When the task is completed, the task can be removed from the list
   Queue: Dequeue() remove the task from the top of the task list.
   When all tasks are completed and removed, use isNull() check if the tasks are all removed.

6. Able to undo or redo the tasks/lists entered or deleted.
   Stack: push()to add back the previous tasks.
   pop() to delete the latest insertion

In no more than 2-5 sentences for each question, please answer the following:
1. What do you find easy about learning data structures and algorithms?

Technically, learning new knowledge is not easy. Once I am familiar with some of the analyzing concepts, building a visualized graph of the algorithm becomes easier.

2. What do you need more help with?
I spend the majority of my time understanding how to develop a proper tree and analyze the data structures and algorithms. In addition, understanding how the data structures and algorithms run is difficult for me. Therefore, I searched for multiple resources online to learn each data structure and algorithm.

3. What is the most important thing you've learnt so far?
The most important thing to learn so far is analyzing the data structures and understanding how the algorithms work.

4. Are there any questions you are left with which were not dealt with in class?

Yes, I would like to know how should we implement each data structure to actual life problems. If there are some examples to show what we should apply to the actual cases, that will be good for us to use in our real life. Not just the knowledge in the book.