CS213 2024 Tutorial Solutions

CS213/293 UG TAs

2024

Contents

Tutorial 1

1. *The following is the code for insertion sort. Compute the exact worst-case running time of the code in terms of n and the cost of doing various machine operations.*

Algorithm 1: Insertion Sort Algorithm

```
   Data: Array A of length n
1  for j ← 1 to n − 1 do
2  |  key ← A[j];
3  |  i ← j − 1;
4  |  while i ≥ 0 do
5  |  |  if A[i] > key then
6  |  |  |  A[i + 1] ← A[i];
7  |  |  end
8  |  |  else
9  |  |  |  break;
10 |  |  end
11 |  |  i ← i − 1;
12 |  end
13 |  A[i + 1] ← key;
14 end
```

Solution: To compute the worst-case running time, we must decide the code flow leading to the worst case. When the if-condition evaluates to false, the control breaks out of the inner loop. The worst case would have the if-condition never evaluates false. This will happen when the input is in a strictly decreasing order.

Counting the operations for the outer iterator j (which goes from 1 to $n-1$). Consider for each outer loop iteration:

1. Consider the outer loop without the inner loop. Then it has 1 Comparison (Condition in for loop), 1 Increment (1 Assignment + 1 Arithmetic) (Updation in for loop), 3 Assignments (Line 2, 3, 11) and 2 Memory Accesses (A[j] and A[i+1]) and 1 Jump. Hence $(C + 2Ar + 4As + 2M + J)$
2. while loop runs for j times (as i goes from $j-1$ to 0). Without the if-else, the while loop still has to do 1 Comparison (while condition), 1 Decrement (1 Assignment + 1 Arithmetic) ($i \leftarrow i-1$), 1 Jump, every iteration. Hence $(C + As + Ar + J) * j$
3. The if condition is checked in all iterations of the while loop. It involves 1 Comparison (if) and 1 Memory Access (A[i] in the if condition). if block is executed for all iterations of while in the worst case. if block has 2 Memory Accesses and an Assignment and then a Jump happens to outside the if-else statement. Hence $(C + 3M + As + J) * j$

The total cost would then be (terms are for outer loop, inner loop, if and else):

$$\sum_{j=1}^{n-1} ((C + 2Ar + 4As + 2M + J) + (C + As + Ar + J) * j + (C + 3M + As + J) * j)$$

$$= (C + 2Ar + 4As + 2M + J) * (n-1) + (C + As + Ar + J) * (n * (n-1)/2)$$
$$+ (C + 3M + As + J) * (n * (n-1)/2)$$

This means Insertion Sort is $\mathcal{O}(n^2)$.

(C is Comparison, As is Assignment, Ar is Arithmetic, M is Memory Access, J is Jump)

Note: Some intermediate operations might have been omitted but they do not affect the overall asymptotic performance of the algorithm. The time taken for comparisons, jumps, arithmetic operations, and memory accesses are assumed to be constants for a given machine and architecture.

2. *What is the time complexity of binary addition and multiplication? How much time does it take to do unary addition?*

Solution: Note that time complexity is measured as a function of the input size since we aim to determine how the time the algorithm takes scales with the input size.

Binary Addition
Assume two numbers A and B. In binary notation, their lengths (number of bits) are m and n. Then the time complexity of binary addition would be $\mathcal{O}(m+n)$. This is because we can start from the right end and add (keeping carry in mind) from right to left. Each bit requires an O(1) computation since there are only 8 combinations (2 each for bit 1, bit 2, and carry). Since the length of a number N in bits is $\log N$, the time complexity is $\mathcal{O}(\log A + \log B) = \mathcal{O}(\log(AB)) = \mathcal{O}(m+n)$.

Binary Multiplication
Similar to above, but here the difference would be that each bit of the larger number (assumed to be A without loss of generality) would need to be multiplied by the smaller number, and the result would need to be added. Each bit of the larger number would take $\mathcal{O}(n)$ computations, and then m such numbers would be added. So the time complexity would be $\mathcal{O}(m \times \mathcal{O}(n)) = \mathcal{O}(mn)$[1]. Following the definition above, the time complexity is $\mathcal{O}(\log A \times \log B) = \mathcal{O}(mn)$.

Side note: How do we know if this is the most efficient algorithm? Turns out, this is NOT the most efficient algorithm. The most efficient algorithm (in terms of asymptotic time complexity) has a time complexity of $\mathcal{O}(n \log n)$ where n is the maximum number of bits in the 2 numbers.

Unary Addition
The unary addition of A and B is just their concatenation. This means the result would have $A+B$ number of 1's. Iterating over the numbers linearly would give a time complexity of $\mathcal{O}(A+B)$ which is linear in input size.

3. *Given $f(n) = a_0 n^0 + \ldots + a_d n^d$ and $g(n) = b_0 n^0 + \ldots + b_e n^e$ with $d > e$, then show that $f(n) \notin \mathcal{O}(g(n))$*

Solution: Let us begin by assuming the proposition is False, ergo, $f(n) \in \mathcal{O}(g(n))$. By definition, then, there exists a constant c such that there exists another constant n_0 such that

$$\forall n \geq n_0, f(n) \leq cg(n)$$

. Hence, we have

$$\forall n \geq n_0, a_0 n^0 + \ldots + a_d n^d \leq c b_0 n^0 + \ldots + b_e n^e$$

$$\forall n \geq n_0, \sum_{i=0}^{e} (a_i - c b_i) n^i + a_{i+1} n^{i+1} + \ldots + a_d n^d \leq 0$$

By definition of limit

$$\lim_{n\to\infty} \sum_{i=0}^{e} (a_i - cb_i)n^i + a_{i+1}n^{i+1} + \ldots + a_d n^d \leq 0$$

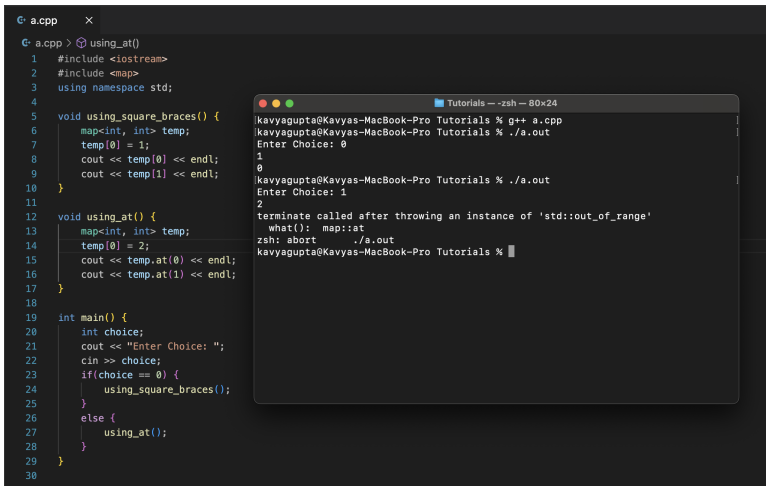
$$\implies a_d \leq 0$$

Assuming $a_d > 0$ (since we are dealing with functions mapping from $\mathbb{N}$ to $\mathbb{N}$), this results in a contradiction; thus, our original proposition is proved.

4. *What is the difference between "at" and "..[..]" accesses in C++ maps?*

Solution: Both accesses will first search for the given key in the map but will behave differently when the key is not present. The at method will throw an exception if the key is not found in the map, but the $[\,]$ operator will insert a new element with the key and a default-initialized value for the mapped type and will return that default value.

Look at the following illustration:-



5. *C++ does not provide active memory management. However, smart pointers in C++ allow us the capability of a garbage collector. The smart pointer classes in C++ are*

   - $\mathrm{auto\_ptr}$
   - $\mathrm{unique\_ptr}$
   - $\mathrm{shared\_ptr}$
   - $\mathrm{weak\_ptr}$

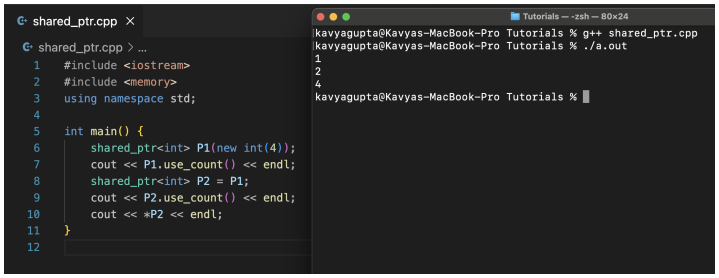
   Write programs that illustrate the differences among the above smart pointers.

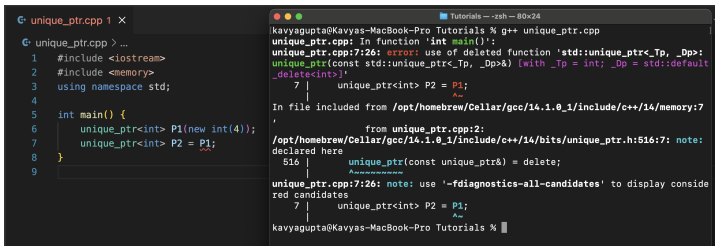
Solution: Memory allocated in heap (using new or malloc) if not de-allocated can lead to memory leaks. Smart pointers in C++ deal with this issue. Broadly, they are classes that store reference counts to the memory they point. When a smart pointer is created, reference count to that memory is increased by one. Whenever a smart pointer (pointing to the same memory) goes out of scope, its destructor is automatically executed, which reduces the reference count of that memory by one and when it hits zero, that memory is deallocated.

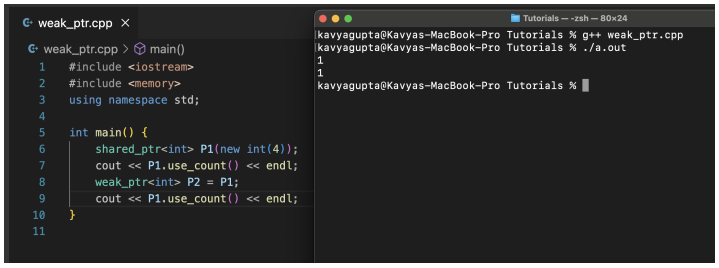
- shared_ptr allows multiple references to a memory location (or an object)
- weak_ptr allows one to refer to an object without having the reference counted
- unique_ptr is like shared_ptr but it does not allow a programmer to have two references to a memory location
- auto_ptr is just the deprecated version of unique_ptr

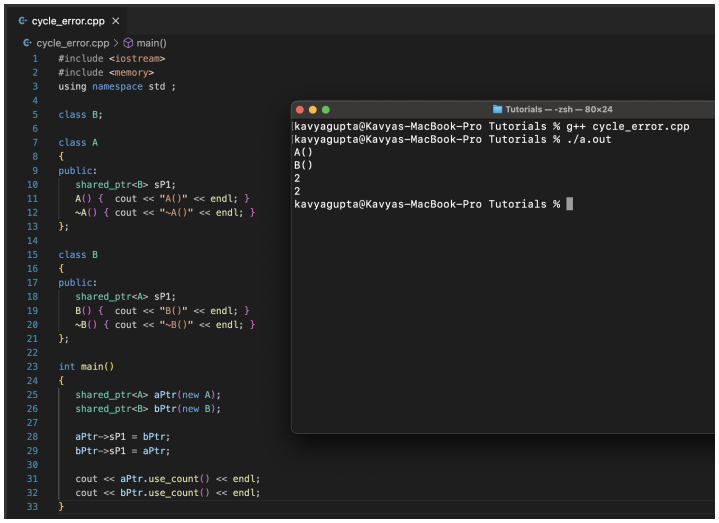
The use for weak_ptr is to avoid the circular dependency created when two or more object pointing to each other using shared_ptr.

You can see the reference count of a shared_ptr using its use_count() function.









6. *Why do the following three writes cause compilation errors in the C++20 compiler?*

```
class Node {
    public :
    Node(): value(0) {}
    const Node& foo(const Node* x) const {
        value = 3; // Not allowed because of ______________
        x[0].value = 4; // Not allowed because of ______________
        return x[0];
    }
    int value;
};

int main () {
    Node x[3], y ;
    auto &z = y.foo(x);
    z.value = 5; // Not allowed because of ______________
}
```

Solution: The line 'value = 3;' is not allowed since the method 'foo' is marked **const** at the end. Using **const** after the function signature and its parameters for a class method implies that the class members cannot be changed and the object itself is constant within the bounds of the function. As a result, an error is raised.

The line 'x[0]. value = 4;' raises an error since the parameter 'x' is of type '**const** Node*', meaning that different values can be assigned to the pointer itself, but the subscripts of the pointer cannot be assigned, since it points to constant members of class 'Node'. Likewise, '*x.value = 3;' is equally illegal. Note that 'Node **const** *' also does the same thing, whereas 'Node * **const**' means that the subscripts of the pointer can be assigned since it does not point to constant members of class 'Node' but the pointer itself cannot be assigned to point to a different Node or array of Nodes. '**const** Node* **const**' or 'Node **const** * **const**' imply that the pointer cannot be assigned AND subscripts or dereferences cannot be assigned.

The last line 'z.value = 5;' is illegal since the datatype of 'z', as determined from the method 'foo' is '**const** Node &' NOT 'Node &'. The implication is that 'z' refers to a Node, which is constant and hence cannot be assigned. Note that assignments to class members are as bad as assignments to the class object itself regarding constant objects in C++.

Tutorial 2

---

1. *The span of a stock's price on i^{th} day is the maximum number of consecutive days (up to i^{th} day and including the i^{th} day) the price of the stock has been less than or equal to its price on day i.*

   Example: for the price sequence 2 4 6 5 9 10 of a stack, the span of prices is 1 2 3 1 5 6.

   Give a linear−time algorithm that computes s_i for a given price series.

   Solution: The idea here is to find out the latest (most recent) day till the i^{th} day where the price was greater than the price on day i. To do this, we will maintain a stack of indices, and for each index i, we will keep the stack in a state such that the following invariant holds for each day i:

   If the stack is not empty, then the price on the day whose index is at the top of the stack is the most recent price that was strictly greater than the current price.

   With this invariant, the updates to the stack and the stock span follow as:

   - we remove the top index in the stack till the price corresponding to the top index becomes strictly greater, or the stack becomes empty
   - if the stack is empty, the current price is the largest, so the span is the number of days so far
   - otherwise the span is the number of days since the index at the top of the stack

   (Such a stack is known as a "Monotonic Stack").

   The base case is that at day 1 the price is the only encountered price, and hence is the largest, so the span is 1.

   The time complexity is indeed $\mathcal{O}(n)$ (n being the number of days/size of array s) because each element is pushed **exactly once** in the stack and popped **atmost once**.

   Look at Algorithm 2 for the pseudo-code.

Algorithm 2: Stock Span Algorithm

```
Data: Array price of length n and an empty stack St
for i ← 0 to n − 1 do
    s[i] ← i + 1 //If price[i] is ≥ all the previous prices, then s[i] will be i + 1
    while not St.isEmpty() do
        if price[St.top()] > price[i] then
            s[i] ← i − St.top()
            break
        end
        St.pop()
    end
    St.push(i)
end
```

2. *There is a stack of dosas on a tava, of distinct radii. We want to serve the dosas of increasing radii. Only two operations are allowed:*

   (i) serve the top dosa
   (ii) insert a spatula (flat spoon) in the middle, say after the first k, hold up this partial stack and flip it upside-down and put it back

Design a data structure to represent the tava, input a given tava, and to produce an output in sorted order. What is the time complexity of your algorithm?

This is also related to the train-shunting problem.

Solution: Read the first line of the question with the emphasis being on **stack**. We will use the array representation of a stack for the tava.

Our stack-tava has the following abstraction:

- Initialization and input: Initialize an array S of size n, to represent the stack. Using the given input, fill the array elements with the radii of dosas on the tava in the initial stack order, with the bottom dosa at index 0 and the top at $n-1$. Maintain a variable sz which contains the current size of the stack, and initialize it to n.
- Serving the top dosa: This method is essentially carrying out the "pop" operation on our stack S. Return the top dosa in the stack, $S[sz-1]$, and decrement sz by 1. Clearly, this method takes a constant $\mathcal{O}(1)$ time for every call.
- Flipping the top k dosas: This method is equivalent to reversing the slice of the array S from index $sz-k$ to $sz-1$. This is simple enough: initialize an index i to $sz-k$ and another index j to $sz-1$, swap elements at indices i and j, increment i and decrement j by 1, and repeat the swap and increment/decrement while $j > i$. This method takes $\mathcal{O}(k)$ time for every call.

Note that in this implementation, the serve operation is the only way by which the problem can be reduced in size (number of dosas reduced by 1). The flip operation, if used wisely, can give us access to dosas that can help us reduce the problem size.

With this implementation, the algorithm to serve the dosas in increasing order of the radii can be designed as follows. While the stack is not empty, repeat the following:

- Iterate over the array S and find the index of the minimum element in the array. Let this index be m. This is the position of the dosa having the smallest radius
- Flip over the top $sz-m$ dosas in the stack, using the flipping operation. This brings the smallest dosa from index m to index $sz-1$, id est, the top of the stack
- Serve the top dosa, using the serving operation. This pops the smallest dosa off the stack

It is easy to argue the correctness. The invariant is that we keep serving the minimum radii dosa from the remaining stack. The resulting order has to be sorted.

For computing the time complexity: in a stack of size z, finding the index of the minimum element takes $\mathcal{O}(z)$ time, flipping over the top $k \leq z$ elements takes $\mathcal{O}(k)$ time (and thus $\mathcal{O}(z)$ time) and serving off the top dosa takes $\mathcal{O}(1)$ time. Thus serving the smallest dosa takes $\mathcal{O}(z)$ time overall. This has to be repeated for all dosas, so the stack size z goes from n to 1, decrementing by 1 every time. Therefore the algorithm has a time complexity of $\mathcal{O}(n^2)$.

3. *(a) Do the analysis of performance of exponential growth if the growth factor is three instead of two? Does it give us better or worse performance than doubling policy?*

   (b) Can we do the similar analysis for growth factor 1.5?

Solution: Let us do the analysis for a general growth factor α. Suppose initially N $= 1$ and there are n $= \alpha^i$ consecutive pushes. So, total cost of expansion is (refer to slides for detailed explanation):

$$(\alpha + 1) \cdot (\alpha^0 + \alpha^1 + \alpha^2 ... + \alpha^{i-1})$$
$$\frac{\alpha + 1}{\alpha - 1} \cdot (\alpha^i - 1)$$
$$\frac{\alpha + 1}{\alpha - 1} \cdot (n - 1)$$

For α equals to 3 cost of expansion is $2 \cdot (n - 1)$, it's better than doubling policy. For α equals 1.5 cost of expansion is $5 \cdot (n - 1)$, which is worse than doubling policy. Trade-off involved is extra memory allocation, maximum extra memory allocated is $\alpha - 1$ times the requirement, so with increasing alpha extra memory allocation is increasing.

4. *Give an algorithm to reverse a linked list. You must use only three extra pointers.*

Solution: Let us initialise three pointers - *prev, curr*, and *next*, initialised to null, head and $head -> next$ respectively. If head is null, then it is an empty linked list and we return. Otherwise, we will be iterating over each element in the linked list. In each iteration, perform the following updates.

- $curr -> next =$ prev
- prev $=$ curr
- curr $=$ next
- next $= next -> next$

The loop terminates when $next$ is null, in which case, set head to curr.

5. *Give an algorithm to find the middle element of a singly linked list.*

Solution: The idea is as follows:

- Make 2 pointers middle and end initialized to head of linked list
- At each step, increment middle by one and end by 2
- Continue this process until end reaches the end of linked list
- return the middle element
- return null if the head itself was null (empty linked list)

Note: When there is an odd number of elements, the above algorithm returns the element at the median location. What happens when there is an even number of elements?

6. *Given two stacks S1 and S2 (working in the LIFO method) as black boxes, with the regular methods: "Push", "Pop", and "isEmpty", you need to implement a Queue (specifically: Enqueue and Dequeue working in the FIFO method). Assume there are n Enqueue/Dequeue operations on your queue. The time complexity of a single method Enqueue or Dequeue may be linear in n, however the total time complexity of the n operations should also be $\Theta(n)$*

Solution:

Let us try to formulate a simple implementation of a queue using 2 stacks.

Label the stacks as A and B. We will use stack A to store elements of the queue sequentially, and stack B to implement the dequeue operation.

Enqueue: To insert a new element to the queue, we can just *push* the element to the top of stack A. Here, each *enqueue* operation requires $\mathcal{O}(1)$ time.

Dequeue: To remove an element from the queue in a FIFO manner, we must somehow delete the first element pushed to stack A. Since we cannot access any element of the stack other than the top one (last inserted), we will have to use stack B to implement this.

To get access to the first inserted element, we need to *pop* all the remaining elements from stack A and store them into stack B. As you can see in the figure 1, the order of elements is reversed while moving them to stack B. When we reach the first element of A, we can just *pop* it without then adding it to B.

We must now decide what to do with the elements of stack B. One idea is to simply use stack B for storing the queue in the place of stack A, but this doesn't work since the order of elements would change.

Another possible approach would be to again transfer all the elements saved in stack B to stack A and restore the queue. This would preserve the order of elements and allow new elements to be pushed to A while maintaining correctness. However, think of the worst case time complexity of this approach. For every *dequeue* operation, all the elements have to be shifted twice - once from A to B and then from B to A. This causes each *dequeue* operation to be of $\mathcal{O}(n)$ time complexity and the total time complexity of n operations to be $\mathcal{O}(n^2)$. (think when?)

The blowup in time complexity is caused due to elements being shifted back to A from B on every *dequeue*. Do we really need that? What happens if we just let the elements stay in B?

What happens now if we have 2 successive dequeue operations? Suppose we transferred the elements of queue from stack A to B in the first operation. For the 2nd dequeue operation, we can simply *pop* from stack B. Hence we don't need to re-transfer the elements. Now, if we have n operations, our complexity reduces to $\mathcal{O}(n)$.

To sum it up: Look at Algorithm 3 for the pseudo-code

Algorithm 3: Queue using Two Stacks

```
1  Function Enqueue(element):
2  |  S1.Push(element)
3  end
4  Function Dequeue():
5  |  if S1.isEmpty() and S2.isEmpty() then
6  |  |  throw("Queue is empty.")
7  |  end
8  |  if S2.isEmpty() then
9  |  |  while not S1.isEmpty() do
10 |  |  |  S2.Push(S1.Pop())
11 |  |  end
12 |  end
13 |  return S2.Pop()
14 end
```

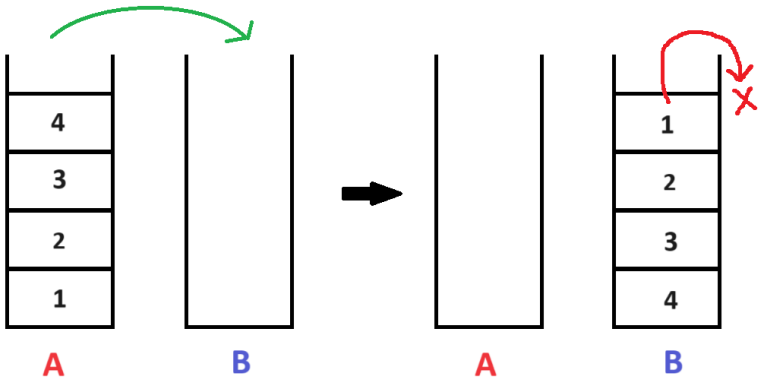


Figure 1: 2 Stacks making a Queue

Tutorial 3

---

1. *What is the probability for the 3rd insertion to have exactly two collisions while using linear probing in the hash table?*

> **Solution:** Let the size of the hash table be $n \geq 3$. We assume that insertions take place via linear probing. Further, we assume that no deletions or insertions occur in the same sequence. Lastly, we assume that the hash function is uniform, i.e., for a random key, the likelihood of any hash value occurring is exactly $1/n$. After collisions, let the first two insertions occur at indices i and j.
>
> If $((i+1)\ mod\ n) \neq j$ and $((j+1)\ mod\ n) \neq i$, then the third insertion cannot have two collisions. The reason is that if the hash value of the third key collides with the first key, then index $(i+1)\ mod\ n$ is vacant, and there is only one collision. If the hash value is at index j, then index $(j+1)\ mod\ n$ is vacant and hence, only one collision. Any other hash value implies no collision. Let us neglect this case.
>
> We focus only on cases where i and j are adjacent to each other on the table. Let us re-label the indices as a and b such that $((a+1)\ mod\ n) = b$. In this case, for any constant $0 \leq c < n$, the absolute value of $P(a = c)$ (not conditional) can be thought of as the probability that $((i = c$ and $j = (i+1)\ mod\ n)$ or $(j = c$ and $i = (j+1)\ mod\ n))$. The first event occurs when the second hash value is either i or $(i+1)\ mod\ n$, with a chance of $(1/n) \cdot (2/n)$. The second event occurs when the second hash value is $(n+i-1)\ mod\ n$, with a chance of $(1/n) \cdot (1/n)$.
>
> From the above, we can say that $P(a = c) = (3/n^2)$ for all constants c, we safely say that two collisions occur if and only if the third hash-value is a, which occurs with a probability of $\frac{1}{n}$. However, c and, therefore, a can take any value. Thus, the product $(3/n^2) \cdot (1/n)$ has to be summed up across all values (n times), giving us the required answer: $(3/n^2)$.

2. *Given that k elements have to be stored using a hash function with target space n. What is the probability of the hash function having an inherent collision? What is an estimate of the probability of a collision in the insertion of n elements?*

> **Solution:** To compute the probability, we first count the negative case: no collisions at all. This means that the k keys get mapped to distinct k keys out of n. We assume that for a random key and for any constant c, $P(hash(k) = c) = 1/n$.
>
> The first key has n choices of hash values. Given each choice of the first key, the second key has $n-1$ possible hash values such that there is no collision. Given the first two keys and hash values, the third has $n-2$ choices, and so on, till the k^{th} key has $n-k+1$ choices. Overall, the ordered k-tuple has $\prod_{i=0}^{k-1}(n-i)$ choices for no collision. All those are mutually exclusive events with likelihoods $(1/n)^k$ each, being the product of k probabilities of $1/n$ for k independent events and their respective outcomes. Thus,
>
> $$P(No\ collisions) = \frac{n!}{(n-k)! \cdot n^k}$$
>
> One minus this expression gives the likelihood of a collision.
>
> If $k = n$, the expression boils down to $(n!/n^n)$. We apply Stirling's approximation here:
>
> $$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} < n! < \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

$$\sqrt{2\pi n}\left(\frac{1}{e}\right)^n e^{\frac{1}{12n+1}} < \frac{n!}{n^n} < \sqrt{2\pi n}\left(\frac{1}{e}\right)^n e^{\frac{1}{12n}}$$

For large n, we can neglect the terms $e^{(1/(12n+1))}$ and $e^{(1/12n)}$ as both are very close to 1. However, the term $\sqrt{2\pi n} \cdot e^{-n}$ tends to zero as n grows very large. This means that the required probability is sandwiched between terms that exponentially decay toward zero.

The implication of the above is the likelihood that all n keys get uniquely mapped to the n possible hash values decay exponentially with n, and thus, collisions are almost certain for large n. This can be linked with the birthday paradox in that in a class of 365; it is almost impossible for all students to have different birthdays; even a single tutorial batch has a high likelihood of at least two birthdays being the same. In the context of hashing, although the expected number of terms sharing a hash value is one per hash value, we expect a good number of collisions to occur during hashing.

3. *Let $C(i)$ be the chain of array indices that are queried to look for a key k in linear probing where $h(k) = i$.*

   (a) How does this chain extend by an insertion, and how does it change by a deletion?

   (b) A search for a key k ends when an empty cell is encountered. What if we mark the end of C (i) with an end marker? We stop the search when this marker is encountered. Would this work? Would this be efficient?

   (c) Is there a way of not using tombstones?

Solution: To recall, linear probing covers indices i, $i+1$, ... until an empty index is found (then k not found in the table) or the cell stores k. Note that tombstones do not count as empty cells in the hash table; counting them so after deletion would break existing chains and result in false negatives when searching. Including them means no change in these chains, answering part of the first (a) subproblem.

If the chain terminates at k (key found), then inserting a new element does not change anything. If the added element is k itself, then it occupies the position at the end of the chain, a vacant cell. So, no change. In the other case, the chain extends by one if the inserted element is at the end of the chain and is not k. The chain could extend a lot more if the inserted element joins the existing chain with another series of filled cells in the table. Note that the chain, except for the end, has only filled or tombstone cells; thus, the insertion cannot occur in the middle of the chain. If the insertion of the element that is not k does not occur at the end of the previous chain for searching k, then there is no change.

The answer to the second (b) part is that it would work but not be any more efficient than the standard algorithm. For this to work, an empty cell once encountered during the search should be marked an 'end.' Otherwise, the first search would go on an infinite loop, looking for an end marker. If this is the case, the search must look for both end marks and empty cells in each iteration, making it slightly less efficient but at the same time complexity.

For the third (c) part, a slight modification should be made to the algorithm discussed above: all cells should be marked 'end' at the start. Otherwise, a deletion without a tombstone would cause false negatives in future searches, just like the discussions without tombstones. If this modification is made, then we basically replace "empty" with an "end" marker and "tombstone" marker with "empty". The algorithm is, otherwise, identical to the normal algorithm with tombstones, and hence, we really do not avoid tombstones, although we do not use them. Note that marking the freed cells as 'end' on deletions would be as bad as not using tombstones, which results in false negatives.

4. *Let $m = 11$, $h_1(k) = (k \ mod \ 11)$, $h_2(k) = 6 - (k \ mod \ 6)$.*
   Let us use the following hash function for an open addressing scheme.

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \ \ mod \ m$$

   1. *What will be the state of the table after insertions of 41, 22, 44, 59, 32, 31, and 74?*
   2. *Let $h_2(k) = p - (k \ mod \ p)$. What should be the relationship between p and m such that h is a valid function for linear probing?*
   3. *What is the average number of probes for an unsuccessful search if the table has α load factor?*
   4. *What is the average time for a successful search?*

Solution: To recall, the standard linear probing uses the function $h(k,i) = (h(k) + i) \ \ mod \ m$ where m is the size of the hash table [Here 11]. The way it works is that we check the cells indexed at $h(k,0), h(k,1), h(k,2), \ldots$ till we find an empty cell $h(k,a)$ for some a. The same logic should be extended here, except $h(k,i)$, which is defined in the problem statement.

Let the table before adding 41 be

Now, $h_1(41) = 8$ and $h_2(41) = 1$. This means that $h(41,0) = 8$ and cell 8 is empty. So, we add 41 there:

								41		

$h_1(22) = 0$ and cell 0 is empty so we can add 22 there. $[h(k,0) = h_1(k)]$

22								41		

$h_1(44) = 0$ but cell 0 is filled. Now, $h_2(44) = 4$ and $h(44,1) = 4$ which is an empty tile.

22				44				41		

$h_1(59) = 4$ which is a filled cell. $h_2(59) = 1$ and $h(59,1) = 5$ is an empty tile.

22				44	59			41		

$h_1(32) = 10$ and $h_1(31) = 9$ are both empty and different cells, so we can add them.

22				44	59			41	31	32

Lastly, $h_1(74) = 8$, which is occupied. $h_2(74) = 4$. Now, $h(74,1) = 1$ which is free.

22	74			44	59			41	31	32

This is how the table looks after inserting the elements in the given order.

Moving on to the next part, a valid function for linear probing should eventually cover all cells in the grid as i increments. This is not possible if $gcd(h_2(k), m) > 1$, for one or more values of k. The reason is that $h(k,i) \ mod \ gcd(h_2(k), m)$ will be fixed, and hence, $h(k,i) \ mod \ m$ will only take some values. But if that gcd is 1, then it means that for all $0 \leq i < m$, $gcd \cdot i \ mod \ m$ will have a unique value, and the number of these unique values will be m, so all cells of the table will be covered.

Note that $gcd(h_2(k), m) = 1$ should hold for all values of k, not just for any one k. Keeping $h_2(k) = p - (k \ mod \ p)$ can take any value in $1, 2, \ldots p$ depending on the value of k. Thus, none of the first p natural numbers should share a common factor with m. In other words, the smallest prime factor (not 1) of m, the size of the hash table, should be larger than p. It is not necessary that m is prime – for $p = 6$, $m = 7 \cdot 13 = 91$ also works.

Exact expression of expected number of probes in a search (unsuccessful/successful) is mathematically complex hence we'll prove an appropriate upper bound for them. Claim: The mentioned expression for an unsuccessful one is atmost $1/(1-\alpha)$. This is assuming $\alpha < 1$.

Proof: Let X be the number of probes made in an unsuccessful search, and let A_i be the event that an i^{th} probe occurs and it is to an occupied slot. Then $X \geq i$ when the event $A_1 \cap A_2 \cap \ldots \cap A_{i-1}$ occurs. Hence,

$$\begin{aligned} P(X \geq i) &= P(A_1 \cap A_2 \cap \ldots \cap A_{i-1}) \\ &= P(A_1) \times P(A_2|A_1) \times P(A_3|A_2 \cap A_1) \times \ldots \times P(A_{i-1}|A_1 \cap \ldots \cap A_{i-2}) \end{aligned}$$

$P(A_1) = n/m$ because there are n elements and m slots. For $j > 1$, the probability that there is a j^{th} probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is $(n-j+1)/(m-j+1)$ because we are finding one of the remaining $(n-(j-1))$ elements in one of the $(m-(j-1))$ unexamined slots, and we have uniform hashing so each of the slots is equally likely to be chosen next.

Because we have $n < m$, we have $(n-j)/(m-j) \leq n/m \ \forall \ 0 \leq j < m$. Thus,

$$P(X \geq i) = \frac{n}{m} \times \frac{n-1}{m-1} \times \ldots \times \frac{n-i+2}{m-i+2} \leq (\frac{n}{m})^{i-1} = \alpha^{i-1}$$

So $E[X] = \sum_{i=1}^{m-1} P(i^{th} \text{ probe occurs}) = \sum_{i=1}^{m-1} P(X \geq i) \leq \sum_{i=1}^{m-1} \alpha^{i-1} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = 1/(1-\alpha)$. Hence proved.

Claim: Expected number of probes for a successful search is atmost $\frac{1}{\alpha} ln(\frac{1}{1-\alpha})$.

Proof: A search for a key k reproduces the same probe sequence as when the element with key k was inserted. If k was the $(i+1)^{th}$ key inserted into the hash table, there were only i keys in the table at the time, so the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$ (as an insertion is equivalent to an unsuccessful search, here $\alpha = i/m$). Averaging over all n keys in the hash table, we get,

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} \frac{1}{k} \\ &\leq \frac{1}{\alpha} \int_{m-n}^{n} \frac{1}{x} \, dx \\ &= \frac{1}{\alpha} ln(\frac{m}{m-n}) \\ &= \frac{1}{\alpha} ln(\frac{1}{1-\alpha}) \end{aligned}$$

Hence Proved.

5. *Suppose you want to store a large set of key-value pairs, for example, $(name, address)$. You have operations in this set: addition, deletion, and search of elements. You also have queries about whether a particular name or address is in the set, and if so, then count them and delete all such entries. How would you design your hash tables?*

Solution: Let us recall that in the classic hashing, the hash function operates on the key, and the value is simply stored alongside the key. An intuitive idea is to keep the key (from the point of view of hashing) as **both** the name and the address, with no special value otherwise. The hash function operates on this and gives us the index. Unfortunately, this gives rise to a problem: there are multiple different hash values for a given name, and hence, searching for the existence of an entry given a name (but not address – we do not know which addresses are there given this name) is as bad as linearly iterating through the table.

One way to get around this is to use a two-level hash table: the outer table N stores the keys as the names alone, and each value corresponding to a name points to a hash table itself. This inner hash table $N[name]$ stores only addresses and only addresses where the name is the corresponding key of the outer table. This makes finding elements based on names easy but does not solve the problem of searching for given addresses.

To solve that, we can use a double two-level hash table: one two-level table N as described above and one two-level table A done the other way round, i.e., A stores the keys as addresses and values as pointers to inner hash tables $A[address]$ that store names corresponding to addresses. Note that this requires additional storage space (double) and more maintenance.

Specifically, it is important to ensure integrity: for every $(name, address)$ pair in the set, $N[name]$ should exist and contain $address$, and $A[address]$ should exist and contain $name$. Insertion should be done in both tables at once. On the other hand, if any $(name, address)$ pair does not exist, then $A[address]$, if it exists, should not contain $name$, and $N[name]$, if it exists, should not contain $address$. This should be reflected in the deletion of pairs.

Tutorial 4

---

1. *Given a tree with a maximum number of children as k. We give a label between 0 and $k-1$ to each node with the following simple rules.*

   (i) The root is labelled 0.
   (ii) For any vertex v, suppose that it has r children, then arbitrarily label the children as $0, \ldots, r-1$.

   This completes the labelling. For such a labelled tree T, and a vertex v, let $Seq(v)$ be the labels of the vertices of the path from the root to v. Let $Seq(T) = \{Seq(v) \mid v \in T\}$ be the set of label sequences.

   What properties does $Seq(T)$ have? If a word w appears, what words are guaranteed to appear in $Seq(T)$? How many times does a word w appear as a prefix of some words in $Seq(T)$?

   Solution: Some properties are $Seq(T)$ are:-

   1. Size of this set will be n (n is the number of nodes in the tree) (See that for different v_1, v_2 vertices, $Seq(v_1) \neq Seq(v_2)$).
   2. If there are h_j nodes of j length in set $Seq(T)$, then the tree has h_j nodes at j^{th} level.
   3. If we consider ordering of set $Seq(T)$ in lexicographical order as the metric, then it will just represent a DFS on the tree.
   4. If we consider ordering of set $Seq(T)$ by length followed by lexicographical order as the metric, then it will just represent a BFS on the tree.

   $p_0 = 0$ (root of tree)

   $$w = p_0 p_1 p_2 .. p_n$$
   $$S_w = \{x \mid x = p_0 p_1 p_2 .. p_i q \text{ where } 0 \leq i \leq n-1 \ \& \ 0 \leq q \leq p_{i+1}\} \bigcup \{0\}$$

   We can prove S_w is the set which is guaranteed to be present if w is present by following argument:- Each element of set S_w is guaranteed to be present if w is present. We can prove this by considering two claims,

   1. Any prefix of w will be present since this just represents the path from root to ancestor of w.
   2. If a word $l_0 l_1 l_2 .. l_n$ is present then all the words of form $l_0 l_1 l_2 .. l_{n-1} q$ where $0 \leq q \leq l_n$ will also be present since these are just the child of the node represented by sequence $l_0 l_1 l_2 .. l_{n-1}$

   If $w = Seq(v)$ for some $v \in T$ and the number of nodes in the subtree of v be x, then w appears as a prefix for x number of words in $Seq(T)$.

2. *Write a function that returns $lca(v, w, T)$. What is the time complexity of the program?*

   Solution: This is for a general tree and not just a binary tree.

   See the Pseudo-code in Algorithm 4.

At any node (say x), if you find the LCA in x's subtree, return it. Otherwise, see if v, w are present in x's subtree (subtree includes x). If yes, then x is the LCA, otherwise return $NULL$.

Time complexity is $\mathcal{O}(n)$, n being the number of nodes of T, as we go through each node atmost once.

If there existed a "parent" pointer in each node, then can you form a new LCA algorithm ?

Algorithm 4: LCA

```
Function LCA(curr, v, w):
    if curr = NULL then
        return False, False, NULL //This return type is an ordered tuple
    end
    v_present ← (curr = v) // True or False
    w_present ← (curr = w)
    for child ∈ children[curr] do
        v_in_child, w_in_child, LCA_child ← LCA(child, v, w)
        if LCA_child ≠ NULL then
            return True, True, LCA_child
        end
        v_present ← v_present || v_in_child
        w_present ← w_present || w_in_child
    end
    if v_present = True and w_present = True then
        return True, True, curr
    end
    return v_present, w_present, NULL
end
// Call LCA(root, v, w) in the main function
```

3. *Given $n \in T$, Let $f(n)$ be a vector, where $f(n)[i]$ is the number of nodes at depth i from n.*
   - *Give a recursive equation for $f(n)$.*
   - *Give a pseudo-code to compute the vector $f(root(T))$. What is the time complexity of the program?*

Solution:

Observations:

- For any node n, the only node at depth 0 from n is the node n itself.
- For a leaf node l, there is no node n such that *depth(n)* from l is greater than or equal to 1.
- For a node n, if node m is at depth k, then there exists a *unique path* from n to m of length k, and this path must go through one of the children of n (let's say p) i.e. node m is at a depth $k-1$ from node p.
  Conversely, if a node m is at a depth $k-1$ from node p, then it must be at a depth of k from node n where n is parent of p.

With these observations in mind, we can say that $f(n)[i] = \sum_{m \in children(n)} f(m)[i-1]$.
Recursive equation of $f(n)$ then can be written as:

$$f(n) = [1] + (\sum_{m \in children(n)} f(m))$$

which is nothing but concatenating [1] in front of the sum obtained from the f vectors of the children.
Here base case will be when n is a leaf, for that we have

$$f(n) = [1, 0, 0, \cdots, 0]$$

Pseudocode: (Look at Algorithm 5)
As you all know, we will be doing some kind of traversal to get $f(root(T))$!!!
Can you think of the traversal method we gonna use by looking at the recursion obtained in **part 1** ???
I hope you got it correct, we are going to use post-order traversal (why so?)
We will use array data structure for storing $f(i)$ of size n, where n is the number of nodes in the tree
The algorithm given above, does a post order traversal on tree (note that this algorithm holds good for any tree and not just a binary tree), on getting to a leaf, it returns an array $[1, 0, \cdots, 0]$, and on a internal node, it returns f according to the recursion obtained above.
Time complexity
Assume that vectors at all the nodes are of size n. There will be $\mathcal{O}(n)$ edges and for each edge we will do addition of two such vectors. Hence this algorithm has $\mathcal{O}(n^2)$ time complexity.

Algorithm 5: Computing $f(n)$

```
Data: Tree T having n nodes
Function ModifiedPostOrderWalk(p):
    f ← [1, 0, ..., 0]
    if p.isLeaf() then
        return f
    end
    for n' ∈ children[p] do
        f_c ← ModifiedPostOrderWalk(n')
        for j ← 1 to n − 1 do
            f[j] ← f[j] + f_c[j − 1]
        end
    end
    return f
end
```

4. *Give an algorithm for reconstructing a binary tree if we have the preorder and inorder walks.*

Solution: The key idea here is to note that we will recursively use both traversals in parallel. We will use the inorder traversal for finding the node index and iterate over the preorder to build the tree.

We assume that we have a function that can search for a node in the inorder traversal given the start and end indices of the search range.

To utilise both the traversals, we maintain a start and end index for the inorder traversal of the tree and make recursive calls while moving these indices and the iterator of the preorder traversal.

The tree build function can be written recursively as follows:

- if the start index is greater than the end index the tree is $NULL$
- otherwise we first create the root as the first element of the preorder
- if the start index and the end index are equal then the root is the answer
- otherwise we search for the index of the root in the inorder traversal
- the left sub-tree can be built recursively by setting the end index as the index previous to the root while the right sub-tree can be built after this by setting the start index as the index next to the root (note that the preorder traversal will be linearly iterated over by both these recursive calls but the iterator will be common across the calls)
- we return the root after setting the results of the sub-tree computations

Explanation: Note that if we know the position of the root in the inorder traversal, the left side of the inorder traversal is the inorder traversal of the left sub-tree and similarly for the right sub-tree. We also note that the first element in the preorder traversal is always the root, and the following elements will be the preorder traversal of the left sub-tree followed by the right sub-tree. The time complexity of this will be $\mathcal{O}(n^2)$ (think what is the worst case).

Correctness: Arguing for correctness is straightforward. Within each recursive call, the root is the element pointed to by the preorder iterator, which increments by one for each recursive call. After that, we find the root in the inorder traversal and then build the tree using the left side of the inorder traversal and the next set of the elements in the preorder traversal (both of these correspond to the left sub-tree). Then we build the tree using the right side of the inorder traversal and the remaining set of the elements in the preorder traversal (both correspond to the right sub-tree). Since the invariants (left-root-right for inorder and root-left-right for preorder) are maintained, the recursion terminates correctly.

Additional: What is the **minimal** set of changes required for building a tree using the postorder and inorder traversals?

5. *Let us suppose all internal nodes of a binary tree have two children. Give an algorithm for reconstructing the binary tree if we have the preorder and postorder walks.*

Solution:

Let preorder walk be $f = (f_1, f_2, \cdots, f_n)$ and postorder walk be $l = (l_1, l_2, \cdots, l_n)$
Base case
f and l have just 1 element let's say e, then the tree is nothing but a single-node tree with root as e.
Recursion step
Observations for tree with each internal node having two children and number of nodes greater than 1

1. In preorder walk, *first* element is *root* and *second* element is left child of root.
2. In postorder walk, *last* element is *root* and *second last* element is right child of root.

So, now with these observations in mind let's try to build the tree from f and l.
We have the root as $r = f_1 = l_n$

Left child of root as f_2 and right child of root as l_{n-1}
Now we will break f and l so as to get preorder and postorder walks for left and right subtree
Given f, which element in right subtree of r is first to be in the f?
By Observation 1, preorder on right subtree (which is a tree) of r should have it's root i.e. right child of r which is l_{n-1} first in the walk and so we have the break of f!!!
Find l_{n-1} in f, let us say f_k, then we have $f' = (f_2, f_3, \cdots, f_{k-1})$ as preorder walk of left subtree of r and $f'' = (f_k, f_{k+1}, \cdots, f_n)$ as preorder walk of right subtree of r
Similarly, given l, which element in left subtree of r is the last to be in the l?
By Observation 2, postorder on left subtree (which is a tree) of r should have it's root i.e. left child of r which is f_1 last in the walk and so we have the break of l!!!
Find f_2 in l, let us say l_k, then we have $l' = (l_1, l_2, \cdots, l_k)$ as postorder walk of left subtree of r and $l'' = (l_{k+1}, l_{k+2}, \cdots, l_{n-1})$ as postorder walk of right subtree of r

With f' and l', we **recursively** construct the left subtree and with f'' and l'', the right subtree. The time complexity will again be $\mathcal{O}(n^2)$.

6. (a) *Show that in order printing of BST nodes produces a sorted sequence of keys.*
   (b) *Give a sorting procedure using BST.*
   (c) *Give the complexity of the procedure.*

Solution:

(a) At any non-leaf BST node, the key at any node in the left subtree is not greater than the key at the node. Similarly, the key at any node in the right subtree is not less than the key at the node. An inorder traversal of a BST visits the left subtree first, then the node, and then the right subtree. Thus, the inorder traversal of the BST will produce a sorted sequence of keys.

(b)
- Create a BST with the first element of the unsorted array as the root and no children.
- Insert each of the remaining elements into the BST.
- Perform an inorder traversal of the BST to get the sorted sequence of keys.

(c) The complexity of the procedure is $\mathcal{O}(n \log n)$. This is because average insertion of n elements in a BST takes $\mathcal{O}(n \log n)$ time. And the inorder traversal takes $\mathcal{O}(n)$ time.

7. *Given a BST tree T and a value v, write a program to locate the leftmost and rightmost occurrence of the value v.*

Solution: Leftmost occurrence in a BST means leftmost occurrence in the inorder traversal of that BST.

Look at Algorithm 6 for the Pseudo-code of the leftmost occurrence in a BST.

The code itself is self-explanatory. Similarly you can right this for the rightmost occurrence.

Time complexity is $\mathcal{O}(h)$, h being the height of T, as we go through each level atmost once.

Algorithm 6: Finding the leftmost occurrence

```
   Data: Tree T having n nodes
1  Function Leftmost(curr, v):
2      if curr = NULL then
3          return NULL
4      end
5      if curr → val ≥ v then
6          left_check ← Leftmost(curr → left, v)
7          if left_check ≠ NULL then
8              return left_check
9          end
10     end
11     if curr → val = v then
12         return curr
13     end
14     if curr → val ≤ v then
15         right_check ← Leftmost(curr → right, v)
16         if right_check ≠ NULL then
17             return right_check
18         end
19     end
20     return NULL
21 end
22 // Call Leftmost(root, v) in the main function
```

Tutorial 5

---

1. *Given a BST T and a key k, the task is to delete all keys $< k$ from T.*

   (a) Write pseudocode to do this.

   (b) How much time does your algorithm take?

   (c) What is the structure of the tree left behind?

   (d) What is its root?

Solution:

(a) The code (See Algorithm 7) provided deletes all keys $< k$ from T and returns the root of the modified BST.

(b) The complexity of the procedure is $\mathcal{O}(h)$, h being the height of T, as we are visiting each level atmost once (worst case being reaching a leaf node).

(c) The structure of the tree left behind is still a BST.

(d) If the previous root was greater than or equal to k, then the new root is the same as the previous root. Otherwise, the new root is different.

Algorithm 7: Deleting all keys $< k$

```
Function DeleteKeys(curr, k):
    if curr = NULL then
        return NULL
    end
    if curr → val ≥ k then
        (curr → left) ← DeleteKeys((curr → left), k)
        return curr
    end
    else
        return DeleteKeys((curr → right), k)
    end
end
// Call DeleteKeys(root, k) in the main function
```

2. *Let $H(n)$ be the expected height of the tree obtained by inserting a random permutation of [n]. Write the recurrence relation for $H(n)$.*

Solution: The height of a binary tree equals one plus maximum of that of left and right subtrees. In a random permutation of $[n]$, any element (say k) can be the root of the subtree with probability $1/n$. The left subtree will contain $k-1$ nodes and the right one $n-k$.

Let X_n be the random variable representing the height of a n-node BST. Also, $H(n) = E[X_n]$. Then the recurrence relation is given as :-

$$\forall n \in \mathbb{N},\ H(n) = E[X_n] = 1 + \frac{1}{n}(\sum_{i=1}^{n} E[\max(X_{i-1}, X_{n-i})])$$
$$\leq 1 + \frac{1}{n}(\sum_{i=1}^{n} E[X_{i-1} + X_{n-i}])$$
$$= 1 + \frac{1}{n}(\sum_{i=1}^{n} E[X_{i-1}] + \sum_{i=1}^{n} E[X_{n-i}])$$
$$= 1 + \frac{2}{n}(\sum_{i=1}^{n-1} E[X_i]) = 1 + \frac{2}{n}(\sum_{i=1}^{n-1} H_i)$$

Base Case: $H(0) = 0$

Solving the recurrence gives $H(n)$ as $\mathcal{O}(\log n)$. You can refer this for a rough idea of its solution.

3. *Prove that after rotation the resulting tree is a binary search tree.*

Solution: See Figure 2. Suppose we go from left to right.

A becomes the left child of B and T_2 becomes the right child of A. $A \to val \leq B \to val$ as B was initially the right child of A (definition of BST) and $T_2 \to val \geq A \to val$ as T_2 was originally in the right subtree of A. T_1 is a BST, so is T_2, hence A's subtree is a BST too. Hence the new structure follows the rules of a BST.

Similarly we can prove for right to left in the figure.

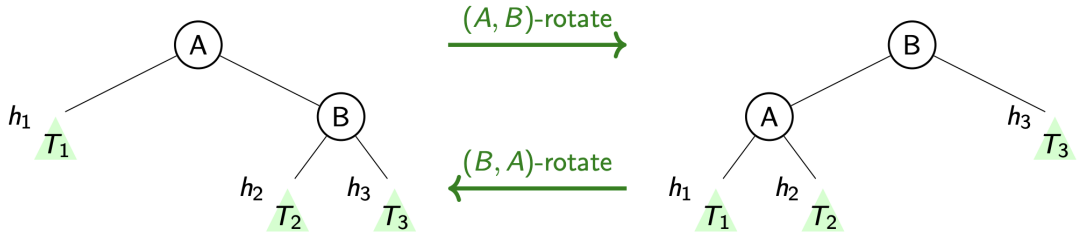


Figure 2: Rotation

4. *Insert sorted numbers 1,2,3,..., 10 to an empty red-black tree. Show all intermediate red-black trees.*

Solution: See Figure 3 below.

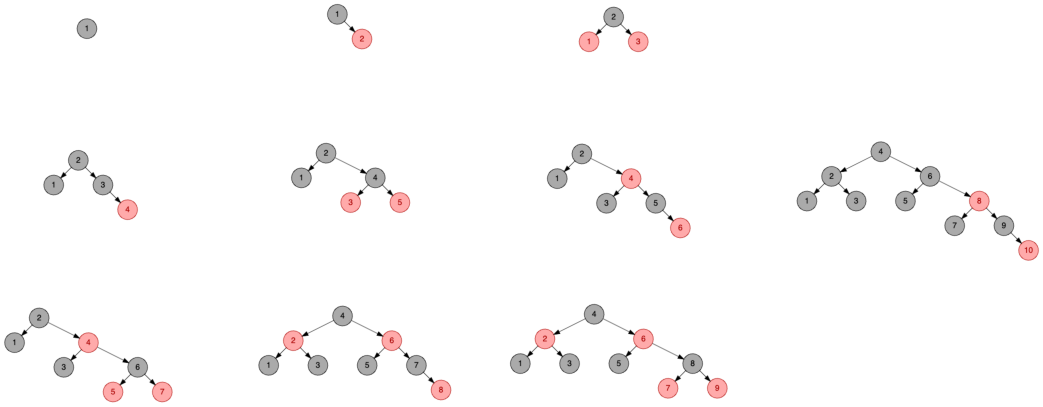
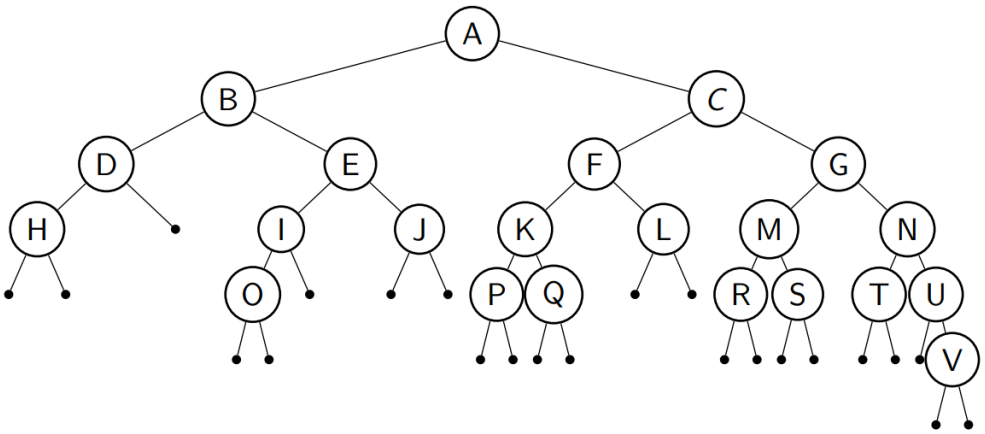


Figure 3: RBT Tree Insertions

5. *Consider the tree below. Can it be coloured and turned into a red-black tree? If we wish to store the set* $1, \ldots, 22$*, label each node with the correct number. Now add* 23 *to the set and then delete* 1*. Also, do the same in the reverse order. Are the answers the same? When will the answers be the same?*



Solution: If we color this RB tree, Black height of this tree must be 3. Why?? (Hint: Argue in terms of property of RB tree) Colour the nodes H, E, O, C, N, V, K and M red; the rest should be black. This is a red-black tree with a black height of 3. Using the inorder traversal of the tree,
$A = 8, B = 3, C = 14, D = 2, E = 6, F = 12, G = 18, H = 1, I = 5, J = 7, K = 10$
$L = 13, M = 16, N = 20, O = 4, P = 9, Q = 11, R = 15, S = 17, T = 19, U = 21, V = 22$

Insertion Followed by Deletion
The insertion is very straightforward as 23 gets added as the right child of V = 22, let's call it W. Red black tree condition gets violated as V and W are of same color now. So, according to case 3, apply a left rotation such that after rotation N.right=V, V.left=U and V.right = W. H=1 is a red color leaf, so we can simply delete it without any trouble of black height violation.

Deletion Followed by Insertion
In this case, our answer will be the same as the one above. Since, deletion of H doesn't change anything significant for the right tree and we can simply insert 23 as described above. Difference can be in cases where there is upward propagation of violation while dealing with

violation after insertion and deletion. Since, upward propagation checks and affect different parts and colors of the tree

6. *(a) Give running time complexities of delete, insert, and search in red-black tree.*

   (b) What are the advantages of red-black tree as compare to Hash table, where every thing is constant time?

Solution: (a) $\mathcal{O}(\log n)$, n being the number of nodes of RBT. (Refer slides to know why).

(b) These are the following advantages:-

- Red-black trees maintain elements in a sorted order, which is useful for operations that require order (e.g., finding the minimum/maximum, range queries). Hash tables, on the other hand, do not maintain any order among elements.
- The performance of a red-black tree is predictable and does not depend some factor like hash table does on the quality of the hash function or the distribution of data.

7. *Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k,l\} : k \neq l \text{ and } h(k) = h(l)\}$?*

Solution: Let's define a binary random variable X_{ij}:-

$$X_{ij} = \begin{cases} 1 & \text{if there is a collision between the } i^{th} \text{ and } j^{th} \text{ keys during insertion,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_{ij}] = 1/m$ as the probability that the key that is inserted later gets the same slot as the one inserted earlier is $1/m$. Our hashing function is **uniform**, i.e. probability of getting any slot is equal and hence is $1/$(number of slots).

Let X be the random variable returning the number of collisions during insertion of the n keys, then,

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

This is because the RHS represents the sum over all possible distinct unordered pairs $\{\{k,l\}, k \neq l\}$ seeing if there is a collision within each pair or not. Hence,

$$\begin{aligned} E[X] &= E[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}] \text{ (Due to linearity of the Expectation operator)} \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{m} \text{ (Explained above)} \\ &= \frac{\binom{n}{2}}{m} = \frac{n(n-1)}{2m} \text{ (There are } \binom{n}{2} \text{ total unordered pairs)} \end{aligned}$$

Hence the expected number of collisions is $\frac{n(n-1)}{2m}$.