Data Structures and Algorithms

Raahim Salman

Academic Year 2019-2020

Contents

1 Week 1

1.1 Solving a problem

The solution of a problem has 2 parts:

- How to organize data
  - Data Structures:
    - A systematic way of organizing and accessing data
- How to solve the problem
  - Algorithm
    - step-by-step procedure for performing some task in *finite* time.

1.2 Linear Search

Assuming that the array below is a set of n different integers stored in non-decreasing order in an array L

0	1	2	3	4	5	6
3	9	11	12	15	19	27

Figure 1: Linear Search

How linear search works is that it has a target value of x and it starts iterating through the list until $val(i) \geq x$ (in the case of ordered), but if it is unordered then it traverses through the whole array $[0 - (n-1)]$

Algorithm 1: LinearSearch(L,n,x)

Input: Array L of size n and value x
Output: Position i, $0 \leq i \leq n$, such that $L[i] = x$ if $x \in L$ or -1 if x not in L

```
1 i ← 0
2 while (i < n) and L[i] ≠ x do
3 |   i++
4 |   if i == n then
5 |   |   return -1
6 |   else
7 |   |   return i
```

For an algorithm to be correct we need to show two things.

- The algorithm terminates
- The algorithm produces the correct output

1.2.1 Proof of termination

Proof. i takes on initial value of 0, and as we are iterating through the array we notice that in the loop i get incremented and therefore will take on values $0, 1, 2, \cdots, n$ and due to one of the while loop conditions being $i < n$ as soon as $i = n$ the loop will terminate and thereby can not perform more iterations. □

1.2.2 Proof of correctness

Proof. The algorithm compares x with every value in $L[1], L[2], \cdots, L[n-1]$ and if x is not in the given array implying that the value of $x = n$ has been assigned the algorithm will return -1. If for some value of $i, L[i] = x$, then that the index of that element is returned in the form of the value i. □

1.2.3 General Notes/Intuition

- This algorithm runs in $O(n)$ time and due to since it's implementation is often an array the follows time complexities are noted:
  - Access: $O(1)$
  - Search: $O(n)$ (Linear Search)
  - Insert: $O(n)$
  - Removal: $O(n)$
- When in doubt use, Linear search it is not a *bad* algorithm and it is quite good since it runs in linear time.
- Clues to spot for:
  - Array is unsorted
  - n is rather not large
  - instant indexing, unlike maps
  - better structure of information, and able to observe all elements
  - can search for object types not only primitive.

1.3 Binary Search

Search a *sorted* array by repeatedly dividing the search interval in half by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Main idea is that you're *abusing* the fact that the array is sorted to reduce the time complexity to $O(\log n)$

Algorithm: Compare x with the middle element, if x matches the middle element return the index of the middle element. Else if $x > [mid]$, then x *must* lie in the second half of the array thereby recur right, else if $x < [mid]$ then x *must* lie in the first half thereby recur left.

Algorithm 2: BinarySearch(L,x,first,last)

Input: Array L of size n and value x
Output: Position i, $0 \leq i \leq n$, such that $L[i] = x$ if $x \in L$ or -1

```
1  if first > last then
2  |  return -1
3  else
4  |  mid ← ⌊(first+last)/2⌋
5  if x = L[mid] then
6  |  return mid
7  else if x < L[mid] then
8  |  return BinarySearch(L,x,first,mid-1)
9  |  else
10 |  |  return BinarySearch(L,x,mid+1,last)
```

The important thing to note here is the recursive calls, and especially the $mid - 1, mid + 1$ The $mid - 1$ is called *when* $x < L[mid]$ implying that the value of x is smaller than the current middle element hence the element has to be in the left hand side of the array there by discarding the middle element and moving one to the previous middle. This shortens the search radius and hence the algorithm repeats itself.

1.3.1 Proof of Termination

Proof. If $x - L[mid]$ the algorithm will terminate immediately. If $x < L[mid]$ or $x > L[mid]$, the value of $L[mid]$ is discarded from the next recursive call and therefore decreases the size of L by at least 1 every recursive call proving there will be a finite number of recursive calls. After a finite number of recursive calls either $x = L[mid]$ or $L.size() = 0$ and therefore the algorithm ends. □

1.3.2 Proof of Correctness

Proof. The algorithm only discards values different from x, so if all the values of L are discarded it shows that x is not in L and correctly returns -1. Inversely if $x \in L$ then in some recursive call $x = L[mid]$ so the algorithm correctly returns mid □

1.3.3 General Notes and Intuition

If the word *sorted* shows up use binary search and *think* binary search.

- Time complexity is O(log n), this is the case as it divides the array by a factor of two every time, a proof of the time complexity is left as a exercise for the reader.

1.4 Comparing Algorithms and Time Complexity

There are five major ways to compare algorithms:

- Conceptual Simplicity

- Difficulty to implement
- Difficulty to modify
- **Running Time**
- **Space (Memory) Usage**

Complexity of an algorithm is defined as the amount of computer resources it uses, and since we care about Running Time, and Space Usage we have complexity functions

- Space Complexity: amount of memory that the algorithm needs.
- Time Complexity: amount of time needed by the algorithm to complete.

There are two major ways to measure time complexity, the draw backs for Experimental are listed below:

- Expensive
- Time consuming
- Results depend on input selected
- Results depend on the particular implementation

This problem is addressed by using a model of computation.

1.4.1 Model of Computation

Since we have to depend on assigning a theoretical value of the time complexity we have to have a model of computation that can determine how "expensive" each operation really is. Hence in this class we use the RAM model of computation.

Primitive Operation
A *basic* or primitive operation is that which will require a ***CONSTANT*** amount to complete in our model of computation. The following operations are examples of those that fit this model

$$\leftarrow, +, -, \times, /, <, >, =, \leq, \geq, \neq$$

Where constant is defined as independent from the size of the input.

Example: The algorithm *Linear Search*: The first line is an assignment statement and outside the while loop we can say it has time complexity c_1, in the while loop there are two comparisons, 1 assignment, 1 incremental, 1 "if" , 1 "else", and 2 return, lets assign all these times some constant c_2. Since all these are constant time operations we now are left with determining how many times each of these operations are being run due to the while loop. Since the main conditions of the while loop stipulate that it should run until $i = n$, we can conclude the while loop runs n times and therefore the operations that occur inside it (c_2) must also run n times. Hence the time complexity equation for this is $f(n) = c_1 + n \cdot c_2$

1.4.2 Asymptotic Notation

We want to characterize the time complexity of an algorithm for large inputs without worrying about implementation and or computer speed. Hence we try to determine the Asymptotic Behaviour of the time complexity function to determine what order the function is in.

Definition 1 (Asymptotic Notation)**.** *Let $f(n)$ and $g(n)$ if a constant $\exists c \in \mathbb{R}$, given $c > 0$ and a constant $n_0 \in \mathbb{Z} \geq 1$, such that the following inequality is satisfied: $f(n) \leq c \cdot g(n)$ $\forall n \geq n_0$*

The use of the word constant in this definition implies that neither the values of c, n_0 are dependent on n.

2 Week 2

2.1 Asymptotic Analysis

Asymptotic analysis is describing the properties of a function $f(n)$, as n becomes very large. In essence it is a method of describing the limiting behavior of a function. The current dominance relationships can be described below:

$$n! > 2^n > n^3 > n^2 > n \log n > n > \log n > 1$$

There *must* be care placed when working with in Big-Oh notation. A good tip is to always revert back to the definition of what it means to be the worst run case of an algorithm. Intuitively the worst case will *SHOULD* serve as the upper bound for all other runtimes for an algorithm. Hence based on this fact that a certain time complexity is a certain order if and only if the order multiplied by the constant is bigger, serving the worst case, than the time complexity function. This is demonstrated with a few examples below:

Is $2^{2n+1} = O(2^n)$
By this logic $f(n) = O(g(n))$ iff there exists a constant c such that for some sufficiently large n_0 such that.

$$\begin{aligned} f(n) &\leq c \cdot g(n) \\ 2^{n+1} &= 2^n \cdot 2^1 \\ 2^1 \cdot 2^n &\leq c \cdot g(n) \\ 2^1 \cdot 2^n &\leq c \cdot 2^n \end{aligned}$$

If we are to select $c \geq 2$, then this inequality will hold and therefore the upper bound of the function is indeed any $c_1 \geq 2$ that emulates the worst behavior of this time complexity function.

Is $(x + y)^2 = O(x^2 + y^2)$

Proof. By definition this expression is valid iff there exists c, such that $(x + y)^2 \leq c \cdot (x^2 + y^2)$, since $(x + y)^2 = x^2 + 2xy + y^2$, without the $2xy$, the inequality can be easily proven but since there is a $2xy$, we need to relate the $2xy$ to $(x^2 + y^2)$. If $x \leq y$, then $2xy \leq 2y^2 \leq 2(x^2 + y^2)$, in the case $x \geq y$, then $2xy \leq 2x^2 \leq 2(x^2 + y^2)$ Therefore we can determine that this inequality is bounded by 2 times $(x^2 + y^2)$, so that means $(x + y)^2 \leq 3(x^2 + y^2)$ □

2.1.1 Identities to be aware of

Adding Functions

$$O(f(n)) + O(g(n)) \Rightarrow O(\max(f(n), g(n))$$

This is important as it allows us to quickly pick out the highest order and determine that is the behavior of the overall function.

Multiplying Functions

$$O(c \cdot f(n)) \Rightarrow O(f(n))$$

This is important as it allows us to get rid of all constants (given that $c > 0$)

$$O(f(n)) \cdot O(g(n)) \Rightarrow O(f(n) \cdot g(n))$$

This is important as it shows that the if two functions occur at the same time within each other, their effects are multiplied by each other.

2.1.2 How to compare algorithms

Comparing algorithms is done by looking at their asymptotic behaviour, but there are some caveats. The asymptotic behaviour shows the running time of *EVERY* implementation of that algorithm. This follows the intuition we have been building so far that the speed of an algorithm is bound between its upper (O representing the worst case) bound, and its lower bound (Ω representing the best case). Hence every implementation of an algorithm should lie

below or at its upper bound. Based on this we can compare algorithms based on their Asymptotic behaviour.

Example:
Algorithm A has asymptotic behaviour $O(f(n))$
Algorithm B has asymptotic behaviour $O(g(n))$

This results in two cases:

Case 1: $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$

If this is the case that means that both algorithms have the same asymptotic behaviour and thereby have the same set of possible running times and this implies that $O(f(n)) = O(g(n))$

Case 2: $f(n)$ is $O(g(n))$ and $g(n)$ is **NOT** $O(f(n))$.

This implies that $f(n)$ has the same asymptotic behaviour as $g(n)$, **BUT**, $g(n)$ does **NOT** have the same asymptotic behaviour as $f(n)$. This is only possible if the order of $g(n) > f(n)$, as $f(n)$ is a subset of the behaviour of $g(n)$, yet $g(n)$ is not a subset of the behaviour of $f(n)$, there by we can conclude that $f(n)$ is contained in $g(n)$, and hence $O(f(n)) \subset O(g(n))$.

2.2 Execution of a Recursive Algorithm

When executing an algorithm that is recursive the computer is used as an execution stack. The execution stack is needed to be able to execute algorithms that invoke other algorithms. When a certain algorithm is invoked, part of the execution stack is reserved to store *local variables, parameters, return address, return value.* This portion is called an activation record or a frame.

This paper will walk through the Binary Search algorithm as an example, it is referenced above as well as $L = \{4, 7, 11\}$, and the target is 11. The main method executes the Binary Search algorithm under the variable *pos*. Currently the return address is set to END, as the main method is void method and does not have a return type, therefore this character will tell the program to terminate.

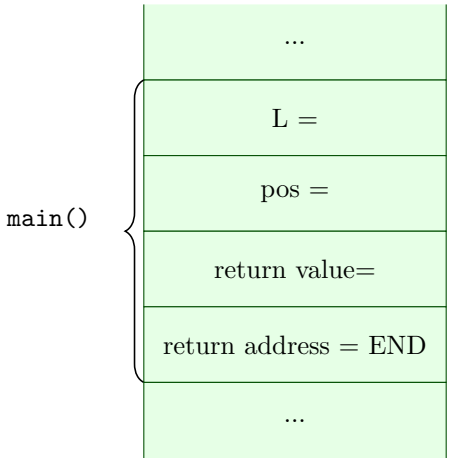


Figure 2: Execution Stack for a recursive algorithm 1

Once the activation record has been created the execution of the `main()` starts. First an array $L = \{4, 7, 11\}$ is created and the address of is stored in L, let that address be denoted by A_4. Next line of execution Binary Search is invoked, every time the method calls it's self a new execution stack is created. Since the BinarySearch algorithm needs to have (L,x,first, last) and the local variable mid, as well as the address and the return value.

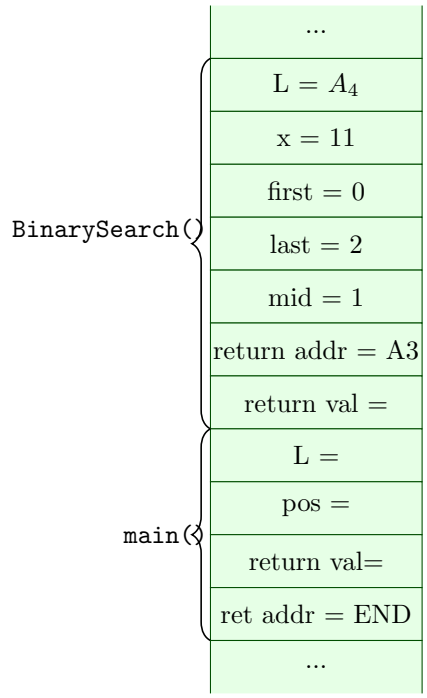


Figure 3: Execution Stack for a recursive algorithm 2

Once the activation record is created the values of the parameters are stored and the return address is also assigned. Upon completion of the creation of the activation record the execution of the algorithm starts. Since our initial condition of $first > last$ is false the algorithm will compute mid as $(0 + 2)/2 = 1$ Then as x is larger than $L[1]$ the second recursive call is invoked. For the sake of diagram simplicity this paper will not show the complete stack, but nothing is removed until explicitly denoted.

For the second recursive another activation record is created and the values of the parameters are stored as demonstrated, by the figure. Since this is a stack the only activation record that is active is the top one.

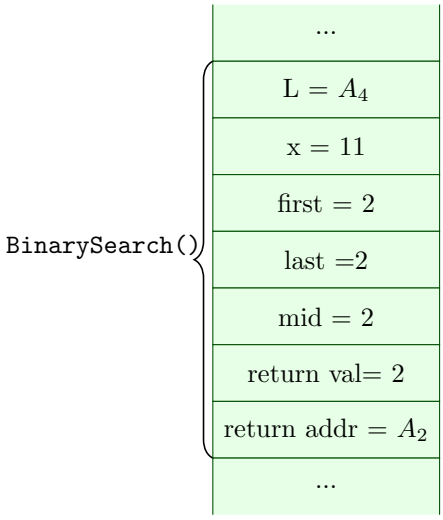


Figure 4: Execution Stack for a recursive algorithm 3 (Last method called)

Since $first == last$ the first else statement is executed and the value of mid is computed to 2, and then x is compared to $L[2]$ and since both values are the same, the algorithm returns mid. When the return statement is executed, first the value of *mid* is stored in the activation record as the return value, and the activation record is popped off the stack. Since the return addr of the first instance is A_2, and that has a return value of 2, the value 2 is stored in the

return val for the first instance of BinarySearch(). The value returned by BinarySearch() is 2, which is then finally stored in pos. Since end is reached after that the program terminates and all elements are popped off of the stack.

3 Week 3

3.1 Time Complexity functions for various algorithms

3.1.1 Time Complexity of LinearSearch()

For reference here is the LinearSearch() algorithm.

Algorithm 3: LinearSearch(L,n,x)

Input: Array L of size n and value x
Output: Position i, $0 \leq i \leq n$, such that $L[i] = x$ if $x \in L$ or -1 if x not in L

```
1 i ← 0
2 while (i < n) and L[i] ≠ x do
3 |   i++
4 |   if i == n then
5 |   |   return -1
6 |   else
7 |   |   return i
```

To formally compute the time complexity we have to derive a function for the time complexity. Let $f(n)$ denote such a function. In line one that is outside the *while* loop we see the assignment statement. Let c_1 denote that assignment statement. So far $f(n) = c_1$. In line 2, we observe that the while loop has 3 conditions, $((i < n)$, **and**, $(L[i] \neq x))$ and since these are also all constant time operations let us denote these with c_2, this is different from c_1, as the number of times c_2 is performed is dependent on the number of times the while loop runs. Furthermore in line 3, there is one more constant time operation, making that $2c_2$, the "if" statement in line 4 and its conditional statement are 2 more constant time operations hence the tally is up to $4c_2$. What is remaining are two return statements which are also constant time operations, but in the worst case only one of them gets hit which is the return -1, therefore we increment the tally to $5c_2$. Now to compute the full function we have to determine the number of times the while loop runs. Since one of the conditionals is $i < n$, and i is getting incremented every time the loop runs, in the worst case this while loop will run until $i = n$, thereby performing $5c_2$ operations n times. Therefore the time complexity of this algorithm can be described as:

$$f(n) = c_1 + 5nc_2$$

Now to compute the asymptotic behaviour we can use our identities from above, since $5nc_2$ contains an n the max of the two is $5nc_2$, and furthermore with the second identity, we can drop the 5 and be left with nc_2 thereby demonstrating that the asymptotic behaviour can be described by $O(n)$

3.1.2 Time Complexity of BinarySearch()

For reference here is the BinarySearch() algorithm.

Algorithm 4: BinarySearch(L,x,first,last)

```
Input: Array L of size n and value x
Output: Position i, 0 ≤ i ≤ n, such that L[i] = x if x ∈ L or -1
1  if first > last then
2  |  return -1
3  else
4  |  mid ← ⌊(first+last)/2⌋
5  if x = L[mid] then
6  |  return mid
7  else if x < L[mid] then
8  |  return BinarySearch(L,x,first,mid-1)
9  |  else
10 |  |  return BinarySearch(L,x,mid+1,last)
```

As the same of linear search, the worst case for binary search is when the element is not in the set L. A strategy for solving this would be to use a recurrence equation and then solving the equation. For this we would need two cases, one base case, the other recursive case. Let $f(n)$ denote the number of operations that the algorithm BinarySearch needs to perform when the size of the input is n. By this logic $f(0) - c'$, where c' are the number of constant time operations performed when the array size is 0. When the size of the array is larger than 0 the algorithm will perform c number of operations to find the element in the middle of L,compared that with x, and decide which way to split. We make the assumption that both halves of the array have the same size $\frac{n-1}{2}$. Hence the total number of operations that the algorithm performs when $n > 0$, is $f(n) = c + f((n-1)/2)$. This results in the following recurrence equations:

$$f(0) = c'$$
$$f(n) = c + f\left(\frac{(n-1)}{2}\right) if n > 0$$

We can solve this by repeated substitution.

$$\begin{aligned}
f(n) &= c + f\left(\frac{n-1}{2}\right) \\
f\left(\frac{n-1}{2}\right) &= c + f\left(\frac{\frac{n-1}{2}-1}{2}\right) \\
&= c + f\left(\frac{n-2^0-2^1}{2^2}\right) \\
f\left(\frac{n-2^0-2^1}{2^2}\right) &= c + f\left(\frac{n-2^0-2^1-2^2}{2^3}\right) \\
&\vdots \\
f\left(\frac{n-2^0-2^1-\cdots-2^j}{2^{j+1}}\right) &= c + f\left(\frac{n-2^0-2^1-\cdots-2^{j+1}}{2^{j+2}}\right)
\end{aligned}$$

Since the value of the argument on the terms in the left hand side decreases with each new equation and since we proved earlier that this algorithm is finite the term will eventually = 0.

$$f\left(\frac{n-2^0-2^1-\cdots-2^{j+1}}{2^{j+2}}\right) = 0$$

Since $f(n)$ is the cumulative sum of all of its previous terms we can write $f(n)$ such as.

$$\begin{aligned} f(n) &= c + c + c + c + \cdots + c + f\left(\frac{n - 2^0 - 2^1 - \cdots - 2^{j+1}}{2^{j+2}}\right) \\ &= c + c + c + c + \cdots + c + f(0) \\ &= c + c + +c + c + \cdots + c + c' \end{aligned}$$

Since the repeated equations have $j + 2$ equations, the number of c terms will also be $j + 2$, hence the expression is

$$f(n) = (j + 2)c + c'$$

To determine a value for j, we have to simplify this expression.

$$\begin{aligned} \left(\frac{n - 2^0 - 2^1 - \cdots - 2^{j+1}}{2^{j+2}}\right) &= 0 \\ n &= 2^0 + 2^1 + 2^2 + \cdots + 2^{j+1} \\ &= \sum_{i=0}^{j+1} 2^i \\ &= 2^{j+2} - 1 \\ \log_2(n + 1) &= j + 2 \end{aligned}$$

Therefore after taking the logarithm we can see that

$$f(n) = c \log_2(n + 1) + c'$$

Ignoring the constant terms as shown in the identities we can finally conclude that $f(n) = O(\log n)$

3.2 Abstract Data Types

Abstract Data Types are user defined data types that contain two parts.

- A name or type, specifying a set of data (i.e. dict or map)
- Descriptions of all the operations or methods that manipulate the data type

The descriptions indicate *what* the operations do, not **HOW** they do it. This is most similar to an interface in `Java`. It is the preferred way of designing and implementing data structures. 2 general principles are used: Information hiding & re-usability.

3.2.1 Information Hiding

The program or algorithm using the data structure should **not** care about how the structure is implemented i.e. the details of its implementation. Implementation should be interchangeable between various languages/programmer preferences, and therefore implementation information *should* be hidden.

3.2.2 Re-usability

If the data structure is useful for one application more than likely it is also useful for other applications and therefore we should design it to be as re-usable as possible.

3.2.3 ADT for a dictionary/map

- `get(key)`: returns the data associated with the given key, or **null** if no record has the given key.
- `put(key,value)`: inserts a new record with the given key data, or throws an error if the map already contains a record with the given key.
- `remove(key)`: removes the record of the given key, or throws an error if there is no record with the given key.

Outlined below is the interface for a dictionary in `Java`, this shows the methods required to interact with the data structure, `K,V` represent generic types and can be substituted with any object or primitive type.

```
public interface Dictionary<K,V> {
    public V get(K key);
    public void put(K key, V value) throws DuplicatedKeyException;
    public void remove(K key) throws NoKeyException;
}
```

Figure 5: Java interface for a Dictionary

3.3 Dictionary/Map

A dictionary/map models a searchable collection of key-value pairs. The main operations are for searching, inserting, and deleting items. Multiple entries are **NOT** allowed.

A `Node[]` or `LinkedList` is often the implementation of the this data structure. Below the algorithms for the three operations described above are defined.

3.3.1 The `get(V key)` algorithm

```
public V get(K key){
    p = header;
    while p is not null do{
        if p.element().getKey() = k then   //The .getKey() method is a method defined by object V
            return p.element().getValue()  //The .getValue() method is a method defined by object V
        else p = p.next();
    return null;                           //No value with key = to k was found.
    }
}
```

Figure 6: The `get(k)` Algorithm

Some things to note here, while the current node which is defined by `p` is not null the rest of the algorithm get executed. Furthermore the whole list has to be traversed to find an element making this a $O(n)$ operation.

3.3.2 The `put(K key, V value)` algorithm

```
public void put(K key, V value) {
    p = header;
    while p is not null do
        if p.element.getKey() = k then ERROR
        else p = p.next()
    p = new node storing (k,v)
    p.setNext (header)
    header = p
    n = n + 1
}
```

Figure 7: The `put(k key, V value)` algorithm

Some things to note on how this algorithm works, since we are inserting an element into the linked list we have to traverse the linked list and make sure the key we are trying to insert is not already inside the linked list. If `key` is not in the linked list we create a new node and assign it to the variable p, and set the next node to p the header from

the current linked list, then we assign the header to the newly added node p. Furthermore since all the operations other than the while are constant time this is another $O(n)$ algorithm.

3.3.3 The `remove(K key)` algorithm

```
public void remove(K key) {
    p = header;
    prev = null;
    while p is not null do
        if p.element.getKey() = k then {
            if prev is not null then
                prev.setNext(p.next())
        else header = p.next()
        n = n - 1                          \\Lower number of entries by 1
    }
    else{
        prev = p
        p = p.next()
    }
}
```

Figure 8: The `put(k key, V value)` algorithm

To expand on this algorithm a little bit, we can note again that it must traverse the full array to find the specific key we are looking to remove, and hence will result in $O(n)$ time complexity, furthermore if the element is found, and it is in the middle of the linked list, the links to that node are severed as the previous node's next pointer is set to the next node from p, furthermore if it is the first element in the list (i.e prev is null) then the header just becomes p.next.

4 Week 4

4.1 Hash Maps

As mentioned above a dictionary can be implement with different data structures, but all have $O(n)$ time complexity (except for `get()` implemented with a sorted array), to over come this design flaw hash maps are used. For a good hash function you need three things:

- Low time complexity
- Causes few collisions
- Must spread keys *uniformly* across entire table

4.1.1 Hash Function

The input to the hash function is the key component of the dictionary, and therefore can be of any type : int, char, string,[ ].

Hash Code
The hash code maps the key to a very **Large** integer to ensure all the entries of the hash map are used. Once this is accomplished the next step is to put this resulting integer through a compression map. Oftentimes this requires converting a string into an integer, this is usually done by casting such as:

$$(int)\ char \rightarrow int$$

Then, apply some functions to the integer corresponding to the characters of the string to produce a single integer from the whole string.

$$g(\text{“}C_{k-1}C_{k-2}\cdots C_1C_0\text{”}) = \sum_{i=0}^{k-1}(int)\ C_i$$

This is not a good hash code as it produces small integer values, the reason for this is ASCII has integers ranging from 0-127, and the sum will be composed of k values from 0-127. Even though for longer strings the number *could* get big it still is not big enough.

Compression Map
This is a function that takes in a very large integer and maps its evenly into a position on the hash map often the output is restricted to a from 0 - $n-1$ that is the size of the array.

4.1.2 Polynomial Hash Code

Due to the aforementioned problem we have to find a different way to hash our strings. A proposal to solve this problem is the polynomial hash code. This is done quite similarly as the above procedure except the sum of the (int) char is not being taken, instead they serve as coefficients for a polynomial function.

$$S = \text{“}C_{k-1}C_{k-2}\cdots C_1C_0\text{”}$$

Say this is our string S, now for every char in this string we take the int cast and multiply it by x^{k-1} power, where k is the number of letters. A good choice for x, is often a prime in the range $\{33, 37, 39, 41\}$. Furthermore care must be taken when implementing this function as its computation might produce very large numbers that might cause a problem either via computing or via production of incorrect values.

$$g(\text{“}C_{k-1}C_{k-2}\cdots C_1C_0\text{”}) = ((int)\ C_{k-1}x^{k-1} + (int)\ C_{k-2}x^{k-2} + \cdots + (int)\ C_1x^1 + (int)\ C_0)$$

This is what the hash code should look like when computing a polynomial hash code. This equation can be further simplified using **Horner's Rule** for solving polynomials.

$$g(\text{“}C_{k-1}C_{k-2}\cdots C_1C_0\text{”}) = (int)\ C_0 + x\left((int)\ C_1 + x\left((int)\ C_2 + x\left(C_3 + \cdots +\right)\right)\right)$$

The equation doe look very intimidating, yet think of it as a Russian doll, each layer is the last layer multiplied by x, hence as you see $x^2C_2 = x(C_1 + x(C_2))$ This goes on until $k-1$ is achieved.

Now in-terms of practicality, since the value of this hash code will be very large the compression map will be a modulo function with a prime integer as the size of the hash map. The size of the hash map M is prime due to reasons from number theory. Hence our final polynomial **Hash Function** is the hash code with a compression map:

$$g(\text{“}C_{k-1}C_{k-2}\cdots C_1C_0\text{”}) = ((int)\ C_{k-1}x^{k-1} + (int)\ C_{k-2}x^{k-2} + \cdots + (int)\ C_1x^1 + (int)\ C_0) \mod M$$

4.1.3 hashFunction() Algorithm

Algorithm 5: hashFunction(S,x,M)

Input: String S = $g(\text{“}C_{k-1}C_{k-2}\cdots C_1C_0\text{”})$, value x, and table size M
Output: position for S in hash table

```
1 val ← (int) C_{k-1}
2 for i = k − 2 to 0 do
3 |  val ← (val × x + (int)C_i) mod M
4 return val mod M
```

Quickly solving for the time complexity of this algorithm we see line 1,4 are both constant time operations which will require c_1 time, while inside the for loop there is another c_2 operation being performed, yet since the for loop runs for $(k-1)$ hence the equation is $c_1 + (k-1)c_2$ which is $O(k)$, and there by $O(1)$, since k is constant.

4.2 Handling Collisions

First of all what is a collision? A collision occurs when different elements with different hash codes map to the same position in the array due to an unforeseen reason. Now there are multiple ways to handle these collisions each with their own pros and cons.

4.2.1 Seperate Chaining

Separate chaining is a technique that handles collisions by letting each cell in the table point to a linked list of entries that are all mapped to the same index. Hence think an array of linked lists. Visually speaking this is what one would look like:

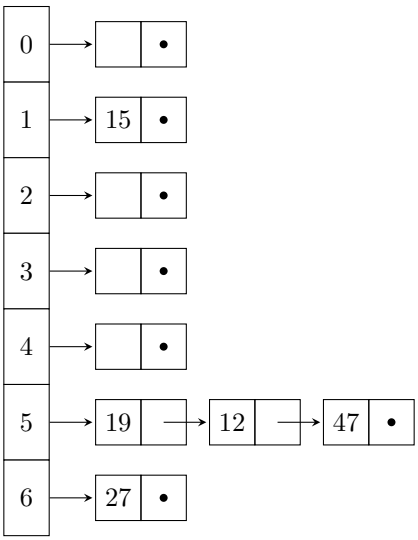


Figure 9: Hash table of the following hash function: $h(x) = k \mod 7$ while $k = 19, 27, 12, 47, 15$ in that order, with collisions being handled through separate chaining.

The worst case for `get(k), put(k,v), remove(k)` is $O(n)$, due to traversal of the whole linked list, and this furthermore also takes up too much memory due to there being more objects being created. The algorithm for `get(k)` is written below and implemented with separate chaining.

Algorithm 6: get(key)

Input: Key
Output: Value from key or null if key is not in the table

```
1 p ← T[h(key)]
2 while p ≠ null and p.getKey() ≠ key do
3 |  p ← p.getNext()
4 if p = null then
5 |  return null
6 else
7 |  return p.getValue()
```

Time Complexity Analysis
Lets do a quick time complexity analysis of this algorithm. Line 1, 4-7 all perform constant time operations which we can denote as c_1 inside the while loop in line three another constant time operation which we denote as c_2 also occurs. The while loop iterates through the entire list meaning that the number of iterations is the length of the list, therefore the worst case time complexity which can be attributed to a bad hash function which maps all values to one element will be $O(n)$

4.2.2 Open Addressing

Other than separate chaining there is another way of handling collisions: Open Addressing, which involves placing the colliding item in a different cell in the table.

Linear Probing
Handles collisions by placing the colliding item in the next available table cell. Consider this example:

$$h(x) = x \mod 7$$

Here are the results from linear probing (summarized):

1. $h(19) = 19 \mod 7 = 5$
2. $h(27) = 27 \mod 7 = 6$
3. $(h(12) + 2) \mod 7 = (5 + 2) \mod 7 = 0$
4. $(h(47) + 3) \mod 7 = (5 + 3) \mod 7 = 1$
5. $(h(15) + 1) \mod 7 = (1 + 1) \mod 7 = 2$

0	1	2	3	4	5	6
12	47	15			19	27

Figure 10: Hash table of the following hash function: $h(x) = k \mod 7$ while $k = 19, 27, 12, 47, 15$ in that order, with collisions being handled through linear probing.

Linear probing follows a unique pattern to apply for when you observe a collision:

$$h(k), (h(k) + 1) \mod M,\ (h(k) + 2) \mod M,\ (h(k) + 3) \mod M,\ \cdots$$

A drawback of linear probing is that colliding items are often clumped together, causing future collisions to cause a longer sequence of clustering (i.e. from the example observe indices 0,1,2)

Algorithm 7: get(key)

Input: Key
Output: Value from key or null if key is not in the table

```
1  pos ← T[h(key)]
2  c ← 0
3  while T[pos] ≠ null and T[pos].getKey() ≠ key do
4  |   pos ← (pos + 1) mod M
5  |   c ← c + 1
6  |   if c = M then
7  |   |   return null
8  if T[pos] = null then
9  |   return null
10 else
11 |   return T[pos].getValue()
```

This is the get method implementing linear probing to handle collisions. To walk through the algorithm lets first understand what it is trying to do, and then end with calculating it's time complexity. The variable pos takes on the role of the index where which the key is mapped to from the hash function. The first condition on the while loop ensures that there already exists a *unique* key already in the place of pos, that is not the input key, After then it keeps adding 1 to both the pos and c as a way to make sure that the while loop is able to exit. If $c = M$, it means the whole table is full and therefore return null. Furthermore if the index at the array is already empty there is no value to return hence null is returned, and in the last case the value of the key is returned.

Now lets compute the time complexity of this algorithm, we start by observing lines 1-2, and 8-11 having c_1 operations and inside the while loop from lines 4-7 there being c_2 operations that are run, the number of times the while loop is run. The worst case for the while loop is when the array is full and therefore it will iterate M times, but M is the size of the hashmap and therefore the while loop iterates n times. The time complexity function is given by

$$f(n) = c_1 + n \cdot c_2$$

Double Hashing

Double hashing uses a secondary hash function $h'(x)$ and handles collisions by placing in item in the first available cell from the following series:

$$h(x), (h(x) + h'(x)) \mod M,\ (h(x) + 2h'(x)) \mod M,\ (h(x) + 3h'(x)) \mod M,\ \cdots,$$

For double hashing to work the size of the hashmap M ***MUST*** be prime. This is due to a number theory issue which would cause a cycle for the value of 10.

Here are the results from double hashing summarized:

1. $h(19) = 19 \mod 7 = 5$
2. $h(27) = 27 \mod 7 = 6$
3. $(h(12) + h'(12)) \mod 7 = (5 + 3) \mod 7 = 1$
4. $(h(47) + 2 \cdot h'(47)) \mod 7 = (5 + 6) \mod 7 = 4$
5. $(h(15) + 3 \cdot h'(15)) \mod 7 = (1 + 15) \mod 7 = 2$

0	1	2	3	4	5	6
	12	15		47	19	27

Figure 11: Hash table of the following hash function: $h(x) = k \mod 7$ while $k = 19, 27, 12, 47, 15$ in that order, with collisions being handled through double hashing.

4.2.3 Average Time Complexity of `get()` Operation

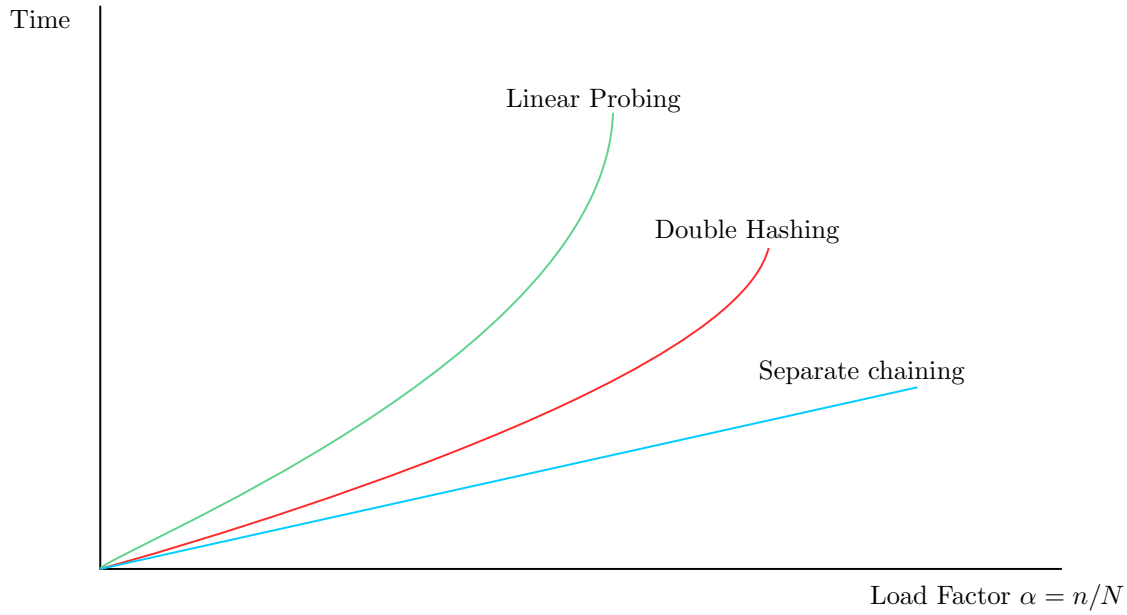


Figure 12: Average Time complexity of `get()` Operation

Here we can see the average load factor and time for the get operation to work as we start to scale up the load factor. Separate chaining is the still the best in terms of time complexity as if the hash function is defined with the intent of it being well defined and having few collisions the linked list traversal would not be that long, otherwise the other two have to compute a lot more to arrive to their new address, Linear probing is the highest due to its clustering problem if the item we are trying to retrieve is near the end of the cluster there will be lots of computation.

5 Week 5

5.1 Trees

A tree is an abstract model of a hierarchical structure.

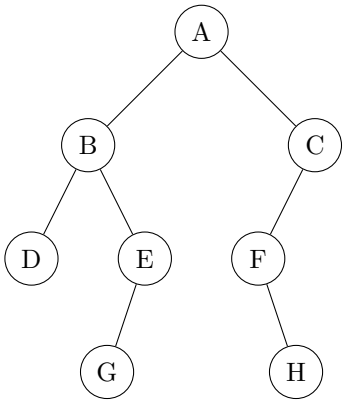


Figure 13: Tree Diagram

5.1.1 Tree Terminology

Now for some terminology, A is called the root note, and B is the parent of D and E, while C is siblings with B The line connecting the two nodes, are called edges, or links. While each circle is called a node.

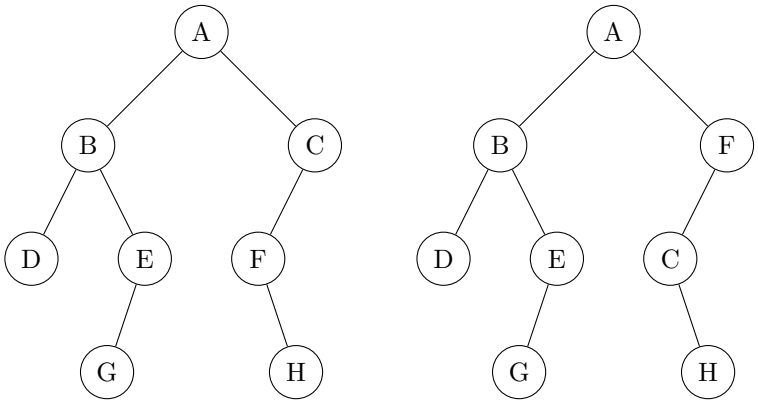
Ancestors are referred to the "generation" above: so B and C, share A as an ancestor, while A has no ancestors. *Internal Nodes* are nodes with atleast one children, and *External Nodes/Leaves* are nodes without children, this would be G and H in our example.

Depth or Level of a Node
This is the number of ancestors, the depth of E, is 2. This is because B, A are both ancestors of E.

Height of a tree
This is the maximum depth of any node. The tree in our example has a maximum depth of 3. This is because G, H our leaf nodes both have 3 ancestors. the height will be the maximum number of ancestors.

Degree of a node
The degree of a node is the number of children it has. The degree of B is 2, and the degree of E is 1.

Subtree
A subtree is a tree consisting of a node and its descendants.



(a) Ordered Tree (b) Unordered Tree

Figure 14: Ordered and unordered tree

Tree Properties

Number of edges = number of nodes - 1

This property holds true as the root is not connected to any edges, yet every other node has an edge associated to it, therefore to compute the number of edges you take the number of nodes less 1.

5.1.2 Traversing Algorithms

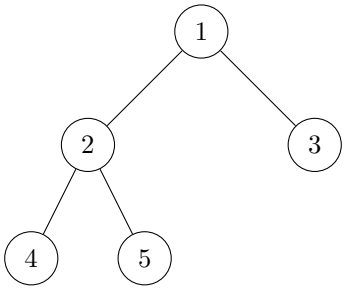


Figure 15: Example Tree

Preorder Traversal
In preorder traversal, a node is visited before its descendants.

```
Algorithm Preorder
1. Visit the root
2. Traverse the Left Subtree (i.e. by calling Preorder(left-subtree))
3. Traverse the Right Subtree (i.e. by calling Preorder(right-subtree))
```

Uses for Preorder Traversal

- Useful for when creating a copy of the tree
- When evaluating a pre-fix expression on an expression tree.

The example tree will be traversed as 1,2,4,5,3.

Postorder traversal
In postorder traversal, a node is visited *after* its descendants.

```
Algorithm Postorder
1. Traverse the Left Subtree (i.e. by calling Preorder(left-subtree))
2. Traverse the Right Subtree (i.e. by calling Preorder(right-subtree))
3. Visit the Root.
```

Uses for postorder traversal

- Useful for when deleting the tree
- Useful to get the *postfix* expression from an expression tree.

The example tree will be traversed as: 4,5,2,3,1.

Inorder traversal
In inorder traversal, a node is visited *after* its left subtree, but *before* its right subtree.

```
Algorithm Postorder
1. Traverse the Left Subtree (i.e. by calling Preorder(left-subtree))
2. Visit the Root.
3. Traverse the Right Subtree (i.e. by calling Preorder(right-subtree))
```

Uses for inorder traversal

- In the case of sorted tree, inorder traversal gives elements/nodes in non decreasing order.

The example tree will be traversed as: 4,2,5,1,3.

5.2 Binary Trees

A binary tree is very specific type of tree. Its properties are outlined below:

5.2.1 Properties

- Internal nodes have ≤ 2 children (exactly two for *proper* binary trees)
- The children of a node are an ordered pair
- # of nodes at level $i \leq 2^i$
  - Level 0 – 2^0 Nodes
  - Level 1 – 2^1 Nodes
- # of leaves $\leq 2^{height}$
- $\log_2(\#\text{leaves}) \leq \text{height} \leq \#$ of internal nodes

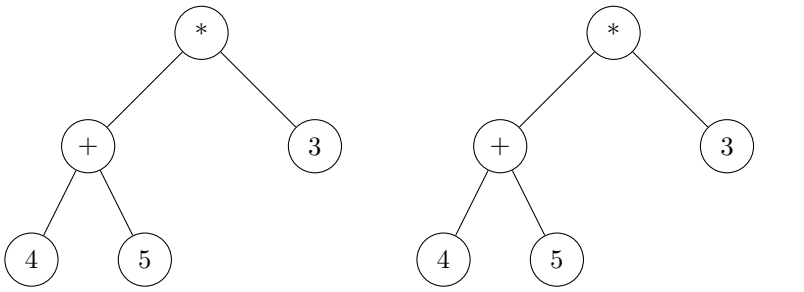
The children of an internal node are called left child, and right

Claim 1. *# of leaves = # of internal nodes + 1*

Proof. Let n = number of nodes, and since it follows that the number of edges are $n-1$, it follows again that the number of edges is also 2(number of internal nodes) as each internal node has 2 edges coming out of it. Therefore, $n-1 = 2$(number of internal nodes) Since n = number of nodes, which can be redfined as number of leaves + number of internal nodes, therefore it follows that the number of leaves = the number of internal nodes + 1. □

5.2.2 Types of Binary Trees

Arithmetic Expression Tree Binary tree associated with an arithmetic expression i.e. prefix and postfix expressions.



(a) prefix expression = *,+,4,5,3 (b) postfix expression = 4,5,+,3,*

Figure 16: Example Tree

Another example of a binary tree is a *Decision Tree*, which has the internal nodes as yes/no questions and the leaf nodes as decisions, meaning you follow you decision down to the leaf.

6 Week 6

6.1 Tree Algorithms

6.1.1 `Height()` Algorithm

Recurrence equation for the height of a tree.
height(r) = 0 if r is a leaf

height(r) = max{height of a subtree} + 1 if r is internal

Algorithm 8: Height(r)

```
   Input: Root r of a tree
   Output: Height of the tree
1  if r is a leaf then
2  |  return 0
3  else
4  |  mh ← −1
5  |  for each child c of r do
6  |  |  h ← height(c)
7  |  |  if h > mh then
8  |  |  |  mh ← h
9  |  return mh + 1
```

Analysis of Algorithm

This paper will analyze the algorithm according to this tree:

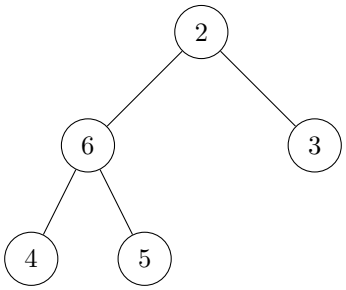


Figure 17: Tree for analyzing.

Visually we can see the height of the tree should be 2, as both 4,5 have 2 ancestors each while 3 only has 1.

This algorithm is designed to find the height of a tree, given its root r. Recall that the height of a tree is the maximum depth, meaning the maximum number of ancestors. The algorithm `height(r)` is a recursive algorithm, where we start at the root of the tree, and since the 2 is not a leaf, the `else` statement will get called and mh gets assigned to -1. For each child of 2 $\cdots$. Lets start with the left side of the tree. h gets assigned a function call for $height(c)$, where c is node 6. So again we head to the top, since 6 is not a leaf we get bumped into the `else` statement. Where $mn \leftarrow -1$ and $h \leftarrow height(c)$, where c is the node 4 (left most node). Now that since c is a leaf node the last function call will return 0 and since $0 > -1$, $mh = 0$. Then c increments to the second child of 6 and the same thing happens, but the if statement doesn't get called as $0 \not> 0$. Then we go back up node 6, since the whole subtree is done the statement returned is $mh + 1 = 0 + 1$. Since $1 > mh$ then $mh \leftarrow 1$, and then we move on node 3 (last child of r). Since 3 is a leaf node, 0 gets returned and the if statement fails the condition. The for loop exits and $mh + 1$ is returned which is 1+1=2. Which aligns with what we observe visually.

Time Complexity of this Algorithm

Lines 1,2 are both constant time operations so we denote it to be c_1, Inside the `else` block we see lines 4, 9 to be more constant time operations which we will denote as c_2. Inside the for loop we see 3 constant time operations lines 6-8, which we will denote as c_3, while ignoring the recursive call. Since the `for` block is dependent on the children of r, we can say the loop runs $degree(r)$ times. This leaves us with c_1 operations in the base case, and $c_2 + c_3 \cdot degree(r)$ for the recursive case. What is left for us to determine is the number of calls performed per node, and based on learning how the algorithm runs we can determine that one call is performed per node. Now we can compute the

total number of operations

$$\sum_{leaves} c_1 + \sum_{internal\ node(r)} (c_2 + c_3 \cdot degree(r))$$
$$= (leaves) \cdot c_1 + (internal\ node) \cdot c_2 + c_3 \sum_{internal\ node} degree(r)$$
$$= (leaves) \cdot c_1 + (internal\ node) \cdot c_2 + c_3(n-1)$$

Since the number of leaves and internal nodes are both constant we end up with a time complexity of $O(n)$. The $n-1$ expression comes from the fact we have to sum the children of every internal node, and since the root node is considered an internal node that results in the number of edges. Which we have an expression for $n-1$.

6.1.2 `TOC()` Algorithm

We will begin writing this algorithm, and then analyze how it works, its space complexity. Furthermore this algorithm is called the TOC algorithm, as in Table of Contents, and it prints out the structure of the tree as if it were a book.

Algorithm 9: TOC(r, indentation)

Input: Root r of a tree representing the structure of a book, integer indentation (in the initial call to the algorithm the value of the indentation is 0)
Output: Print table of contents properly indented.

```
1 for i ← 1 to indentation do
2 |  print (" ")
3 print r.data
4 for each child c of r do
5 |  TOC(c, indentation + 1)
```

This algorithm works by printing the value of the root node, and then adding an indentation for each generation you go down. It represents a book structure. The root node is level one, its children are level 2, and its grandchildren are level 3, and so on.

Space Complexity Of This Algorithm
Space complexity is the amount of memory needed for execution stack and for data structures. We need to find the maximum number of activation records simultaneously in the execution stack is equal to the height of the tree + 1.

6.1.3 `diskSpace()` Algorithm

This is an algorithm to compute the total space used by a certain file system.

Algorithm 10: diskSpace(r)

Input: root r of a file system tree
Output: Total space used by the file system

```
1 s ← 0
2 for each child c of r do
3 |  s ← diskSpace(c) + s
4 return s + r.space
```

Lets determine what this algorithm actually does. It takes in a the root of a file system tree, and assigns the variable s to 0, and traverses it's children while calling it's self on each of its children. Until it traverses to the left most leaf node it will return 0 + *r.space*, which will then get updated for the variable s. Furthermore, after that it will then run the algorithm on the right child of the node. Until it reaches another leaf node. Thereby it sums up all the *r.space* calls from every visited leaf node and then works it's way back up the tree.

Time Complexity of `diskSpace()`

We see that liens 1,4 have constant time operations for which we denote c_1 the for loop iterates through every child of r therefore runs a $degree(r)$ times, and inside there is a constant time operation, (c_2) (ignoring the recursive call). Therefore we end up with the following expression

$$\begin{aligned} f(n) &= \sum_{nodes} (c_1 + c_2 \cdot degree(r) \\ &= \sum_{nodes} c_1 + c_2 \sum_{nodes} degree(r) \\ &= nc_1 + c_2(n-1) \end{aligned}$$

Hence this algorithm is $O(n)$

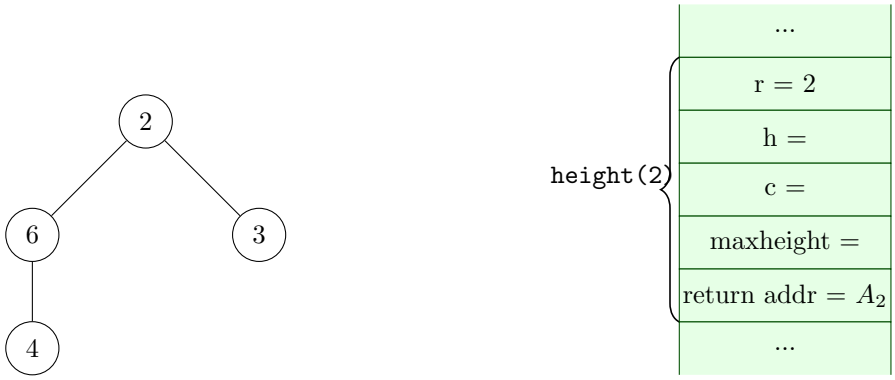
6.2 Space Complexity

The space complexity of an algorithm measures the amount of memory that the algorithm needs to execute correctly. An algorithm will require memory to store any data structures it uses, and if the algorithm is recursive it will also require memory to store the execution stack. To drive home this concept we will be working with the `height()` algorithm.

If we use the following main method:

```
main{
    b = createTree();
    h = height(B)       (A2)
}
```

Then when the first call is made, the activation record stores the value of the parameter r, and the local variables $height, c, h$ and the return address.



(a) Tree Diagram (b) Execution Stack

Figure 18: Current Position in the Algorithm

The algorithm will then start its execution and since r (2) is not a leaf, the value of max height will be set to 0, and it will start iterating through the children of r, meaning it will run the algorithm from node 6. A new activation record is created.

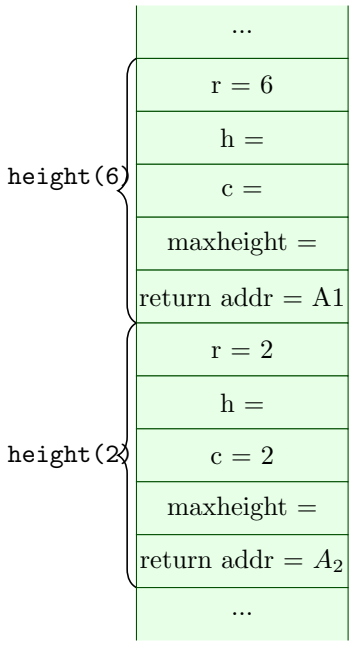


Figure 19: Execution Stack after first recursive call

Since 6 is not a leaf, node another activation record gets called onto the execution stack and since 4 is a leaf node 0 is returned to the call of $height(6)$ and then it $height(6)$ returns 1. Since 4 was the last child it returns back to the second c of r, which is 3. Since 3 is a leaf node it returns 0.

To determine the space complexity we have to observe in what scenario we had the most activation records. That was the case when we were at node 4. There were three recursive calls (one for 2, one for 6, and one for 4) hence we can deduce due every activation record using constant space c' the total space needed is c'(height of tree + 1) which is O(height of tree)

7 Week 7

7.1 Binary Search Trees

Ordered Dictionary (Map)

An ordered dictionary is an ADT that allows users to store a collection of records as key,value pairs, where the *keys* come from a total order.

Definition 2 (Total Order)**.** *Given two keys, k, k' we are* always *able to determine if they are less than, greater than, or equal to each other. Total order is defined for all $\mathbb{Z}$, $\mathbb{R}$, chars, Strings*

7.1.1 Search Tables

A search table is an ordered map implemented by a sorted sequence of keys. The performance is described below

- Searches - $O(\log n)$ – binary search
- Insert - $O(n)$ – Worst case we to shift $n-1$ items
- Removing an item - $O(n)$ – Worst case shift $n-1$ items to reconfigure order

7.1.2 Structure

A binary search tree is *proper* binary tree **storing records in its internal nodes** such that for each internal node u:

- Every key is the left subtree of u is smaller than $key(u)$
- Every key in the right subtree of u is smaller than $key(u)$

Leaves do not store records, they are not *null* either. They are empty instantiated nodes. Here is a visual

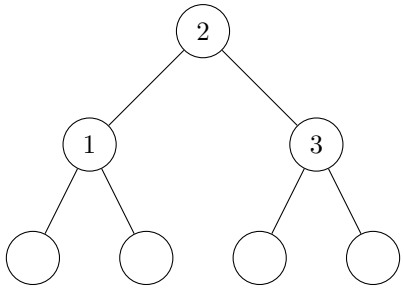


Figure 20: Example Tree

An inorder traversal results in the visiting the keys in order. Recall inorder traversal results in left subtree – root – right subtree. Therefore it will visit 1 – 2 – 3.

7.2 Ordered Tree Algorithm Analysis

For the purposes of discussing the algorithms below we are going to use this tree:

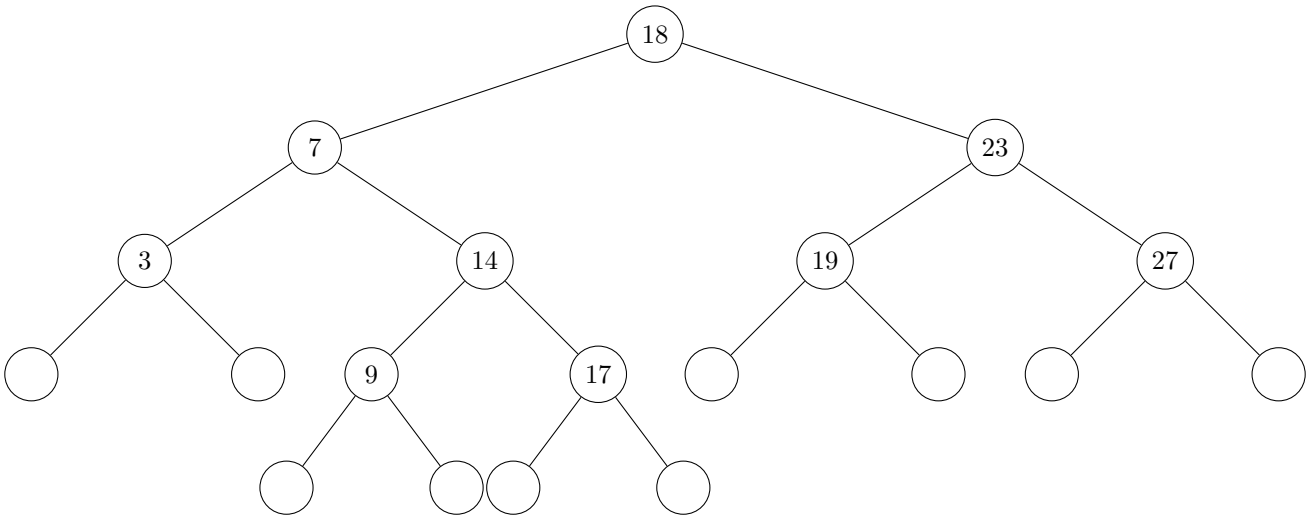


Figure 21: Example Tree

7.2.1 `get(k)` algorithm

Algorithm 11: get(r,k)

Input: Root r of a binary search tree, with key k
Output: Node storing k, or leaf where k should have been stored

```
1 if r is a leaf then
2 |  return r
3 else
4 |  if k = key store in r then
5 |  |  return r
6 |  else if k < key stored in r then
7 |  |  return get(left child of r, k)
8 |  else
9 |  |  return get(right child of r, k)
```

Algorithm Analysis
So the inputs of the algorithm is the root, as well as a the key k. Since it is established we have in our first call, the root node is not a leaf, so we fail the `if` block and move onto the `else` block. In this block we compare k to $r.key$. If it is not and the $k < r.key$ meaning it would be in the left side of the tree we traverse the left child of r looking for k recursively. Otherwise we traverse the right.

Time Complexity Analysis
We see no loops inside this algorithm, just constant time recursive calls we can say that c_1 operations are performed per recursive call. Now we need to determine the number of calls in the worst case scenario. Worst case scenario would be when the key k is not in the tree. Hence there will be at most height + 1 calls. Height of a tree is the maximum depth, which in the case of our diagram would be 4 + 1. So the function becomes:

$$f(n) = c_1(height + 1)$$

Since the only variable contingent on the size of the input is the height the time complexity is $O(height)$

Space Complexity Analysis
We need to determine the maximum number of recursive calls that this algorithm takes, we can deduce the space complexity. Since earlier we concluded that there are $height + 1$ recursive calls. The space complexity follows such: $O(height)$

7.2.2 `put(r,k,d)` algorithm

Algorithm 12: put(r,k,d)

Input: Root r of a binary search tree, with key, value pair (k, d)
Output: True if (k, d) was added to the tree, false otherwise

```
1 p ← get(r, k)
2 if p is an internal node then
3 |   return false
4 else
5 |   p.key ← k
6 |   p.data ← d
7 |   create two children of p
8 |   return true
```

Algorithm Analysis
So the input of the algorithm is the root of the tree r, and key,value pair (k, d). From line 1 we assign p to be the node where (k, d) *could be* inserted or if it is already in the tree. Hence if p is an internal node it implies that it is already inside the tree, and thereby we can immediately return false. In the case where p is the empty leaf node, we fit the key,value pair, and then add two more empty **NON-NULL** children to p and return true.

Time Complexity Analysis
The first line runs the `get(r,k)` algorithm, and we already deduced the time complexity of that algorithm to be $O(height)$ so the time complexity of that assignment will also be $O(height)$. The rest of the algorithm is performed by all constant time operations, and therefore the overall time complexity is also $O(height)$

Space Complexity Analysis
Again, it follows the space complexity from the `get(r,k)` algorithm as there are no additional data structures or activation records being created once after line 1 ends. Therefore $O(height)$ is the time complexity.

7.2.3 `smallest(r)` algorithm

Algorithm 13: smallest(r)

Input: Root r of a binary search tree
Output: Node storing the smallest key, or null if the tree has no data in it.

```
1 if r is a leaf then
2 |   return null
3 else
4 |   p ← r
5 |   while p is an internal node do
6 |   |   p ← left child of p
7 |   return parent of p
```

Algorithm Analysis
So the input of the algorithm is the root node and to find the smallest node after the inputted node. So right off the bat if the we input a *non-null* leaf then we return null, as there are no values that are smaller in the tree than the empty leaf. So we assign p the value of the input node, and use a while loop to traverse the left subtree of the given input node, until we arrive to a leaf node in the very left of the tree. In the case of our example it would be the left child of three. Since the node we are at right now is empty, we have to return the parent of p, which indeed is the smallest node:3.

Time Complexity Analysis
The `if` block are all constant time operations and therefore we will assign it some constant c_1. The `else` line 4 and 7 are both constant time operations assigned them c_2. Inside the while loop another constant time operation c_3. Now to determine the time complexity we have to determine how many time the while loop runs in the worst case scenario. In the worst case we give it the root node of the whole tree and it has to traverse the every left subtree. Therefore the total will be the numbers in the left most branch. Which has to be less than or equal to the height + 1 (maximum depth + 1). So we can conclude that the time complexity is

$$f(n) = c_1 + c_2 + c_3(height + 1)$$

which simplifies to $O(height)$

Space Complexity Analysis
Since no auxiliary data structures are created, and there are no recursive calls. This algorithm is $O(1)$ for space complexity.

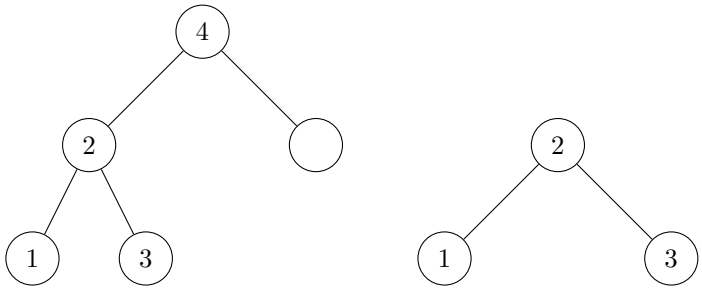
7.2.4 `remove(r,k)` algorithm

Algorithm 14: remove(r,k)

Input: Root r of a binary search tree, with key k
Output: True if node storing k is removed, false otherwise

```
1  p ← get(r,k)
2  if p is a leaf then
3  |  return false
4  else
5  |  if p has a child c that is a leaf then
6  |  |  p' ← parent of p
7  |  |  c' ← the other child of p
8  |  |  if p is the root then
9  |  |  |  make c' the new root
10 |  |  |  c'.parent ← null
11 |  |  else
12 |  |  |  Make c' the child of p' replacing p
13 |  else
14 |  |  s ← smallest(rightchildofp)
15 |  |  copy key data from s to p
16 |  |  remove(s, key in s)
17 |  return true
```

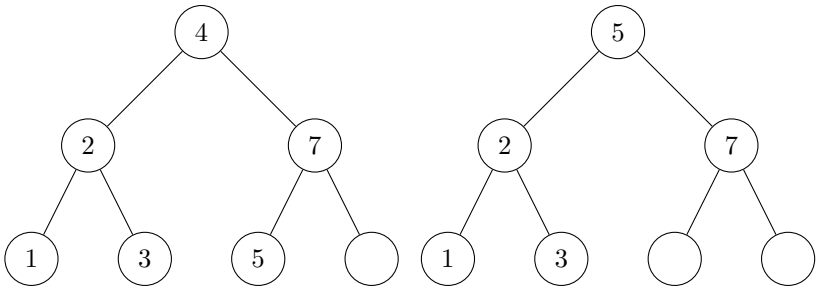
Algorithm Analysis
So the first line locates where in the tree, the given key is and furthermore tests if it is a leaf node then it would return false as leaf nodes are empty and there is nothing to remove. So then we move into the `else` bock and then we test for the condition that if the current node with key k has a child c that is a leaf, implying that the node p is in the last layer of the tree, then it makes p' the parent of p, and assigns c' to the *empty* child of p. Now here we consider an interesting case. What do we do if p is the root node and the tree looks as follows:



(a) Tree before removal of (4) (b) Tree after removal of (4)

Figure 22: `remove(4,k)`

So, just running through the algorithm we arrive at the following values: $p' =$ `null`, $c' = 2, p = 4$. So since we trigger the if p is root then we have to make c' the new root, and set $c'parent \leftarrow null$. In this case we do not hit the `else` block with the recursive call. So lets illustrate that case.



(a) Tree before removal of (4) (b) Tree after removal of (4)

Figure 23: `remove(4,k)`

In this case fail the `if` block for p having a child that is a leaf as (4) does not have any leaves. Hence we skip to $s \leftarrow$ `smallest(rightchildofp)` So what this does is run the `smallest` algorithm discussed above, which will traverse down the subtree of the right child of p which in this case is 7 being the root, and 5 being the return. So s will hold a value of 5. Then the data from s replaces the data in p, and s will now be removed, by re-entering the algorithm with s as the parameter it will return false, and then we will exit `else` block and then return true overall.

Time Complexity Analysis
Line 1 has a time complexity of $O(n)$ as discussed above, lines 2,3 and 17 all have constant time operations which we can denote as c_1, inside the first `else` block we see all constant time operations which we denote as c_2. Going to the second `else` block we observe a recursive call, as well as one function all to `smallest`. These operations are both $O(height)$ and that leaves us with one constant time operation in the middle c_3. The final time complexity will be $O(height)$

Space Complexity Analysis
Since there are no auxiliary data structures being created, as well as no recursive calls within recursive calls. Each recursive algorithm runs until it returns an answer, meaning the space complexity will be $O(height)$

7.2.5 `successor (r,k)` algorithm

The purpose of this algorithm is to find the smallest key greater than k.

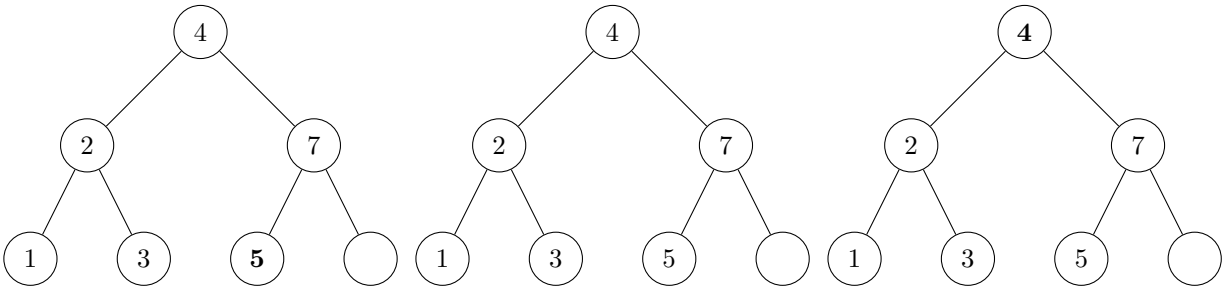
Algorithm 15: successor(r,k)

Input: Root r of a binary search tree, with key k
Output: Node storing the successor of k or null if k has no successor

```
1  if r is a leaf then
2      return null
3  else
4      p ← get(r, k)
5      if p is an internal node and right child of p is an internal node then
6          return smallest(right child of p)
7      else
8          p′ ←parent of p
9          while (p ≠ r) and p is the right child of p′ do
10             p ← p′
11             p′ ← parent of p
12         if p = r then
13             return null
14         else
15             return p'
```

Algorithm Analysis
So this algorithm starts out by giving us the root node r, with key k. Right off the bat if r is a leaf node then it will have no successors so it returns null. Once we skip that `if` block, the value of p becomes the node location

of p. If the node p is an internal node and the right child of p is also not a leaf node then we can just return the `smallest(right child of p)`. This is visualized below:



(a) Successor of (4) (b) Successor of (7) (c) Successor of (3)

Figure 24: `successor(4/7,k)`

We can see that the (a) the successor of 4 will have to return 5, as the `smallest` algorithm is called on 7 (right child of 4), and that will return 5. Now in the secondary case, where the `else` block gets called, $p'(4)$ will become the parent of $p(7)$, and then we enter a while loop where $7 \neq 4$(root node r) and $p(7)$ is the right child of $p'(4)$ the algorithm will then reassign p to p' so now $p = 4$, and $p' = null$(parent of 4). But this breaks the while loop as $p(4) = r(4)$, and since this equality holds it will return null as predicted because there is no successor to 7. Now in the case that $p \neq r$ we meet the situation where p is no longer the right child of p' . We can examine that situation by running the algorithm on (3). In the case where we end up at the `while` block again, $p(3) \leftarrow p'(2)$ and $p'(2) \leftarrow$ parent of $p(4)$. But this while loop breaks here as $p(2)$ is no longer the right child of $p'(4)$, and since $2 \neq 4$ we will return $p'(4)$ which is the logical successor of 3.

Time Complexity Analysis
Lines 1,2,5,8,12-15 are all constant time so we can denote that as c_1, now line 4, calls an algorithm that runs at $O(height)$, and so does line 6. What is left now is for us to deduce how many iterations of the while loop occur. The maximum number of times this loop would have to run would be height, as it could be the right most node on the left subtree from the root and this while loop would have to traverse all the way back up to the root to return the successor, therefore we can assign it a time complexity of $O(height)$ and inside 2 c_2 operations are performed. Therefore the overall time complexity of this algorithm is $O(height)$.

Space Complexity Analysis
Since there are no auxiliary data structures being created, as well as no recursive calls within recursive calls. Each recursive algorithm runs until it returns an answer, meaning the space complexity will be $O(height)$