Introduction to Algorithms

Ian Poon

September 2024

Contents

1 Introduction: Asymptotics Review

Fact 1

Asymptotic Notation

- upperbounds (O), lower bounds (Ω),tight bounds (Θ)

1.1 Exercises

Lemma 2

Every function that is in $O(\log(\log n))$ is also in $O \log n$ but not vice versa

$$O(\log \log n) \subset O(\log n)$$

Proof. Let $f(n) = O(\log(\log n))$. By definition Big O there exists $c, n_0 > 0$

$$f(n) \leq c \cdot \log(\log(n))$$

for all $n \geq n_0$. However we know that for sufficiently large n, $\log(\log n) \leq \log n$(think about it graphically) hence there exists $c_1 \geq c$ where

$$f(n) \leq c \cdot \log(\log(n)) \leq c_1 \cdot \log n$$

This shows that $f(n) = O(\log n)$. However you cleary can't do the same for the converse again because $\log(\log n)$ grows much more slowly than $\log n$

Lemma 3

$$m^m = \Omega(2^m)$$

Proof. Essentially we want to show that $2^m = O(m^m)$ which is to say there exists c, m_0 such that for all $m \geq m_0$,

$$m^m \leq c \cdot 2^m$$

Notice that there certainly exists $m \geq m_0$ where

$$\left(\frac{m}{2}\right)^m \geq c$$

Hence multiplying both sides by 2^m yields the lemma

Example 4

Find a simple, *tight* asymptotitc bound for $\log_{6006}\left(\left(\log(n^{\sqrt{n}})\right)^2\right)$

Remark 5. *Tight upper bound means the lowest upper bound/suprenum. That is there is no number smaller than the tight upper bound that is an upperbound.*

Solution. Consider

$$\begin{aligned}\log_{6006}\left(\left(\log(n^{\sqrt{n}})\right)^2\right) &= \frac{2}{\log 6006} \log\left(\sqrt{n} \log n\right) \\ &= \Theta(\log n^{\frac{1}{2}} + \log\log n) = \Theta(\log n)\end{aligned}$$

To see why the last line follows consider

$$f(n) = \frac{2}{\log 6006}(\frac{1}{2} \log n + \log\log n)$$

by definition we can ignore the constant term and bound the above by $\Theta(\log n^{\frac{1}{2}} + \log\log n)$ but this is not the simplest form. Consider that if we can show there exists c_1, c_2, n_0

$$c_1 \log n \leq \frac{1}{2}\log n + \log(\log n) \leq \log n \leq c_2 \log n$$

for all $n \geq n_0$ then we get the above relations. For the upper bound this is certainly possible because $\log\log n$ grows much slower than $\log n$ so we have the 2nd inequality for large enough n. Then by archimedian principle c_2 exists. For the lower bound, for large n otice that the $\frac{1}{2}\log n$ term ensures that $\frac{1}{2}\log n + (\log\log n)$ cannot be smaller than a constant multiple of $\log n$. That is we can let $c_1 = \frac{1}{2}$.

Example 6
Show that $(\log n)^{\log n} = \Omega(n)$

Solution. Note that $m^m = \Omega(2^m)$. Then setting $n = 2^m$ completes the proof

Example 7
Show that $(6)! \notin \Theta(n!)$ but $\log((6n)!) \in \Theta(\log(n!))$

Solution. recall that the sterling formula gives

$$\log(n!) = \Theta(n\log n)$$

substituting $6n$ we have

$$\log((6n)!) = \Theta(6n(\log n + \log 6)) = \Theta(6n(\log n) + 6n\log 6))$$

however notice that there certainly exists(same kind of logic used previously recall) for large enough n

$$c_1 \cdot n\log n \leq (n\log n + n\log 6) \leq 2n\log n \leq c_2 \cdot n\log n$$

because $n\log n$ certainly grows faster than $n \cdot 1$. So we have $\log((6n)!) \in \Theta(\log(n!))$ as desired.

now substituting $n = 6$ we have

$$\log(6!) = \Theta(6\log 6) = \Theta(1)$$

so clearly $(6)! \notin \Theta(n!)$

2 Data Structures

Definition 8
A **data structure** is a way to store data with algorithms that support **operations** on the data. Interface is **specification**: what operations are supported(the problem) while data structure is a **representation**: how operations are supported(the solution).

In this class there two main interfaces: **sequence** and **set**. We will give some examples of implementations for both which we will cover until the chapter on `hashing`. In fact there is still more and with even better performance as you will see in the chapter of `trees` and beyond.

Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n

Figure 1: Sequence Implementations

Container	`build(X)`	given an iterable `X`, build sequence from items in `X`
	`len()`	return the number of stored items
Static	`iter_seq()`	return the stored items one-by-one in sequence order
	`get_at(i)`	return the i^{th} item
	`set_at(i, x)`	replace the i^{th} item with x
Dynamic	`insert_at(i, x)`	add x as the i^{th} item
	`delete_at(i)`	remove and return the i^{th} item
	`insert_first(x)`	add x as the first item
	`delete_first()`	remove and return the first item
	`insert_last(x)`	add x as the last item
	`delete_last()`	remove and return the last item

Figure 2: Sequence Interface

Sequences maintain a collection of items in an **extrinsic** order where each item stored has a **rank** in the sequence. By extrinsic we mean that the item is positioned not because of what the item is but because of some external party put it there.

Remark 9. *However there are 2 special case sequence interfaces namely* **stack** *and* **queue** *whom carry out dynamic operations in a unique manner. In particular stack follows "LIFO" while queue follows "FIFO" type of insertion and deletions. We will learn more later*

Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access Array	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n

Figure 3: Set Implementations

Container	`build(X)`	given an iterable `X`, build sequence from items in `X`
	`len()`	return the number of stored items
Static	`find(k)`	return the stored item with key `k`
Dynamic	`insert(x)`	add `x` to set (replace item with key `x.key` if one already exists)
	`delete(k)`	remove and return the stored item with key `k`
Order	`iter_ord()`	return the stored items one-by-one in key order
	`find_min()`	return the stored item with smallest key
	`find_max()`	return the stored item with largest key
	`find_next(k)`	return the stored item with smallest key larger than `k`
	`find_prev(k)`	return the stored item with largest key smaller than `k`

Figure 4: Set Interface

In contrast, sets is about **intrinsic** order instead. We maintain a set of items having **unique keys**. Notice there is an extra category called "order" in the set interface compared to that of sequences.

Remark 10. *However there are special case interfaes such as* **dictionary** *which is a set without order operations.We will learn more later.*

2.1 Array Sequence

```python
class Array_Seq:
    def __init__(self): # O(1)
        self.A = []
        self.size = 0
    def __len__(self): return self.size # O(1)
    def __iter__(self): yield from self.A # O(n) iter_seq
    def build(self, X): # O(n)
        self.A = [a for a in X] # pretend this builds a static array
        self.size = len(self.A)

    def get_at(self, i): return self.A[i] # O(1)
    def set_at(self, i, x): self.A[i] = x # O(1)

    def _copy_forward(self, i, n, A, j): # O(n)
        for k in range(n):
            A[j + k] = self.A[i + k]

    def _copy_backward(self, i, n, A, j): # O(n)
        for k in range(n - 1, -1, -1):
            A[j + k] = self.A[i + k]

    def insert_at(self, i, x): # O(n)
        n = len(self)
        A = [None] * (n + 1)
```

```
        self._copy_forward(0, i, A, 0)
        A[i] = x
        self._copy_forward(i, n - i, A, i + 1)
        self.build(A)

    def delete_at(self, i): # O(n)
        n = len(self)
        A = [None] * (n - 1)
        self._copy_forward(0, i, A, 0)
        x = self.A[i]
        self._copy_forward(i + 1, n - i - 1, A, i)
        self.build(A)
        return x
     # O(n)
    def insert_first(self, x): self.insert_at(0, x)
    def delete_first(self): return self.delete_at(0)
    def insert_last(self, x): self.insert_at(len(self), x)
    def delete_last(self): return self.delete_at(len(self) - 1)
```

Listing 1: Array Sequence Full Implementation

2.2 Linked List Sequence

```
class Linked_List_Node:
    def __init__(self, x): # O(1)
        self.item = x
        self.next = None
    def later_node(self, i): # O(i)
        if i == 0: return self
        assert self.next
        return self.next.later_node(i - 1)
```

Listing 2: Linked List Sequence Node Implementation

```
class Linked_List_Seq:
    def __init__(self):
        self.head = None
        self.size = 0
    def __len__(self): return self.size
    def __iter__(self):
        node = self.head
```

```
        while node:
            yield node.item
            node = node.next
    def build(self, X):
        for a in reversed(X):
            self.insert_first(a)
    def get_at(self, i):
        node = self.head.later_node(i)
        return node.item
    def set_at(self, i, x): # O(i)
        node = self.head.later_node(i)
        node.item = x
    def insert_first(self, x): # O(1)
        new_node = Linked_List_Node(x)
        new_node.next = self.head
        self.head = new_node
        self.size += 1
    def delete_first(self): # O(1)
        x = self.head.item
        self.head = self.head.next
        self.size -= 1
        return x
    def insert_at(self, i, x): # O(i)
        if i == 0:
            self.insert_first(x)
            return
        new_node = Linked_List_Node(x)
        node = self.head.later_node(i - 1)
        new_node.next = node.next
        node.next = new_node
        self.size += 1
    def delete_at(self, i): # O(i)
        if i == 0:
            return self.delete_first()
        node = self.head.later_node(i - 1)
        x = node.next.item
        node.next = node.next.next
        self.size -= 1
        return x
        # O(n)
    def insert_last(self, x): self.insert_at(len(self), x)
    def delete_last(self): return self.delete_at(len(self) - 1)
```

Listing 3: Linked List Sequence Full Implementation

2.3 Dynamic Array Sequence

```
class Dynamic_Array_Seq(Array_Seq):
    def __init__(self, r = 2): # O(1)
        super().__init__()
        self.size = 0
        self.r = r
        self._compute_bounds()
        self._resize(0)
    def __len__(self): return self.size # O(1)
    def __iter__(self): # O(n)
        for i in range(len(self)): yield self.A[i]
    def build(self, X): # O(n)
        for a in X: self.insert_last(a)
    def _compute_bounds(self): # O(1)
        self.upper = len(self.A)
        self.lower = len(self.A) // (self.r * self.r)
    def _resize(self, n): # O(1) or O(n)
        if (self.lower < n < self.upper): return
        m = max(n, 1) * self.r
        A = [None] * m
        self._copy_forward(0, self.size, A, 0)
        self.A = A
        self._compute_bounds()
    def insert_last(self, x): # O(1)a
        self._resize(self.size + 1)
        self.A[self.size] = x
        self.size += 1
    def delete_last(self): # O(1)a
        self.A[self.size - 1] = None
        self.size -= 1
        self._resize(self.size)
    def insert_at(self, i, x): # O(n)
        self.insert_last(None)
        self._copy_backward(i, self.size - (i + 1), self.A, i + 1)
        self.A[i] = x
    def delete_at(self, i): # O(n)
        x = self.A[i]
        self._copy_forward(i + 1, self.size - (i + 1), self.A, i)
        self.delete_last()
```

```
        return x
    # O(n)
    def insert_first(self, x): self.insert_at(0, x)
    def delete_first(self): return self.delete_at(0)
```

Listing 4: Dynamic Array Sequence Full Implementation

3 Sorting

3.1 Sorted Array Set

```
class Sorted_Array_Set:
    def __init__(self): self.A = Array_Seq() # O(1)
    def __len__(self): return len(self.A) # O(1)
    def __iter__(self): yield from self.A # O(n)
    def iter_order(self): yield from self # O(n)
    def build(self, X): # O(?)
        self.A.build(X)
        self._sort()

    def _sort(self): # O(?)
        #sorting algo
        return 0

    def _binary_search(self, k, i, j): # O(log n)
        if i >= j: return i
        m = (i + j) // 2
        x = self.A.get_at(m)
        if x.key > k: return self._binary_search(k, i, m - 1)
        if x.key < k: return self._binary_search(k, m + 1, j)
        return m
    def find_min(self): # O(1)
        if len(self) > 0: return self.A.get_at(0)
        else: return None
    def find_max(self): # O(1)
        if len(self) > 0: return self.A.get_at(len(self) - 1)
        else: return None
    def find(self, k): # O(log n)
        if len(self) == 0: return None
        i = self._binary_search(k, 0, len(self) - 1)
        x = self.A.get_at(i)
```

```
        if x.key == k: return x
        else: return None
    def find_next(self, k): # O(log n)
        if len(self) == 0: return None
        i = self._binary_search(k, 0, len(self) - 1)
        x = self.A.get_at(i)
        if x.key > k: return x
        if i + 1 < len(self): return self.A.get_at(i + 1)
        else: return None
    def find_prev(self, k): # O(log n)
        if len(self) == 0: return None
        i = self._binary_search(k, 0, len(self) - 1)
        x = self.A.get_at(i)
        if x.key < k: return x
        if i > 0: return self.A.get_at(i - 1)
        else: return None
    def insert(self, x): # O(n)
        if len(self.A) == 0:
            self.A.insert_first(x)
        else:
            i = self._binary_search(x.key, 0, len(self.A) - 1)
            k = self.A.get_at(i).key
            if k == x.key:
                self.A.set_at(i, x)
                return False
            if k > x.key: self.A.insert_at(i, x)
            else: self.A.insert_at(i + 1, x)
        return True
    def delete(self, k): # O(n)
            i = self._binary_search(k, 0, len(self.A) - 1)
            assert self.A.get_at(i).key == k
            return self.A.delete_at(i)
```

Listing 5: Sorted Array Set

3.2 Selection Sort

```
def selection_sort(A): # Selection sort array A
    for i in range(len(A) - 1, 0, -1): # O(n) loop over array
        m = i # O(1) initial index of max
        for j in range(i): # O(i) search for max in A[:i]
            if A[m] < A[j]: # O(1) check for larger value
                m = j # O(1) new max found
        A[m], A[i] = A[i], A[m] # O(1) swap
```

Listing 6: Selection Sort

3.3 Insertion Sort

```
def insertion_sort(A): # Insertion sort array A
    for i in range(1, len(A)): # O(n) loop over array
        j = i # O(1) initialize pointer
        while j > 0 and A[j] < A[j - 1]: # O(i) loop over prefix
            A[j - 1], A[j] = A[j], A[j - 1] # O(1) swap
            j = j - 1 # O(1) decrement j
```

Listing 7: Insertion Sort

3.4 Mergesort

```python
def merge_sort(A, a = 0, b = None): # Sort sub-array A[a:b]
    if b is None: # O(1) initialize
        b = len(A) # O(1)
    if 1 < b - a: # O(1) size k = b - a
        c = (a + b + 1) // 2 # O(1) compute center
        merge_sort(A, a, c) # T(k/2) recursively sort left
        merge_sort(A, c, b) # T(k/2) recursively sort right
        L, R = A[a:c], A[c:b] # O(k) copy
        i, j = 0, 0 # O(1) initialize pointers
        while a < b: # O(n)
            if (j >= len(R)) or (i < len(L) and L[i] < R[j]): # O(1) check side
                A[a] = L[i] # O(1) merge from left
                i = i + 1 # O(1) decrement left pointer
            else:
                A[a] = R[j] # O(1) merge from right
                j = j + 1 # O(1) decrement right pointer
            a = a + 1 # O(1) decrement merge pointer
```

Listing 8: Merge Sort

Notice the recurrence relation for merge sort is

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

Comparing it with the form of the Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + g(n)$$

we have $a = 2$, $b = 2$, and $g(n) = \Theta(n)$. First, we calculate $\log_b(a)$:

$$\log_b(a) = \log_2(2) = 1$$

We now compare the function $g(n) = \Theta(n)$ with $n^{\log_b(a)} = n^1 = n$. Hence we have $g(n) = \Theta(n)$ $n^{\log_b(a)} = n$ Thus, $g(n) = \Theta(n^{\log_b(a)})$. This fits Case 2 of the Master Theorem, which applies when $g(n) = \Theta(n^{\log_b(a)} \log^k n)$. In this case, $k = 0$, so the recurrence simplifies to:

$$T(n) = \Theta(n^{\log_b(a)} \log^{k+1} n) = \Theta(n \log n)$$

We will see more examples below.

3.5 Recurrence

There are three primary methods for solving recurrences:

- **substitution**: Guess a solution and substitute to show the recurrence holds
- **Recursion tree**: draw a tree representing the recurrence and sum computation at nodes
- **Master Theorem**: A generral formula to solve a large class of recurrrences.

Theorem 11 (Master Theorem)
Let T be a recurrence of the form

$$T(n) = aT(\frac{n}{b}) + g(n)$$

Case 1: If $g(n) = O(n^{\log_b(a)-\varepsilon})$ for some constant $\varepsilon > 0$ then

$$T(n) = \Theta(n^{\log_b(a)})$$

Case 2: If $g(n) = \Theta(n^{\log_b(a)} \log^k(n))$ for some constant $k \geq 0$ then

$$T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$$

Case 3: If $g(n) = \Omega(n^{\log_b(a)+\varepsilon})$ for some constant $\varepsilon > 0$ and $ag(n/b) < cg(n)$ then

$$T(n) = \Theta(g(n))$$

Proof. The master theorem is a special case of **Akra-Bazzi** formula. This will be covered in `6.046 Design and Analysis of Algorithms` the sequel to this course. So stay tuned...

3.6 Exercises

Solve the following recurrences given $T(1) = \Theta(1) = O(n)$(recall the last equality follows by definition of big theta. However the converse implication may not be true refer back to your 6.041 Math for CS notes for more)

Example 12

$$T(n) = T(\frac{n}{2}) + O(1)$$

Solution. Comparing with the form

$$T(n) = aT(\frac{n}{b}) + g(n)$$

we see that $a = 1, b = 2, g(n) = O(1)$. Now since

$$n^{\log_b(a)} = n^0 = 1$$

Hence, there exists $k \geq 0$ namely $k = 0$ such that

$$g(n) = O(1) = \Theta(n^{\log_b(a)} \log^k(n))$$

Therefore by case 2 of the master theorem we have $T(n) = O(\log n)$

Example 13

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

Solution. By direct expansion we have

$$T(n) = T(1) + O(1) + \ldots + O(1)$$

So clearly $T(n) = O(n)$

Example 14

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

Solution. By direct expansion we have

$$T(n) = \sum_{i=1}^{n} T(i)$$

which is just the arithmetic sum so

$$T(n) = O(\frac{n(n+1)}{2})$$

recall polynomial complexity is just the highest degree this simplifies to

$$T(n) = O(n^2)$$

Example 15

$$T(n) = 2T(n-1) + O(1)$$

Solution. By direct expansion we have

$$T(n) = 2^n T(1) + (2^{n-1} + 2^{n-2} + \ldots + 2 + 1)O(1)$$

and again this polynomial reduces to

$$T(n) = O(2^n)$$

Example 16

$$T(n) = T(2n/3) + O(1)$$

Solution. Again case 2 if master theorem so $T(n) = O(\log n)$

Example 17

$$T(n) = 2T(n/2) + O(1)$$

Proof. This recurrence is in the form $T(n) = aT\left(\frac{n}{b}\right) + g(n)$ where:

$$a = 2, \quad b = 2, \quad g(n) = O(1)$$

First, we compute $\log_b(a)$:

$$\log_b(a) = \log_2(2) = 1$$

Now, we compare $g(n) = O(1)$ with $n^{\log_b(a)} = n^1 = n$. Since $g(n) = O(1)$ is smaller than n, we are in Case 1 of the Master Theorem. Thus, the solution is:

$$T(n) = \Theta(n^{\log_b(a)}) = \Theta(n)$$

Hence, the time complexity is $T(n) = O(n)$. □

—

Example 18

$$T(n) = T(n/2) + O(n)$$

Proof. This recurrence is in the form $T(n) = aT\left(\frac{n}{b}\right) + g(n)$ where:

$$a = 1, \quad b = 2, \quad g(n) = O(n)$$

First, we compute $\log_b(a)$:

$$\log_b(a) = \log_2(1) = 0$$

Now, we compare $g(n) = O(n)$ with $n^{\log_b(a)} = n^0 = 1$. Since $g(n) = O(n)$ is larger than $n^0 = 1$, we are in Case 3 of the Master Theorem. Therefore, the solution is:

$$T(n) = \Theta(g(n)) = \Theta(n)$$

Hence, the time complexity is $T(n) = O(n)$. □

—

Example 19

$$T(n) = 2T(n/2) + O(n \log n)$$

Proof. This recurrence is in the form $T(n) = aT\left(\frac{n}{b}\right) + g(n)$ where:

$$a = 2, \quad b = 2, \quad g(n) = O(n \log n)$$

First, we compute $\log_b(a)$:

$$\log_b(a) = \log_2(2) = 1$$

Now, we compare $g(n) = O(n \log n)$ with $n^{\log_b(a)} = n^1 = n$. Since $g(n) = \Theta(n \log n)$, which is of the form $n^{\log_b(a)} \log^k(n)$ with $k = 1$, we are in Case 2 of the Master Theorem. Therefore, the solution is:

$$T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n)) = \Theta(n \log^2 n)$$

Hence, the time complexity is $T(n) = O(n \log^2 n)$. □

—

Example 20

$$T(n) = 4T(n/2) + O(n)$$

Proof. This recurrence is in the form $T(n) = aT\left(\frac{n}{b}\right) + g(n)$ where:

$$a = 4, \quad b = 2, \quad g(n) = O(n)$$

First, we compute $\log_b(a)$:

$$\log_b(a) = \log_2(4) = 2$$

Now, we compare $g(n) = O(n)$ with $n^{\log_b(a)} = n^2$. Since $g(n) = O(n)$ is smaller than n^2, we are in Case 1 of the Master Theorem. Therefore, the solution is:

$$T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^2)$$

Hence, the time complexity is $T(n) = O(n^2)$. □

4 Hashing

We have so far considered 2 set implementations and so far search and dynamic operations have at best $\Theta(\log n))$ time. Are there data structures that could do these faster? Yes! As you should have seen in the implementation diagrams in the data structures section above - the answer is **Direct Access Arrays** and **Hash Tables**

4.1 Direct Access Arrays

```
class DirectAccessArray:
    def __init__(self, u): self.A = [None] * u # O(u)
    def find(self, k): return self.A[k] # O(1)
    def insert(self, x): self.A[x.key] = x # O(1)
    def delete(self, k): self.A[k] = None # O(1)
    def find_next(self, k):
        for i in range(k, len(self.A)): # O(u)
            if A[i] is not None:
                return A[i]
    def find_max(self):
        for i in range(len(self.A) - 1, -1, -1): # O(u)
            if A[i] is not None:
                return A[i]
    def delete_max(self):
        for i in range(len(self.A) - 1, -1, -1): # O(u)
            x = A[i]
            if x is not None:
                A[i] = None
                return x
```

Listing 9: Direct Access Array

Notice that the index already captures the order. That's why to find the max all we are doing is just reverse traversing starting from the biggest index and finding the first non-empty position which will the biggest element!

Well like you have learned in high school the drawbacks is collisions and memory wastage. To solve this we use hash tables

4.2 Hash Tables

```
class Hash_Table_Set:
    def __init__(self, r = 200): # O(1)
        self.chain_set = Set_from_Seq(Linked_List_Seq)
        self.A = []
        self.size = 0
        self.r = r # 100/self.r = fill ratio
        self.p = 2**31 - 1
        self.a = randint(1, self.p - 1)
        self._compute_bounds()
        self._resize(0)
    def __len__(self): return self.size # O(1)
```

```
12      def __iter__(self): # O(n)
13          for X in self.A:
14              yield from X
15      def build(self, X): # O(n)e
16          for x in X: self.insert(x)
17      def _hash(self, k, m): # O(1)
18          return ((self.a * k) % self.p) % m
19      def _compute_bounds(self): # O(1)
20          self.upper = len(self.A)
21          self.lower = len(self.A) * 100*100 // (self.r*self.r)
22      def _resize(self, n): # O(n)
23          if (self.lower >= n) or (n >= self.upper):
24              f = self.r // 100
25              if self.r % 100: f += 1
26              # f = ceil(r / 100)
27              m = max(n, 1) * f
28              A = [self.chain_set() for _ in range(m)]
29              for x in self:
30                  h = self._hash(x.key, m)
31                  A[h].insert(x)
32              self.A = A
33              self._compute_bounds()
34      def find(self, k): # O(1)e
35          h = self._hash(k, len(self.A))
36          return self.A[h].find(k)
37      def insert(self, x): # O(1)ae
38          self._resize(self.size + 1)
39          h = self._hash(x.key, len(self.A))
40          added = self.A[h].insert(x)
41          if added: self.size += 1
42          return added
43      def delete(self, k): # O(1)ae
44          assert len(self) > 0
45          h = self._hash(k, len(self.A))
46          x = self.A[h].delete(k)
47          self.size -= 1
48          self._resize(self.size)
49          return x
50      def find_min(self): # O(n)
51          out = None
52          for x in self:
53              if (out is None) or (x.key < out.key):
54                  out = x
```

```
        return out
    def find_max(self): # O(n)
        out = None
        for x in self:
            if (out is None) or (x.key > out.key):
                out = x
        return out
    def find_next(self, k): # O(n)
        out = None
        for x in self:
            if x.key > k:
                if (out is None) or (x.key < out.key):
                    out = x
        return out
    def find_prev(self, k): # O(n)
        out = None
        for x in self:
            if x.key < k:
                if (out is None) or (x.key > out.key):
                    out = x
        return out
    def iter_order(self): # O(n^2)
        x = self.find_min()
        while x:
            yield x
            x = self.find_next(x.key)
```

Listing 10: Hash Table Set

I know basic methods to prevent collisions and stuff but i probably will want to wait for `6.046 Design and Analysis` to learn more details about good hash functions...

5 Linear Sorting

```
def direct_access_sort(A):
    #"Sort A assuming items have distinct non-negative keys"
    u = 1 + max([x.key for x in A]) # O(n) find maximum key
    D = [None] * u # O(u) direct access array
    for x in A: # O(n) insert items
        D[x.key] = x
    i = 0
```

```
    for key in range(u): # O(u) read out items in order
        if D[key] is not None:
            A[i] = D[key]
            i += 1
```

Listing 11: Direct Access Sort

Essentially you transfer elements from an array to a dictionary with unique keys representative of an order. Doing so its only $O(n)$ because you just need to one pass array A. Now you one pass the dictionary in order of its keys which should give you elements of A in sorted order which is again $O(n)$. You literally did that in ICL year 1 practical haha. Wow looks like we are even faster than $\Omega(n \log n)$ sorting using the comparison model(that is binary comparisons is element $a > b$? then carry out appropriate operations like your typical mergesort,bubblesort etc). Unfortunately it has two drawbacks(exactly analagous with our situation with direct access arrays)

1. It cannot handle duplicate keys
2. it cannot large key ranges

And just like previously we use something similar to hashing, also involving chains - counting sort

```
def counting_sort(A):
#"Sort A assuming items have non-negative keys"
    u = 1 + max([x.key for x in A])
    D = [[] for i in range(u)]
    for x in A:
        D[x.key].append(x)
    i = 0
    # O(n) find maximum key
    # O(u) direct access array of chains
    # O(n) insert into chain at x.key
    for chain in D:
        for x in chain:
            # O(u) read out items in order
            A[i] = x
            i+= 1
```

Listing 12: counting sort

6 Binary Trees

6.1 Basics

Recall in `data structures` section we have foreshadowed a certain set and data structure that can achieve better performance that what was listed. The answer lies in - **binary trees**

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n
Goal	n	$\log n$	$\log n$	$\log n$	$\log n$

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access Array	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n
Goal	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

Figure 5: Binary Trees Implementation

With binary trees these are the desired performances we want to achieve in both set and sequence implementations(as represented in the "goal" row). See how it compares with our existing known data structures.

First we review some terminology

Definition 21

Consider the following terms:

- The **root** of a tree has no parent
- A **leaf** of a tree has no chldren
- Define **depth** of a node $< x >$ in a tree rooted at $< R >$ to be the lenght of path from $< x >$ to $< R >$
- Define **height** of a node $< x >$ to be the maximum depth of any node in the **subtree** rooted at $< x >$

Definition 22

For now we define a binary tree's **traversal order** based on the following implicit characterization

- every node in the left subtree of node $< x >$ comes *before* $< x >$ in the traversal order
- every node in the right subtree of node $< x >$ comes *after* $< x >$ in the traversal order

Next time we will provide two different semantic meanings to the traversal order(which will lead to an efficient implementation of the sequence and set interface respectively). For now we just want to investigate how to preserve traversal order as we manipulate a tree.

```
1 class Binary_Node:
2     def __init__(A, x): # O(1)
3         A.item = x
```

```
        A.left = None
        A.right = None
        A.parent = None
        # A.subtree_update() # wait for R07!
    def subtree_iter(A): # O(n)
        if A.left: yield from A.left.subtree_iter()
        yield A
        if A.right: yield from A.right.subtree_iter()
    def subtree_first(A): # O(h)
        if A.left: return A.left.subtree_first()
        else: return A
    def subtree_last(A): # O(h)
        if A.right: return A.right.subtree_last()
        else: return A
    def successor(A): # O(h)
        if A.right: return A.right.subtree_first()
        while A.parent and (A is A.parent.right):
            A = A.parent
        return A.parent
    def predecessor(A): # O(h)
        if A.left: return A.left.subtree_last()
        while A.parent and (A is A.parent.left):
            A = A.parent
        return A.parent
    def subtree_insert_before(A, B): # O(h)
        if A.left:
            A = A.left.subtree_last()
            A.right, B.parent = B, A
        else:
            A.left, B.parent = B, A
        # A.maintain() # wait for R07!
    def subtree_insert_after(A, B): # O(h)
        if A.right:
            A = A.right.subtree_first()
            A.left, B.parent = B, A
        else:
            A.right, B.parent = B, A
        # A.maintain() # wait for R07!
    def subtree_delete(A): # O(h)
        if A.left or A.right:
            if A.left: B = A.predecessor()
            else: B = A.successor()
            A.item, B.item = B.item, A.item
```

```
            return B.subtree_delete()
        if A.parent:
            if A.parent.left is A: A.parent.left = None
            else: A.parent.right = None
        # A.parent.maintain() # wait for R07!
        return A
```

Listing 13: Binary Node

The idea of the code is not anything new from what you have learned from high school. However pay attention to the time complexity $O(h)$.Notice h refers to the *height* of the binary tree.

Problem 23

Prove that the smallest height for any tree on n nodes is $\lceil \lg(n+1) \rceil - 1 = \Omega(\log n)$

Proof. A complete binary tree is one in which every level of the tree is fully filled, except possibly the last level, which is filled from left to right. The height h of a complete binary tree with n nodes is minimized, so it represents the smallest possible height for a tree of n nodes.(because you always fully fill levels first before going to the next)

A complete binary tree with height h has at most $2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$ nodes. Thus, a tree with height h can have at most $2^{h+1} - 1$ nodes.

Let n be the total number of nodes in the tree. We want to find the minimum height h such that the number of nodes n is at least as large as the number of nodes in a complete binary tree with height h. Since a tree of height h can have at most $2^{h+1} - 1$ nodes, we have:

$$n \leq 2^{h+1} - 1$$

Solving for h, we get:

$$n + 1 \leq 2^{h+1}$$

Taking the logarithm base 2 of both sides:

$$\log_2(n+1) \leq h + 1$$

Subtracting 1 from both sides:

$$\log_2(n+1) - 1 \leq h$$

Thus, the minimum height h is at least $\lceil \log_2(n+1) \rceil - 1$. This represents the smallest possible height for a tree with n nodes.

To express this in asymptotic terms, observe that $\lceil \log_2(n+1) \rceil - 1$ grows logarithmically with n. Therefore, the smallest height of any tree with n nodes satisfies:

$$h = \Omega(\log n)$$

□

6.2 Balanced Binary Trees

As you might have noticed we have unfortunately not met our target yet with our current construction of a binary tree

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Binary Tree	n	h	h	h	h
AVL Tree	n	$\log n$	$\log n$	$\log n$	$\log n$

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Binary Tree	$n \log n$	h	h	h	h
AVL Tree	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

Figure 6: Binary trees vs AVL tree

That's because we our binary tree has yet to achieve a state we call **balanced**. The scheme we will use to "balance" our tree is known as the **AVL** tree scheme.

Definition 24

A tree is **balanced** if its height is $O(\log n)$

Then all the $O(h)$ operations we talked about previously will take only $O(\log n)$ time.

Remark 25. *If you look at the graph of* $\log n$ *you will realize its actually closer to* $O(1)$ *than* $O(n)$

Proposition 26

$O(n)$ rotations can transform a binary tree to any other with same traversal order

Proposition 27

A binary tree with height-balanced nodes has height $h = O(\log n)$(i.e. $n = 2^{\Omega(h)}$

Proposition 28 (Local Rebalance)

Given a binary tree node $< B >$ whose skew is 2 and every other node in $< B >$'s subtree is height-balanced then $< B >$'s subtree can be made height-balanced via one or two rotations, after which $< B >$'s height is the same or one less than before.

Proposition 29 (Global Rebalance)

Add or remove a leaf from height-balanced tree T to produce tree T'. Then T' can be transformed into a height-balanced tree T'' using at most $O(\log n)$ rotations

Proposition 30

A height balanced tree is balanced.

Definition 31

A tree is said to be **height-balanced** if it satisfies the **AVL property**. Specifically define **skew** of a node to be the height of its right subtree minus the height of its left subtree(where the height of an empty subtree is -1). Then the node is height-balanced if its skew is either $-1, 0$ or 1. A tree is height-balanced if every node is height-balanced.

```
1         print("hi")
```

Listing 14: Code

```
1         print("hi")
```

Listing 15: Code

```
1         print("hi")
```

Listing 16: Code

```
1         print("hi")
```

Listing 17: Code