

# CSC265 Notes

MAX XU

'25 Fall

## Contents

1	Day 1: Review and ADTs (Sept 3, 2025)	2
2	Day 2: Priority Queues, Array Representation (Sept 5, 2025)	4
3	Day 3: Assignment Expectations, Heaps (Sept 8, 2025)	6
4	Day 4: Binomial Heaps (Sept 10, 2025)	9
5	Tutorial 1 (Sept 12, 2025)	10
6	Day 5: Trees (Sept 15, 2025)	11
7	Day 6: AVL Trees and Augmenting Data Structures (Sep 17, 2025)	14
8	Tutorial 2 (Sep 19, 2025)	16
9	Day 7: B-Trees (Sep 22, 2025)	17
10	Day 8: Average/Expected-Case Analysis (Sep 24, 2025)	18

## §1 Day 1: Review and ADTs (Sept 3, 2025)

### §1.1 ADTs

Abstract data types (ADTs) are a mathematical object, which has a set of operations. Examples include a set, a sequence, or a graph.

A *data structure* that implements an ADT provides a representation for the object in memory and algorithms for each operation. There may exist many different data structures that implement the same ADT, with varying running time for each operation.

#### Example 1.1 (Dictionary ADT)

The ADT consists of:

- **Object:** a set of elements, with each having a unique key from totally ordered universe  $U$
- **Operations:**
  - Insert**( $S, x$ ) adds an element with key  $x$  to the set  $S$  if  $S$  does not contain an element with key  $x$
  - Delete**( $S, x$ ) removes an element with key  $x$  from  $S$  if it exists
  - Search**( $S, x$ ) returns a pointer to the element in  $S$  with key  $x$ , or nil if such an element doesn't exist

Here we are not worried about key-value pairs, although a specific implementation of this ADT might. We could implement this as a set of keys, with no associated element for each key.

An example implementation of dictionary ADT using singly linked list (unsorted) can have the following properties:

### §1.2 Review

Let  $t(x)$  be the number of steps taken by an algorithm  $A$  on input  $x$ .

Let the worst case step complexity of  $A$  be  $T(n)$ , where

$$T(n) = \max\{t(x) \mid x \text{ is an input of size } n\}$$

Typically, this is very hard to determine exactly, which is why asymptotic notation is used instead, as it still captures how quickly  $T(n)$  grows with respect to  $n$ .

$T(n) \in O(f(n))$  if there exists a constant  $c, n_0 \in \mathbb{N}$ , for all  $n \in \mathbb{N}$  such that when  $n > n_0$ ,  $T(n) \leq cf(n)$ .  $T(n) \in \Omega(f(n))$  if there exists a constant  $c, n_0 \in \mathbb{N}$ , for all  $n \in \mathbb{N}$  such that when  $n > n_0$ ,  $cf(n) \leq T(n)$ .  $T(n) \in \Theta(f(n))$  if  $T(n) \in O(f(n))$  and  $T(n) \in \Omega(f(n))$ .

For upper bound, show that there exists positive constant  $c$  and for all  $n$  large enough, for every input of size  $n$ , the algorithm takes at most  $cf(n)$  steps. We write  $T(n) \in O(n)$ .

For lower bound, show that there exists positive constant  $c$  and for all  $n$  large enough, there exists some input of size  $n$  that makes the algorithm take at least  $cf(n)$  steps. We write  $T(n) \in \Omega(n)$ .

**BubbleSort( $A[1\dots n]$ )**

```

1: last = n
2: sorted = False
3: while not sorted do
4:   sorted = True
5:   for  $j = 1$  to  $\text{last} - 1$  do
6:     if  $A[j] > A[j + 1]$  then
7:       swap  $A[j]$  and  $A[j + 1]$ 
8:       sorted = False
9:     end if
10:  end for
11:  last = last - 1
12: end while

```

**Upper Bound**

The outer loop can occur at most  $n$  times, as last starts at  $n$  and decrements by 1 every time the outer loop runs. The inner loop goes from 1 to last - 1, meaning there are at most last - 1 iterations of the inner loop. This can be written as the sum

$$\sum_{i=1}^n (i - 1) = \left( \sum_{i=1}^n i \right) - n = \frac{n(n+1)}{2} - n = \frac{n^2 - n}{2}$$

meaning for this algorithm  $T(n) \in O(n^2)$ .

**Lower Bound**

We can pick the list  $A = [n, n - 1, \dots, 2, 1]$ . This list has the property that the first element is the largest number, barring the sorted block of size  $i - 1$  present at the end of the array at the start of the  $i$ -th iteration.<sup>1</sup> This means that it will take  $n - i$  swaps, which means it takes

$$\sum_{i=1}^n (n - i) = n^2 - \frac{n(n+1)}{2} = \frac{n^2 - n}{2}$$

steps total, meaning for this algorithm  $T(n) \in \Omega(n^2)$ .

**Theta Bound**

As  $T(n) \in \Omega(n^2)$  and  $T(n) \in O(n^2)$ , we have  $T(n) \in \Theta(n^2)$ .

---

<sup>1</sup>i think iterations start at 1 in this course?

## §2 Day 2: Priority Queues, Array Representation (Sept 5, 2025)

Last class we were looking at worst case analysis for the dictionary ADT.

### Sorted Array

**Insert(A, x)**  $\Theta(n)$ , necessary to shift elements 'greater' than  $x$  right by 1

**Delete**  $\Theta(n)$ , necessary to shift elements 'greater' than  $x$  left by 1

**Search(A,x)** Use binary search, known to take  $\Theta(\log n)$

### Direct Access Table

Suppose universe of keys  $U = \{1, \dots, n\}$  and  $n$  is small. As array indexing takes  $\Theta(1)$  time, all operations with the ADT take  $\Theta(1)$  time.

We have a slot for every single key in the universe, meaning we could hold the keys themselves, pointers to objects in the set, or even numbers, which we can extend to a multiset, etc.

### §2.1 Priority Queue ADT

The object is a set of elements each with a key (sometimes called priority, for consistency with CLRS), with the following operations

**Insert(S, x)** adds an element with key  $x$  to  $S$

**Max(S)** returns the largest key

**Remove(S)** removes element with largest key and returns key

### §2.2 Binary Heap ADT

The set  $S$  is stored in a complete binary tree (every level is filled except possibly for the lowest level, which is filled from left to right)

A complete binary tree with height  $h$  has  $2^h \leq n \leq 2^{h+1}$  nodes. With  $n$  nodes, it has height  $\lfloor \log n \rfloor$ . A max heap is a complete binary tree that satisfies the **max heap** property, where the key of each node is greater than or equal to the keys of its children (if they exist).

### §2.3 Array Representation of Binary Heaps

A binary heap can be represented as an array of length equal to the number of nodes,  $A.heapsize$ . In this course arrays are indexed starting at 1. Advantages include no need to store pointers

**Theorem 2.1.** Given an index  $i$  corresponding to a node in a binary heap represented as an array  $A$ , we have  $left(i) = 2i$ ,  $right(i) = 2i + 1$ ,  $parent(i) = \lfloor \frac{i}{2} \rfloor$

The max heap property for this representation is as follows:

$$\text{For all integers } i, 1 < i \leq A.heapsize, A[\lfloor \frac{i}{2} \rfloor] \geq A[i]$$

### §2.3.1 Operations

**Max(A)** return  $A[1]$ ,  $\Theta(1)$  worst case

**Insert(A, x)** Increment  $A.heapsize$ , set  $A[A.heapsize] = x$ . While the new inserted node has key larger than parent key, swap places with parent.<sup>2</sup>

**IncreaseKey(A, i, k)** Increase  $A[i]$  to  $k$ , takes  $\Theta(\log A.heapsize)$ , used for the ‘bubbling up’ procedure in INSERT.<sup>3</sup>

**MaxHeapify(A, i)**

**Preconditions:**  $A[i].left$  and  $A[i].right$  are max heaps  $A[i]$  may be smaller exactly one of its children.

**Postconditions:**  $A[i]$  is the root of max-heap with same elements

Swap the root with the larger of the children, so that the violation moves down the tree. Similar loop invariant to earlier, worst case runtime  $\Theta(\log n)$ .

**ExtractMax(A)** Set  $r = A[1]$ , for returning at the end. Swap  $A[A.heap.size]$  (last element) and  $A[1]$ . Then decrement  $A.heap.size$ , and call  $\text{MaxHeapify}(A, 1)$ , as putting the last element in the first index may violate the max heap property. Finally return the  $r$ .

### §2.3.2 Building a Max Heap

**Problem 2.2.** Given an array  $A[1..n]$  of  $n$  keys in arbitrary order, we want to modify  $A$ -in place such that it is a MaxHeap.

The naïve non-inplace solution is to perform  $n$   $\text{Insert}(A, x)$  operations, which will take  $\Theta(n \log n)$  steps. We could also reverse sort the array in decreasing order, which ensures the max-heap property is satisfied since the element at each index, which generally takes  $O(n \log n)$ .

<sup>2</sup>To prove correctness, the loop invariant is that below level  $n$ , there are no violations of the max heap property, then you move  $n$  to 0.

<sup>3</sup>I don’t really understand what  $k$  is here, will ask soon

## §3 Day 3: Assignment Expectations, Heaps (Sept 8, 2025)

The CLRS chapter on binomial heaps is posted on Quercus, because it was removed after the 2nd chapter.

### §3.1 Assignment Expectations

For reference, the solutions to Assignment 1 takes up roughly 4-5 pages. TLDR: Emulate the style of proof found in CLRS

#### Bounds

- Need to show upper and lower bound **separately**<sup>4</sup>
- To show  $f(n) \in O(n)$  or  $\Omega(n)$ , we don't need to find specific values for  $c, n_0$ , just need to simplify  $f$  such that it's clear

#### Algorithms

To give/describe an algorithm:

- Start with higher-level details, explain its phases
- Describe all parts of the algorithm in *English*, which can easily be converted to pseudo-code. Don't miss edge cases, and pointer updates when they happen. Your peers should be able to understand it clearly.
- Producing pseudo-code is **optional**, unless explicitly asked for
- You can use algorithms you previously encountered in class or in past courses
- Diagrams are helpful, but cannot replace your English description

#### Proof of Correctness

- Always say what you are proving. For long results, introduce lemmas, for short results, that is unnecessary. No flow of consciousness!
- Make clear what proof technique you're using: "We will prove by induction on the length  $n \dots$ ", "Suppose for contradiction that  $\dots$ "
- We can be less formal, we don't need to explicitly state what  $P$  is, no CSC240-indentation is strictly required. Still use indentations to break up paragraphs though, for clarity.

---

<sup>4</sup>note that no technique exists to directly show  $\Theta$ , without showing  $O$  and  $\Omega$  beforehand

### §3.2 Building Heaps

We want an  $\Theta(n)$  algorithm to make an array  $A$  possess the MaxHeap property (in-place).

#### BuildMaxHeap( $A[1\dots n]$ )

```

1: A.heapsize = n
2: for  $i = \lfloor \frac{n}{2} \rfloor$  to 1 do
3:   MaxHeapify( $A[i]$ )
4: end for

```

The loop invariant is that at the start of the iteration of the floor loop,  $A[i+1], \dots, A[n]$  is the root of a MaxHeap.

At most  $n$  calls will be made to the MAXHEAPIFY function, each taking  $O(\log n)$  steps in the worst case, meaning the total steps is  $O(n \log n)$ . This analysis isn't wrong per-se, it is a valid upper bound, just not a tight one.

The reason the bound wasn't tight is because for the lower levels, we are performing a lot less swaps than  $\log n$ . We observe that there are at most  $2^{h-i}$  nodes at the  $i$ -th level (note that the root is considered to be at level 0). MAXHEAPIFY on a node of height  $i$  takes at most  $ci$  steps.

$$\begin{aligned}
 T(n) &= c \cdot \sum_{i=1}^h 2^{h-i} i \\
 &= c \cdot 2^h \sum_{i=1}^h \frac{i}{2^i} \\
 &\leq cn \sum_{i=1}^{\infty} \frac{i}{2^i} \\
 &= 2cn \in O(n)
 \end{aligned}$$

For the lower bound, at least  $\lfloor \frac{n}{2} \rfloor$  calls to MAXHEAPIFY are made, because of the loop termination with each taking at least 1 step. Hence it is  $\Omega(n)$ .

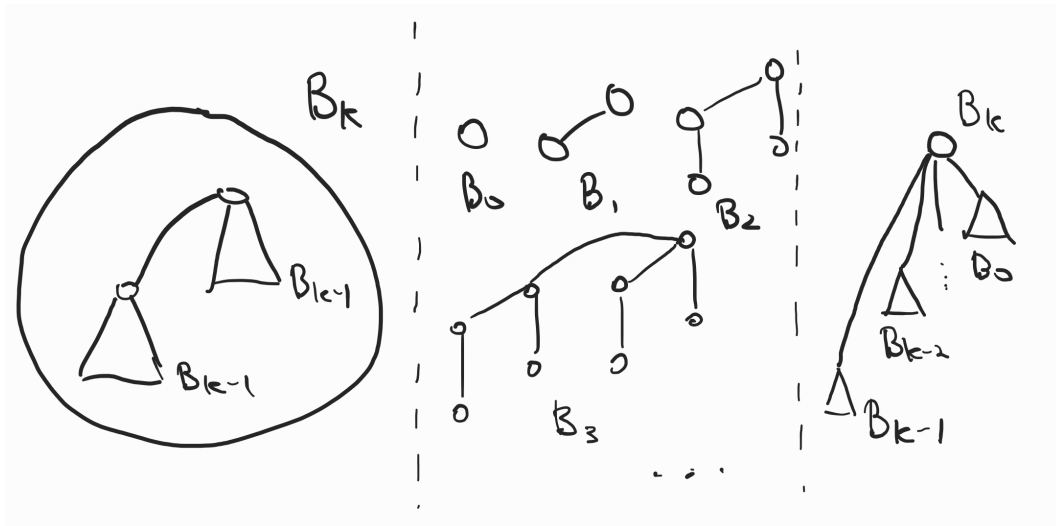
### §3.3 Binomial Heaps

We want *mergeable* priority queues, which support an additional operation.

**Union( $S_1, S_2$ )** Given two priority queues  $S_1$  and  $S_2$ , merge them into a single queue with the union of their elements. Assume the elements in  $S_1$  and  $S_2$  are disjoint.

**Definition 3.1** (Binomial Trees). A binomial tree of degree  $k$  is defined recursively:

- $B_0$  is a single node
- $B_k$  consists of two binomial trees  $B_{k-1}$ , with the root of 1 being the leftmost child of the root of the other



For a binomial tree of degree  $k$ ,

1. There are  $2^k$  nodes
2. The height is  $k$
3. There are exactly  $\binom{k}{i}$  nodes at depth  $i$ <sup>5</sup>
4. The root has degree  $k$  and its  $i$ -th child is the root of a binomial tree  $B_i$  (children are labelled  $k-1$  to  $0$  left to right)

Defining the operations this way makes the operations quite simple, we didn't have time to go over them this class.

<sup>5</sup>follows from combinatorics facts



## §4 Day 4: Binomial Heaps (Sept 10, 2025)

Last week, we looked at binary heaps. Today we will look at binomial heaps, whose definition were given last class. To store  $n$  nodes where  $n$  were not a power of 2, write  $n$  in binary

$$n = b_k 2^k + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

with  $b_i \in \{0, 1\}$ . The collection of binomial trees (binomial forest) where there is a binomial tree  $B_i$  of degree  $i$  for each  $b_i$  when  $b_i = 1$  would have  $n$  nodes.

A **binomial heap** is a binomial forest, where the roots of the binomial trees are in a LinkedList sorted by strictly increasing degree. Each binomial tree satisfies the max-heap property.

To represent a binomial heap in memory, have each node  $x$  store its key, degree, parent, left child (pointer), and right sibling (pointer to the sibling immediately to its right). Note that for the roots of trees  $B_i$ , there will be no right-sibling, so we can use the right-sibling field to achieve LinkedList behavior.

**Link( $H_1, H_2$ )** : link together two binomial trees with the same degree. need to update the degree (+1) and left child of the tree with the larger root key to the tree with the smaller root key. update the right sibling and parent of the tree with smaller root key.  $\Theta(1)$ , as this is a constant number of pointer updates.

**Union( $H_1, H_2$ )** : merge LinkedList of  $H_1$  and  $H_2$  sorted by increasing degree. link trees together by increasing degree until at most 1 tree of each degree link trees together by increasing degree until at most 1 tree of each degree link trees together by increasing degree until at most 1 tree of each degree.  $H_1$  has  $t_1$  trees,  $H_2$  has  $t_2$  trees, merging would take  $O(t_1 + t_2)$ , linking would require  $O(t_1 + t_2)$ , the sum is  $O(\log n)$

**Insert( $H, x$ )** : Create a single node  $N$  with key  $x$ , then call **UNION**( $H, N$ ). Note that if  $n = 2^{k-1}$ , then it would take  $O(\log n)$ , since you will be performing at least  $\log n$  merges. However if  $n = 2^k$ , then the new node will just get added to the linkedlist in constant time. For this reason you can't get a tight  $\Theta$  bound on this operation. Also need to update the max pointer here, if the new inserted node is greater than MAX pointer.

**Max( $H$ )** traverse the LinkedList and return the largest, takes  $O(\log n)$ . we can make this  $\Theta(1)$ , by maintaining a pointer to the root with the largest key.

**ExtractMax( $H$ )** : Traverse the roots to find binomial tree  $B_i$  with the largest key. Remove the root of  $B_i$ , create binomial heap  $H$  from its children. To ensure it's valid, reverse the order, this means sorted by increasing degree, then call **UNION**( $H, H'$ ). Traverse roots to find the largest key, to update the MAX pointer.

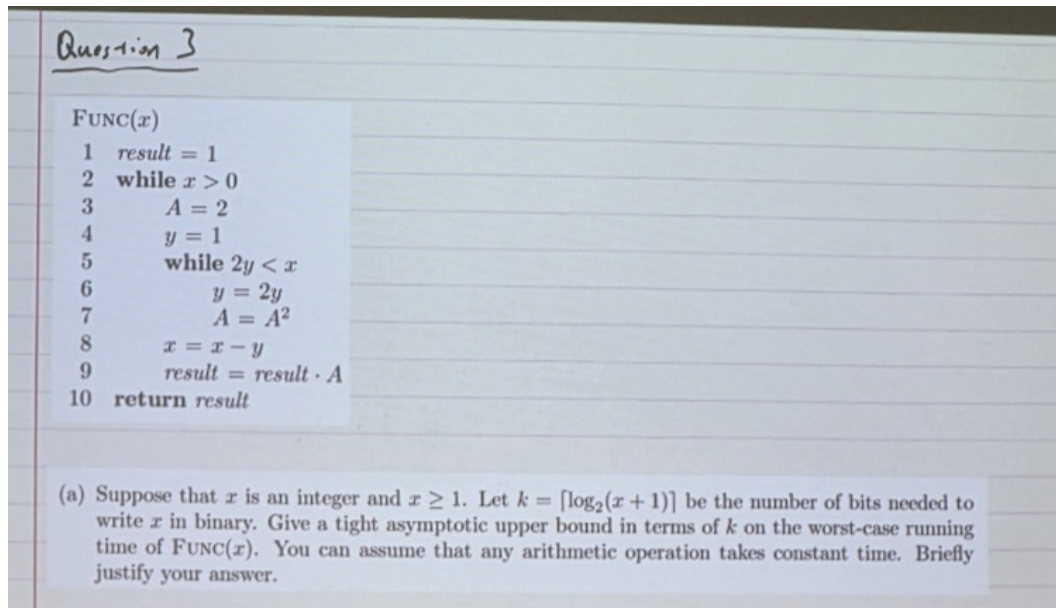
Next week, we will look at balanced binary search trees.

## §5 Tutorial 1 (Sept 12, 2025)

**Problem 5.1.** Show that building a binomial heap from an array  $A$  of  $n$  elements takes  $O(n)$  time.

**Problem 5.2.** Suppose you want to implement a dictionary using a direct access table in some universe  $U = \{1, \dots, n\}$  of keys. Unfortunately every array that you initialize will likely contain garbage information. Describe an implementation that takes  $O(1)$  time to initialize and support  $O(1)$  time search, insert, and delete.

**Problem 5.3.**



Question 3

```

FUNC(x)
1  result = 1
2  while x > 0
3      A = 2
4      y = 1
5      while 2y < x
6          y = 2y
7          A = A^2
8      x = x - y
9      result = result * A
10 return result
  
```

(a) Suppose that  $x$  is an integer and  $x \geq 1$ . Let  $k = \lceil \log_2(x+1) \rceil$  be the number of bits needed to write  $x$  in binary. Give a tight asymptotic upper bound in terms of  $k$  on the worst-case running time of  $\text{FUNC}(x)$ . You can assume that any arithmetic operation takes constant time. Briefly justify your answer.

**Problem 5.4.** Given an implementation for the predecessor operation on a binary search tree  $T$ , which gives a key  $k$ , return a pointer to a node  $T$  with the largest key less than  $k$ .

## §6 Day 5: Trees (Sept 15, 2025)

**Definition 6.1** (Binary Search Tree Property). For each node  $x$  in the binary tree,

- If  $y$  is in  $x$ 's left subtree,  $y.key \leq x.key$
- If  $z$  is in  $x$ 's right subtree,  $z.key \geq x.key$

A BST has  $\Theta(n)$  insert, delete, and search operations. The main issue is that the entire tree can behave like a linkedlist, where we do not make use of the other child field. We aim to solve this by using AVL trees, which have the aforementioned operations but in  $\log_2 n$  time.

Recall that  $\text{DELETE}(T, z)$  where  $z$  is a pointer to the node to be deleted, can be achieved as follows:

- If  $z$  is a leaf, replace the reference with NIL
- If  $z$  has 1 child, replace the reference to  $z$  with its child
- If  $z$  has 2 children, replace  $z.key$  with its successor key and delete the successor node.

The successor of  $z$  is the node you get by going right once and left repeatedly.

**Definition 6.2** (Ideally Height-Balanced). A binary tree is *ideally height-balanced* if every leaf has depth  $h$  or  $h - 1$ , and every node at depth less than  $h - 1$  has 2 children.

Reminiscent of a balanced binary tree, except the last row need not be filled in left to right order. An ideally height balanced tree with  $n$  nodes has height  $\lfloor \log_2 n \rfloor$  just like a balanced binary tree of the same size.

Creating an ideally balanced binary tree is difficult to do in logarithm time, unsure if there are any lower bound results on this. The difficulty of this problem motivates the following definition, where we loosen the idea of an ideally height-balanced tree

**Definition 6.3** (Height-Balanced). A binary tree is *height-balanced* if for every node, the heights of its left and right subtrees differ by at most 1.

**Definition 6.4** (Balance Factor).  $BF(x) = \text{height}(x_R) - \text{height}(x_L)$ . Let  $y$  be a node with 1 child. Define  $BF(\text{NIL}) = -1$ ,  $BF(y) = 0$ .

### §6.1 AVL Trees

We call a height-balanced BST an **AVL tree**. To be an AVL tree, for all nodes  $x$ ,  $BF(x) \in \{-1, 0, 1\}$ .

**Theorem 6.5.** The height of an AVL tree with  $n$  nodes is  $O(\log_2 n)$ .

*Proof.* Let  $M(h)$  denote the minimum number of nodes in an AVL tree with height  $h$ . By property of AVL tree, we have that  $M(0) = 1$ ,  $M(1) = 2$ ,  $M(h) = 1 + M(h-1) + M(h-2) = F_{h+3} + 1$ . From number theory facts (MAT315),  $F_n > \frac{\phi^n}{\sqrt{5}} - 1$ , where  $\phi = \frac{1+\sqrt{5}}{2}$ .  $n \geq M(h) > \frac{\phi^{h+3}}{\sqrt{5}} - 2$ . Then we get

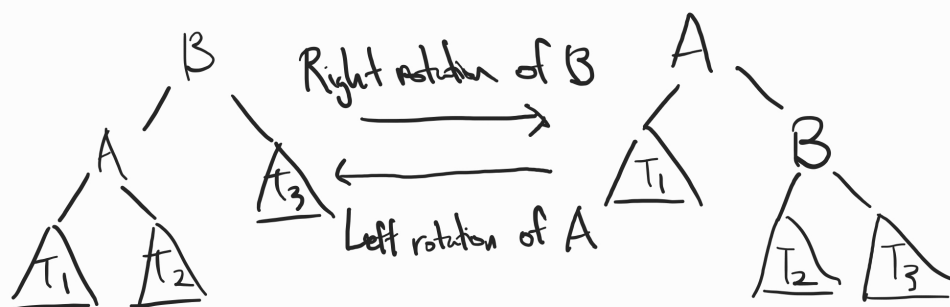
$$h < 1.44 \log_2(n + 1)$$

□

For the insert position, suppose that the insert has successfully occurred. Only the balance factors of nodes that have  $x$  in their subtree get updated. Some may end up with a more than 2 balance factor, which needs addressing.

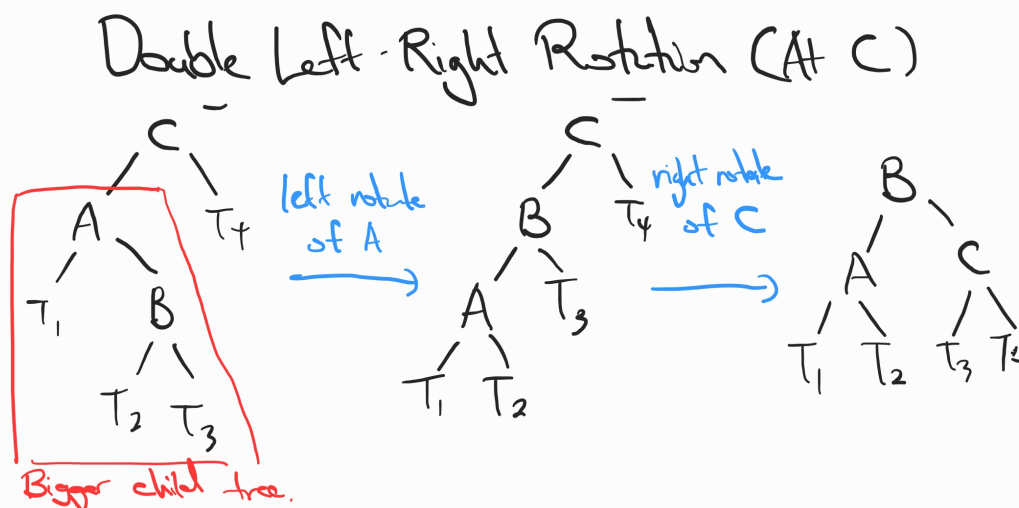
## §6.2 Tree Rotations

There are 4 types of tree rotations, being the single left and right rotation, and the double left-right and right-left rotation. After  $\text{INSERT}(x)$ , one of  $T_1$ ,  $T_2$ , or  $T_3$  will receive the additional node. In below figure,  $T_3$  for the left side tree and  $T_1$  for the right side tree getting  $x$  is not a concern, as in this case all of  $T_1, T_2, T_3$  are of height  $h$ .



You would use the single right rotation on the left-side figure when  $x$  is added to  $T_1$ , where  $A$  has a negative balance factor. We use the single left rotation on the right-side figure when the violation is in  $T_3$ , where  $B$  has a positive balance factor.

This leaves the case where  $A$  has a positive balance factor in the left figure ( $T_2$  has bigger height than  $T_1$ ), and the case where  $B$  has a negative balance factor ( $T_2$  has bigger height than  $T_3$ ). Note that the two cases are mirror images of each other, where we simply flip the signs of the balance factor:



In this figure note that  $A$  has a positive balance factor,  $C$  has a unacceptably negative balance factor, which we then handle using a double left right rotation. In the mirror image,  $A$  will have a negative balance factor,  $C$  will have an unacceptably positive balance factor, where we use a double right left rotation.

**Remark 6.6.** If you see this on the midterm and you're not sure which one you should do, just do all 4 rotations and eventually 1 will work.

To insert on an AVL tree, we perform steps as follows:

- Insert new node  $x$  following standard BST insert
- From inserted node up to root, update balance factor and do rotations to fix the imbalance

**Claim 6.7** — We only need to do at most one single or double rotation for INSERT.

You would prove this by doing a ton of cases. Also start assignment 2 early! It's hard - Jeremy Ko.

## §7 Day 6: AVL Trees and Augmenting Data Structures (Sep 17, 2025)

We start with some administrative details. “You guys definitely need some feedback before the next assignment. Half of you write good, the other half - not so good. Well the first assignment is always a bit rough anyways. Usually when Faith teaches this class, the average is a 50%-60%”. A2 official solutions are around 6-7 pages, so the page limit shouldn't be an issue. New office hours dropping Friday 4 to 5pm.

We continue with AVL trees. Desired properties of rotations while using INSERT include: constant step count, preserve the BST property, fix the imbalance at the node. Let  $m$  denote the first node where the balance factor becomes unacceptable. The rotation should preserve the height of the subtree located at  $m$  before insertion and after rotation.

For AVL trees, each node  $x$  stores: *key*, *left*, *right*, *BF* (balance factor), and *parent*. **If inserting a node does not alter the height of a subtree, then it wouldn't have caused an imbalance.**

**Delete**( $T, z$ ) Begin with standard BST delete, reduces to removing a leaf node. Do cases on a node  $z$ . If  $z$  is a leaf, we are done. If  $z$  has 1 child, the other child must be NIL, by height-balanced property replace  $z$  with  $y$ . From the leaf to root, you would need to update the balance factors and do rotations as necessary. If  $z$  has 2 children, replace  $z$  with its successor which goes back to case 0 or 1 child.

**Theorem 7.1.** DELETE( $T, z$ ) may perform up to  $O(\log_2 n)$  rotations, while INSERT( $T, z$ ) does at most 1.

### §7.1 Augmenting Data Structures

We want to augment existing data structures with new information to implement new operations.

#### InOrderTraversal( $T$ )

- 1: INORDERTRAVERSAL( $T.left$ )
- 2: Print  $T.key$
- 3: INORDERTRAVERSAL( $T.right$ )

An order statistic tree is an augment AVL tree supporting the new operations

**OS-Select**( $T, i$ ) : returns pointer to the  $i$ -th smallest element, according to the in-order Traversal

**OS-Rank**( $T, x$ ) : given pointer to  $x \in T$ , returns the position of  $x$  to the in-order traversal

We can't have nodes store the rank, because this would require linear time updates when you insert a new smallest key. Instead we store *size*, that is the number of nodes in the subtree rooted at  $x$ .

Now we need to modify INSERT/DELETE. Suppose  $x.left$  and  $x.right$  are properly updated. To be careful, we define  $NIL.size = 0$ , then compute  $x.size = x.right.size + x.left.size + 1$ . Also remember to update the size field during a rotation.

We are lucky that we can do updates to *size* from top-down during an insert. For other statistics, we aren't so lucky, e.g. *sum*. Typically we do it from bottom to top, because that way it always works.

**OS-Select( $x, i$ )**

```
1:  $r = x.left.size + 1$ 
2: if  $r = i$  then
3:   return  $x$ 
4: else if  $i < r$  then
5:   return OS-SELECT( $x.left, i$ )
6: else if  $i > r$  then
7:   return OS-SELECT( $x.right, i - r$ )
8: end if
```

## §8 Tutorial 2 (Sep 19, 2025)

**Theorem 8.1.** Let  $f$  be a field that augments an AVL tree  $T$  of  $n$  nodes, and suppose that the value of  $f$  for each node  $x$  depends on only the information in nodes  $x$ ,  $x.left$ ,  $x.right$ . Then we can maintain the values of  $f$  in all nodes of  $T$  during insertion and deletion without asymptotically affecting the  $O(\log n)$  performance of these operations.

Examples of such fields  $f$  include

**size**  $x.size = x.right.size + x.left.size + 1$

**max** the maximum data in the subtree rooted at  $x$ .  $x.max = \max\{x.left, x.right, x.data\}$ , or if we are max-ing over the keys,  $x.max = x.right.max$  if  $x.right \neq \text{NIL}$ ,  $x.key$  otherwise

**sum** sum of all keys in the subtree

**min** minimum data/key in the subtree

**Problem 8.2.** Count the number of inversions of array  $A$  of length  $n$ . An inversion is a pair  $(i, j)$  with  $i < j$  and  $A[i] > A[j]$ .

**Problem 8.3.** Given an example of an  $n$ -node AVL tree such that it contains a node, which when deleted from the tree requires  $\Omega(\log_2 n)$  rotations to rebalance the tree.

Hint: give a recursive definition for a family of trees  $\{T_k\}_{k \in \mathbb{N}}$  of minimal size AVL tree of height  $h$ , for  $h \in \mathbb{N}$ , and consider deletion from  $T_h$ .



## §9 Day 7: B-Trees (Sep 22, 2025)

Today is the last day of balancing trees. Next class, we will study randomization and hashing, so make sure to brush up on probability.

A **B-tree** is a balanced tree with a large ‘branching factor’, meant to be a generalization of a BST. Every node has a lot of children, and that each node stores a lot of keys. Beefier nodes means you traverse to the lower nodes less frequently, which generally translates to faster times: a useful property for secondary storage.

**Definition 9.1** (B-Tree). B-Trees have the following properties

1. Every non-root node contains  $t - 1 \leq n(x) \leq 2t - 1$  keys. The root node contains at most  $2t - 1$  keys, but can have as few as 1 key.  $t \geq 2$  is the minimum degree.
2. Every node has  $n(x)+1$  children (thus between  $t$  and  $2t$  children), being  $c_1, \dots, c_{n(x)+1}$ .
3. Every leaf has the same depth.

Note that this works best where  $t$  is quite large.

**Theorem 9.2.** If  $n \geq 1$ , a B-tree with  $n$  keys and min degree  $t$  has height  $h \leq \log_t(\frac{n+1}{2})$ .

*Proof.* The root node has at least 1 key. The other nodes have at least  $t - 1$  keys. At depth 0, there is 1 node, being the root node. The root node has  $n(x) + 1 = 1 + 1 = 2$  children. At depth 2, each child has  $2t$  children, and at depth  $h$  it is  $2t^{h-1}$ .

$$n \geq 1 + \sum_{i=1}^h (t-1)2^{i-1} = 2t^h - 1$$

Thus  $h \leq \log_t(\frac{n+1}{2})$ . □

**Search**( $T, k$ ) : First use linear search to find  $k$  in node  $x$ , and return if found.

Else find the successor<sup>6</sup>  $k_i$  of  $k$ , and search the subtree rooted at  $c_i$ . If  $k$  is larger than all key, search  $c_{h(x)+1}$ .

**Insert**( $T, k$ ) : Search for the leaf node to insert  $k$ . If it has less than  $2t - 1$  keys, the insert is easy and we are done. If the current node  $x$  has  $2t$  keys after  $k$  is added, we need to split the overflowing node as follows:

Let  $k'$  be the median (key at rank  $t$ ). Create 2 new nodes, one with  $t - 1$  keys less than  $k'$ , and one with  $t$  keys greater than or equal to  $k'$ . Recursively insert  $k'$  to the parent as the new divider between the 2 new nodes. If there is no new parent, then create new root with key  $k'$  and 2 children.

**Delete**( $T, k$ ) : Maintain that the node we enter has at least  $t$  keys (unless it's the root).

We must handle a lot of cases:

1. If  $x$  is a leaf, then remove  $k$  from  $x$  if it exists.
2. If  $x$  is an internal node with key  $k_i$ :
  - a) If child  $y$  that precedes  $k$  in  $x$  has at least  $t$  keys. Let  $k'$  be the largest key in the rightmost leaf of  $y$ . Recursively delete  $k'$  from  $y$ , replace  $k$  with  $k'$  in  $x$ . We don't need to know what  $k'$  is to perform this recursive call.
  - b) If child  $z$  that follows  $k$ , then it is symmetric c) In the case where both  $y$  and  $z$  have  $t - 1$  keys, then TODO
3. If  $x$  is an internal node and  $k$  not in  $x$ , let  $c_i$  be the next node to visit. If  $c_i$  has  $t - 1$  nodes,

---

<sup>6</sup>in-order successor?

## §10 Day 8: Average/Expected-Case Analysis (Sep 24, 2025)

Probability knowledge that you need: basic probability, expectation, linearity of expectation, Markov's inequality. If something is not in CLRS, it will be given to you, so no surprises (hopefully).

Let  $A$  be an algorithm. Let  $S_n$  be a sample space containing inputs of size  $n$  for  $A$ . Let  $t_n(x)$  be the number of steps  $A$  takes on input  $x \in S_n$ . Suppose a probability distribution  $\Pr$  is given over  $S_n$ . Typically, this is given by the problem or assumed (e.g. uniform).

**Definition 10.1** (Average Case Step Complexity).

$$T_{AV}(n) = \mathbb{E}[t_n] = \sum_{x \in S_n} \Pr(x) t_n(x) = \sum_{i \text{ in range of } t_n} i \Pr(t_n = i)$$

$\Pr(t_n = i) = \Pr(t_n^{-1}(i))$ , where  $t_n^{-1}$  is the inverse image, the probability of the set that gets mapped to  $i$  by  $t_n$ . Now, to compute  $T_{AV}(n)$ , we go through the following steps:

1. Define a sample space  $S_n$ .  
Choosing an easy sample space will make your life easier.
2. Given  $\Pr$  distribution over  $S_n$ .
3. Define  $t_n$  and any other random variables.  
Choose something reasonable.
4. Decide which  $T_{AV}(n)$  formula you want to use.  
This step relies heavily on intuition and experience, but typically you try to partition the sample space using another random variable (like an indicator).

If the step count is based on the number of comparisons, then only the relative order among elements does matter. An easy  $S_n$  can be the permutations of the set  $\{1, \dots, n\}$ , and a simple  $\Pr$  is the uniform distribution.<sup>7</sup>

There exist 'nicer' ways to find  $T_{AV}(n)$  for some algorithms that don't require solving recurrences, but they might be harder to find.

Let  $t_n(x, \sigma)$  be the number of steps taken by a randomized algorithm  $A$  on input  $x$  with a sequence of random choices  $\sigma$ . It can be very very hard to define  $\sigma$ , especially if future choices depend on past choices, so generally we hand-wave the specifics.

**Definition 10.2** (Expected Case Step Complexity). The expected step complexity of  $A$  on input  $x$  is

$$\mathbb{E}_\sigma[t_n(x, \sigma)] = \sum_{\sigma} \Pr(\sigma) t_n(x, \sigma)$$

**Definition 10.3** (Worst Case Step Complexity).

$$T_{EX} = \max_{x \in S_n} \{\mathbb{E}_\sigma[t_n(x, \sigma)]\}$$

<sup>7</sup>will we ever encounter a continuous distribution in cs?