

# Typst for CS240

## Contents

<b>1. Introduction .....</b>	<b>2</b>
1.1. For New Typst Users .....	2
<b>2. Pseudocode .....</b>	<b>3</b>
<b>3. Trees .....</b>	<b>4</b>
3.1. Binary tree .....	4
3.2. AVL tree .....	5
3.3. Tries .....	7
3.4. Quadtree .....	9
<b>4. Skiplists .....</b>	<b>10</b>
<b>5. Contribution .....</b>	<b>12</b>

# 1. Introduction

The raison d'être of this document is to help its readers save time that would otherwise have been spent searching for answers on the internet or troubleshooting their own .typ files while working on CS240.

This document is intended for 2 types of readers:

1. People who are new to Typst and wish to learn how to use it as they take CS240.
2. Existing Typst users who are taking CS240.

## 1.1. For New Typst Users

For both types of readers, this document will act as a resource showing how to effectively format homework assignments and draw relevant data structures for CS240.

For those of you that fall into the first camp of readers, those unfamiliar with Typst, here is a brief explanation of what Typst is from the official documentation:

Typst is a new markup-based typesetting system for the sciences. It is designed to be an alternative both to advanced tools like LaTeX and simpler tools like Word and Google Docs. Our goal with Typst is to build a typesetting tool that is highly capable and a pleasure to use.

— [Typst Docs](#)

To familiarize yourselves with Typst I would suggest you read [the official Typst tutorial](#). You won't need to read the whole thing, however this document assumes you know the basics of using Typst. You should be comfortable with the following before continuing:

1. How you will use Typst. Typst has an excellent online editor as well as a CLI. Which will you use?
2. Basic syntax.
3. Using math mode.
4. Simple formatting.
5. Using external packages.

Once you are familiar with Typst and comfortable using it, you will be able to make the most of this guide as it will show you how you can use Typst to succeed in CS240.

## 2. Pseudocode

For writing pseudocode, I recommend using the [algo](#) package. Here is a basic example of using algo from the package's documentation:

```
#import "@preview/algo:0.3.3": *
#algo(
  title: "Fib",
  parameters: ("n",)
)[
  if $n < 0$:#i\           // use #i to indent the following lines
    return null#d\       // use #d to dedent the following lines
  if $n = 0$ or $n = 1$:#i #comment[you can also]\
    return $n$d #comment[add comments!]\
  return #smallcaps("Fib")$(n-1) +$ #smallcaps("Fib")$(n-2)$
]
```

The output from the code will look like this:

```
FIB(n):
1  if  $n < 0$ :
2    return null
3  if  $n = 0$  or  $n = 1$ :           // you can also
4    return  $n$                  // add comments!
5  return  $\text{FIB}(n - 1) + \text{FIB}(n - 2)$ 
```

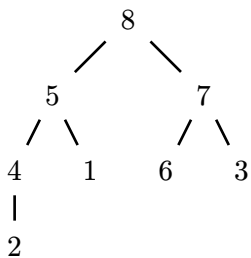
### 3. Trees

#### 3.1. Binary tree

Using the [cetz](#) package we can draw binary trees:

```
#canvas(length: 1cm, {  
  import draw: *  
  
  set-style(content: (padding: .2),)  
  
  tree.tree(  
    (  
      [$8$], // root node  
      (  
        [$5$],  
        (  
          [$4$],  
          (  
            [$2$],  
          ),  
        ),  
        (  
          [$1$],  
        ),  
      ),  
      (  
        [$7$],  
        (  
          [$6$],  
        ),  
        (  
          [$3$],  
        ),  
      ),  
    ),  
  )  
})
```

Which creates the following binary tree:



### 3.2. AVL tree

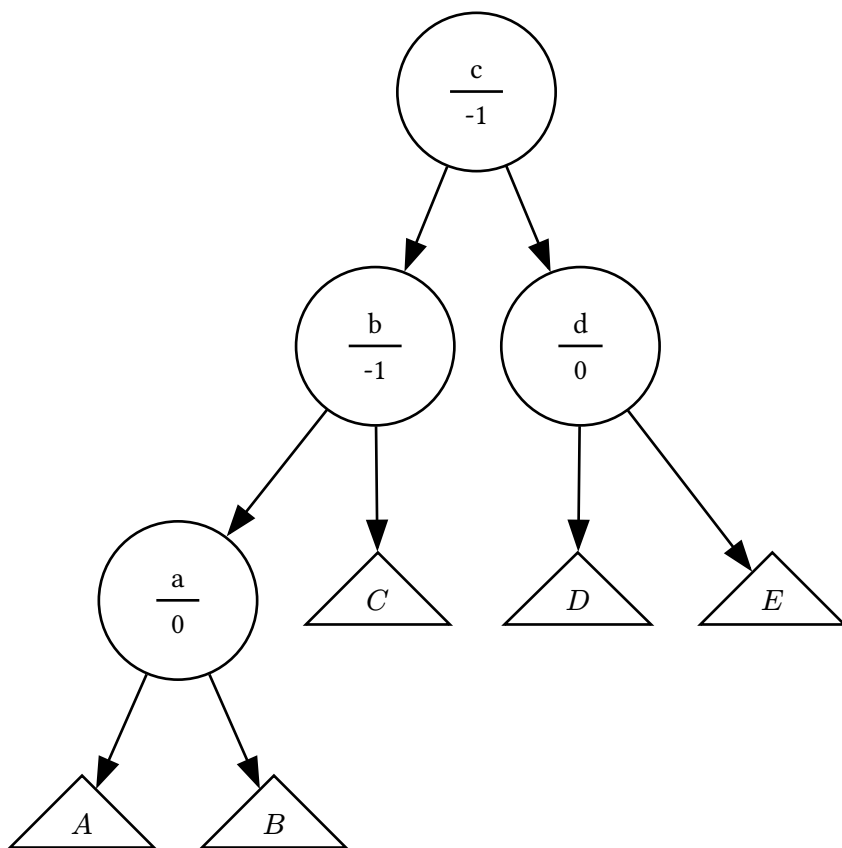
We can use the [diagraph](#) package to draw AVL trees:

```
// import the diagraph package
#import "@preview/diagraph:0.3.0": *

// create a template for our nodes that will allow us to have value and balance
#let node(val, bal) = {
  table(
    stroke: none,
    columns: (auto),
    align: center,
    [#val],
    table.hline(),
    [#bal]
  )
}

#raw-render(
//   replace this comment with 3 backticks -> `
  digraph G{
    // decide shape for each node
    a [shape=circle]
    b [shape=circle]
    c [shape=circle]
    d [shape=circle]
    A [shape=triangle]
    B [shape=triangle]
    D [shape=triangle]
    E [shape=triangle]
    // make an edge between nodes
    c -> b
    c -> d
    b -> a
    a -> A
    a -> B
    d -> D
    // you may have to declare some nodes later in the interest of ordering
    C [shape=triangle]
    d -> E
    b -> C
  }
//   replace this comment with 3 backticks -> `
  ,
  // label the nodes, this is where we apply our template
  labels: (
    "b": [#node("b", "-1")],
    "a": [#node("a", 0)],
    "c": [#node("c", "-1")],
    "d": [#node("d", 0)]
  )
)
```

Creates the following AVL tree:



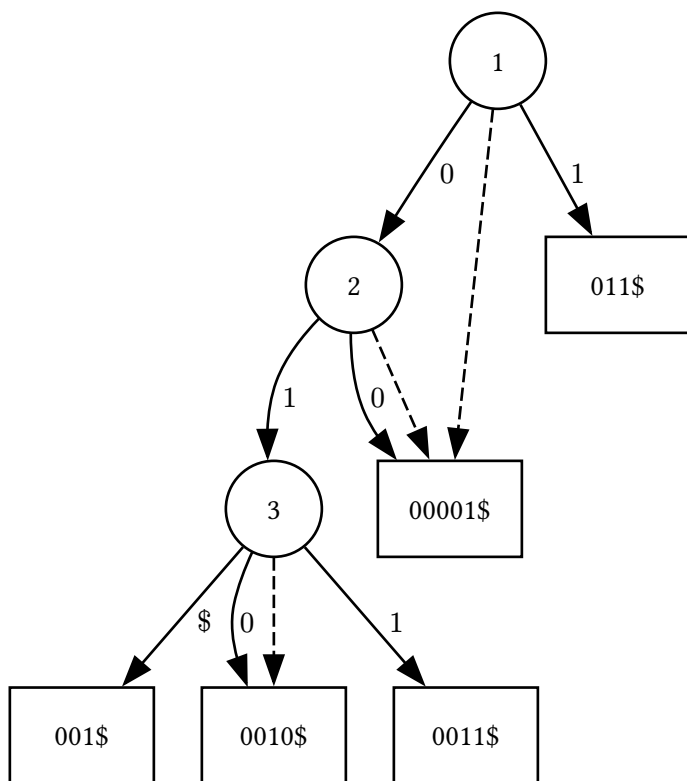
There is likely a prettier way to this with `cetz` instead of `diagraph`.

### 3.3. Tries

To draw tries we will once again make use of the [diagraph](#) package.

```
#raw-render(  
  
  //   replace this comment with 3 backticks -> `  
  digraph G{  
    // decide shapes for nodes  
    A [shape=circle]  
    B [shape=circle]  
    C [shape=circle]  
    D [shape=rectangle]  
    E [shape=rectangle]  
    F [shape=rectangle]  
    G [shape=rectangle]  
    H [shape=rectangle]  
    // add labels and or styls to edges  
    A -> B [label="0"]  
    A -> H [label="1"]  
    B -> G [label="0"]  
    B -> C [label="1"]  
    C -> D [label="\$"]  
    C -> E [label="0"]  
    C -> F [label="1"]  
    A -> G [style=dashed]  
    B -> G [style=dashed]  
    C -> E [style=dashed]  
  }  
  //   replace this comment with 3 backticks -> `  
  ,  
  // label the nodes  
  labels: (  
    A: [1],  
    B: [2],  
    C: [3],  
    D: [001\$],  
    E: [0010\$],  
    F: [0011\$],  
    G: [00001\$],  
    H: [011\$]  
  )  
)
```

Which creates the following trie:

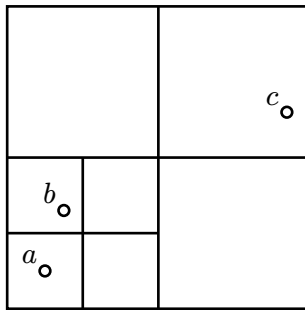


There is likely a prettier way to this with `cetz` instead of `diagraph`.



### 3.4. Quadtree

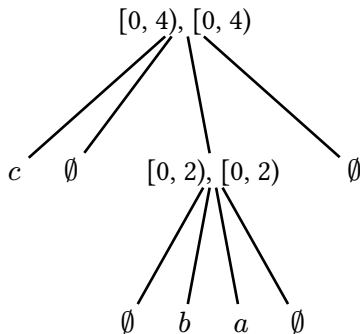
Consider the following grid:



If we wished to draw a quadtree for the points in the grid, we would make use of the [cetz](#) package like so:

```
#import "@preview/cetz:0.2.2": *
#canvas({
  import draw: *
  set-style(content: (padding: .1))
  tree.tree(([\[0, 4\), \[0, 4\)], // this is the root node
    ([\($c$)],
    ([\($nothing$)],
    ([\[0, 2\), \[0, 2\)],
      [\($nothing$)],
      [\($b$)],
      [\($a$)],
      [\($nothing$)],),
    ([\($nothing$)],
  ),
  spread: .75, // spread controls how wide each child spreads from its parent
  grow: 2) // grow controls how far each child is from its parent
})
```

Which outputs the following tree:

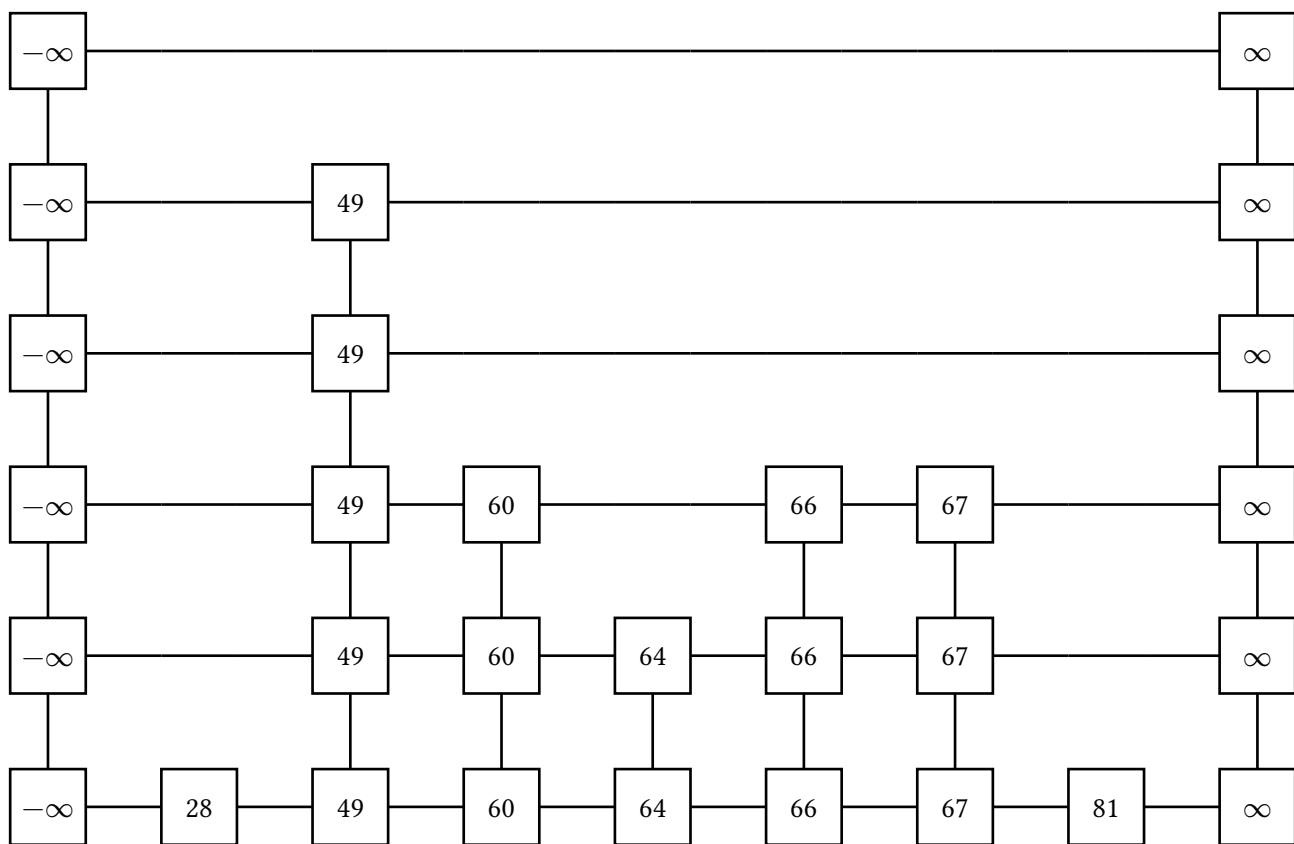


## 4. Skiplists

The following is a function to automatically draw skip lists from an array of values using the [cetz](#) package:

```
#import "@preview/cetz:0.2.2": *
#let skiplist(values) = {
  canvas({
    import draw: *
    let h = values.map(val => val.last().sorted().last() + 1)
    let l = values.len() + 1
    // Create sentries
    for i in range(0, h - 1) {
      rect((0, 2 * i), (rel: (1, 1)))
      content((0.5, 2 * i + 0.5), [${- infinity$})
      line((0.5, 2 * i + 1), (0.5, 2 * (i + 1)))
      rect((2 * l, 2 * i), (rel: (1, 1)))
      content((2 * l + 0.5, 2 * i + 0.5), [${infinity$})
      line((2 * l + 0.5, 2 * i + 1), (2 * l + 0.5, 2 * (i + 1)))
    }
    // Add top level
    rect((0, 2 * (h - 1)), (rel: (1, 1)))
    content((0.5, 2 * (h - 1) + 0.5), [${- infinity$})
    rect((2 * l, 2 * (h - 1)), (rel: (1, 1)))
    content((2 * l + 0.5, 2 * (h - 1) + 0.5), [${infinity$})
    // Add remaining nodes
    for (index, (val, height)) in values.enumerate(){
      for i in range(1, h){
        if i < height {
          rect((2 * (index + 1), 2 * i), (rel:(1, 1)))
          content((2 * (index + 1) + 0.5, 2 * i + 0.5), [#val])
          line((2 * (index + 1) + 0.5, 2 * i - 1), (2 * (index + 1) + 0.5, 2 * i))
        } else {
          line((2 * (index + 1), 2 * i + 0.5), (rel: (1, 0)))
        }
      }
      rect((2 * (index + 1), 0), (rel: (1, 1)))
      content((2 * (index + 1) + 0.5, 0.5), [#val])
    }
    for i in range(0, l){
      for j in range(0, h){
        line((2 * i + 1, 2 * j + 0.5), (rel: (1, 0)))
      }
    }
  })
}
// this is all you have to change
// every entry in the array is of the form (value, height)
#let values = ((28, 1), (49, 5), (60, 3), (64, 2), (66, 3), (67, 3), (81, 1))
#skiplist(values)
```

The output from the code will look like this:



## 5. Contribution

Contributions and feedback are welcomed and encouraged as to serve its purpose this guide must evolve alongside the CS240 curriculum and the Typst project. You can contribute and report issues via [Github](#).