Typst for CS240

Contents

1. Introduction

The raison d'etre 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.

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.

1.1. For New Typst Users

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 and consult the Typst Undergrad guide document. You won't need to read the whole tutorial, 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.

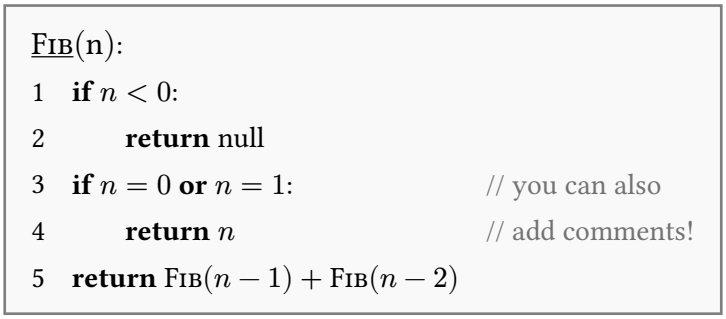
Once you are familiar with Typst and comfortable using it, you will be able to make the most of this guide as it will show you how you can use Typst to succeed in CS240.

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:0.3.4": *
#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:



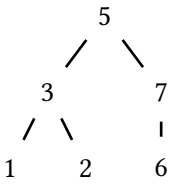
3. Trees

3.1. Binary tree

Using the **cetz** package we can draw binary trees:

```
#canvas({
  import draw: *

  let data = ([5], ([3], [1], [2]), ([7], [6]))
  tree.tree(
    data,
    draw-node: (node, ..) => {
      content((), text(black, [#node.content]))
    },
    draw-edge: (from, to, ..) => {
      let (a, b) = (from + ".center", to + ".center")
      line((a, .4, b), (b, .4, a))
    }
  )
})
```



Alternatively, and perhaps both easier and faster. We can use the **Fletcher** package to create a function that automatically draws binary trees from an array like so:

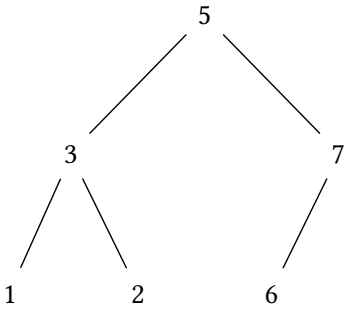
```
#let binary_tree(values) = {
  diagram(
  // Draw the nodes
    for i in range(0, int(calc.log(values.len(), base: 2)) + 1){
      for j in range(calc.min(values.len() - 1, calc.pow(2, i) - 1), calc.min(values.len(),
calc.pow(2, i + 1) - 1)){
        node(((calc.pow(2, calc.floor(calc.log(values.len(), base: 2)))/calc.pow(2, i + 1))
* (1 + 2 * (j + 1 - calc.pow(2, i))), i),
          str(values.at(j)),
          name: str(values.at(j)))
      }
    },
  // Draw the edges
    for i in range(1, int(calc.log(values.len(), base: 2)) + 1){
      for j in range(calc.min(values.len() - 1, calc.pow(2, i) - 1), calc.min(values.len(),
calc.pow(2, i + 1) - 1)){
        edge(((calc.pow(2, calc.floor(calc.log(values.len(), base: 2)))/calc.pow(2, i + 1))
* (1 + 2 * (j + 1 - calc.pow(2, i))), i),
          ((calc.pow(2, calc.floor(calc.log(values.len(), base: 2)))/calc.pow(2, i + 1)) *
(2 * (calc.rem(j, 2) + j + 1 - calc.pow(2, i))), i - 1))
      }
    }
```

```
  )
}

// This is all you need to modify, it is a binary tree as it would be stored in an array
#let my_tree = (5, 3, 7, 1, 2, 6)

#binary_tree(my_tree)
```

This produces the following tree:



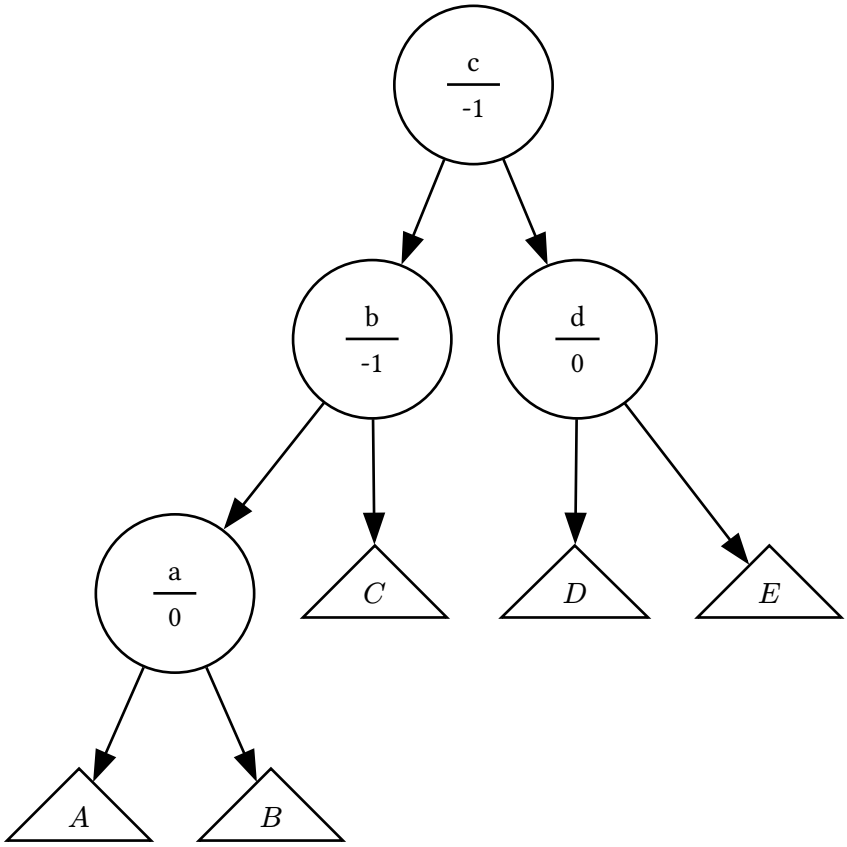
3.2. AVL tree

We can use the **diagraph** package to draw AVL trees:

```
// 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 or fletcher instead of diagraph.

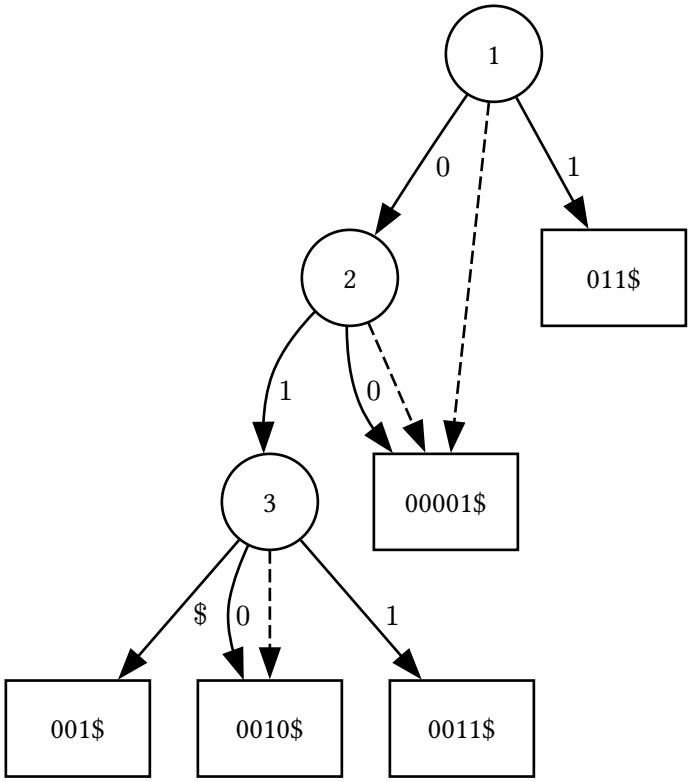
3.3. Tries

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\$]
    )
  )
```

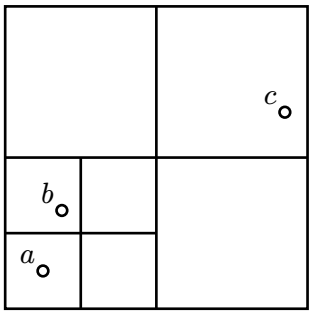
Which creates the following trie:



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

3.4. Quadtree

Consider the following grid:

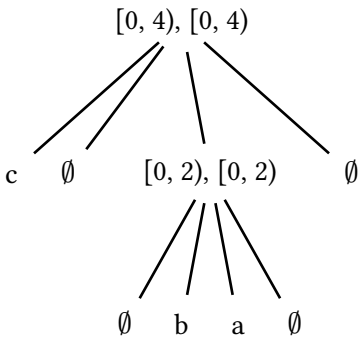


If we wished to draw a quadtree for the points in the grid, we would make use of the **cetz** package like so:

```
#canvas({
  import draw: *

  let data = ([\[0, 4\), \[0, 4\)],
                ([c]),
                ([$nothing$]),
                ([\[0, 2\), \[0, 2\)],
                  [$nothing$],
                  [b],
                  [a],
                  [$nothing$]),
                ([$nothing$]),)
  tree.tree(
    data,
    draw-node: (node, ..) => {
      content((), text(black, [#node.content]))
    },
    draw-edge: (from, to, ..) => {
      let (a, b) = (from + ".center", to + ".center")
      line((a, .4, b), (b, .4, a))
    },
    spread: .75, //controls how far apart nodes spread from each other
    grow: 2 // controls how for down nodes spread from each other
  )
})
```

Which outputs the following tree:

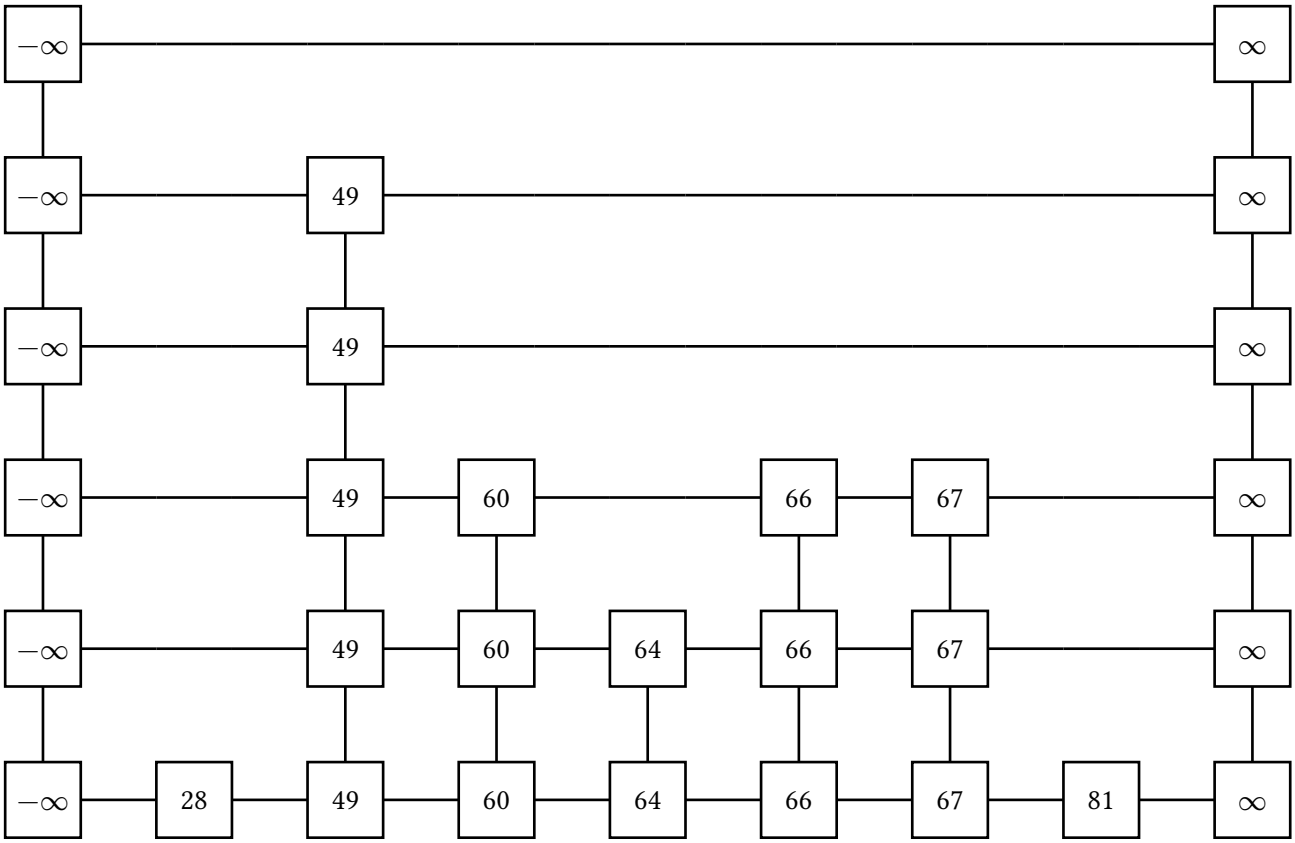


4. Skiplists

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

```
#import "@preview/cetz:0.3.2": *
#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:


−∞ ∞
−∞ 49 ∞
−∞ 49 ∞
−∞ 49 60 66 67 ∞
−∞ 49 60 64 66 67 ∞
−∞ 28 49 60 64 66 67 81 ∞

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.