CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
State machines
State machines

- Known as Finite State Machines (FSM)
- Mathematical model of computation where system has a single state
- Triggering conditions can change that state (events)
- FSMs are defined completely by both their states and the transitions between them
State machines

- The system only exists in one state at a time
- State persists through time
- Certain conditions can change the state to another state
  - These are specific to the current state
State machines

Transitions

• Events trigger transitions between states
• A combination of events can be used
• Transitions are all mutually exclusive
• At any given time there must be a valid transition for a state
  – If no transition is explicitly stated, an implied loopback transition exists
State machines

Benefits

- Provides a formal way to reason about a system
  - Allows for testing before writing any code
- Can be easily visualized
- Are language independent
- States are only dependent on current state and current inputs
State machines

When to use

- Can be used whenever there are a finite set of states for the system
  - Car transmission
  - Stoplight
  - Vending machine
  - Toaster oven
  - Video games
State machines
Use in the SeaSlug

- Transmission protocol
  - Mission management
  - Parameter management
- Operating state
  - Handling errors/system faults
- Calibration
  - Rudder
  - Radio controller
State machines

Diagrams

condition2
action2

STATE_1

action

STATE_2

condition1
action1
typedef enum { STATE_1, STATE_2 } SystemState;
static SystemState state;
{
    switch (state) {
    case STATE_1:
        default:
            if (condition1) {
                Action1();
                state = STATE_2;
            }
            break;
    case STATE_2:
        if (condition2) {
            Action2();
            state = STATE_1;
        } break;
    }
}
State machines
Integrating

Example

typedef enum { STATE_1, STATE_2 } SystemState;
static SystemState state;
int main (void) {
    // Initialize system

    // Event loop
    while (1) {
        // State machine
        switch (state) {
            ...
        }
    }
}
Combination 3-digit Lock 314

123
456
789

NONE correct

NOT 1

NOT 4

ONE correct

UNLOCK

4 cent

Lock

Lock

3 cent

Lent

Lent

Lent
CMPE-013/L

Introduction to “C” Programming

Maxwell J Dunne
Recursion
Recursion

4!  =  4 × 3 × 2 × 1

- Solving problems by breaking them into smaller parts
- "divide and conquer"
- Relies on the problem having self-similarity

Example

```c
int Factorial(int n)
{
    if (n <= 1) {
        return 1;
    }
    return n * Factorial(n - 1);
}
```
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

\[ 5! = 5 \times 4! \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Partial results pushed on stack
Factorial term replaced with result

Stack:

[3] \[4! = 4 \times 3!\]

[4] \[5! = 5 \times 4!\]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Factorial(5)</th>
<th>Factorial(4)</th>
<th>Factorial(3)</th>
</tr>
</thead>
</table>

Partial results pushed on stack
Factorial term replaced with result

[2] \(3! = 3 \times 2!\)
[3] \(4! = 4 \times 3!\)
[4] \(5! = 5 \times 4!\)
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

**Stack** (top)

<table>
<thead>
<tr>
<th>Factorial(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(3)</td>
</tr>
<tr>
<td>Factorial(2)</td>
</tr>
<tr>
<td>2! = 2 * 1!</td>
</tr>
<tr>
<td>3! = 3 * 2!</td>
</tr>
<tr>
<td>4! = 4 * 3!</td>
</tr>
<tr>
<td>5! = 5 * 4!</td>
</tr>
</tbody>
</table>
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0] 1! = 1</td>
<td></td>
</tr>
<tr>
<td>Factorial(1)</td>
<td>[1] 2! = 2 * 1!</td>
<td></td>
</tr>
<tr>
<td>Factorial(2)</td>
<td>[2] 3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td>[4] 5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>

Maxwell J Dunne
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>Factorial(1)</td>
<td>2! = 2 * 1!</td>
<td></td>
</tr>
<tr>
<td>Factorial(2)</td>
<td>3! = 3 * 2!</td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td>4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td>5! = 5 * 4!</td>
<td></td>
</tr>
<tr>
<td>Factorial(5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursion
Evaluation of Recursive Functions

Evaluation of 5!
(based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] 1! = 1</td>
<td>1</td>
<td>= 1</td>
</tr>
<tr>
<td>[1] 2! = 2 * 1!</td>
<td>2</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>[2] 3! = 3 * 2!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3] 4! = 4 * 3!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4] 5! = 5 * 4!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] Factorial(3)</td>
<td>1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td>[1] Factorial(4)</td>
<td>2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>[3] Factorial(6)</td>
<td>4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>[4] Factorial(7)</td>
<td>5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>
Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td></td>
<td>3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td></td>
<td>4! = 4 * 3!</td>
<td>= 4 * 6 = 24</td>
</tr>
<tr>
<td></td>
<td>5! = 5 * 4!</td>
<td></td>
</tr>
</tbody>
</table>

Maxwell J Dunne
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

**Stack** (top)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] 1! = 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[1] 2! = 2 * 1!</td>
<td>2 * 1 = 2</td>
<td>2 * 1 = 2</td>
</tr>
<tr>
<td>[2] 3! = 3 * 2!</td>
<td>3 * 2 = 6</td>
<td>3 * 2 = 6</td>
</tr>
<tr>
<td>[4] 5! = 5 * 4!</td>
<td>5 * 24 = 120</td>
<td>5 * 24 = 120</td>
</tr>
</tbody>
</table>
Recursion

Summary

• Usable for solving problems that are divided into subproblems
  – Divide and conquer
• Initial conditions must be similar to conditions for any of the subproblems
  – No difference between solving the smaller computation stand-alone versus as part of a larger computation
• Requires well-defined termination condition
Recursion

Caveats

• Problem must have a well-defined termination condition/base case
• Must have enough memory
  – Memory use high from filling the function stack
Recursion

Limitations

- Limited stack space

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial(3)</td>
</tr>
<tr>
<td>Factorial(4)</td>
</tr>
<tr>
<td>Factorial(5)</td>
</tr>
<tr>
<td>Factorial(6)</td>
</tr>
<tr>
<td>Factorial(7)</td>
</tr>
<tr>
<td>Factorial(8)</td>
</tr>
<tr>
<td>Factorial(9)</td>
</tr>
<tr>
<td>Factorial(10)</td>
</tr>
</tbody>
</table>
Recursion
Multiple recursion

• Recursion is not limited to a single function call

Example

```c
int Fibonacci(int n)
{
    if (n <= 1) {
        return 1;
    }
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```
Recursion
Evaluation of Recursive Functions

• Evaluation of \( F_4 \)
(based on code from previous slide)

Stack (top)

Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_4 = F_3 + F_2 \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack

Function call replaced with result

\[ F_3 = F_2 + F_1 \]

\[ F_4 = F_3 + F_2 \]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Fibonacci(2)</strong></td>
</tr>
<tr>
<td><strong>Fibonacci(3)</strong></td>
</tr>
<tr>
<td><strong>Fibonacci(4)</strong></td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

\[
F_2 = F_1 + F_0 \\
F_3 = F_2 + F_1 \\
F_4 = F_3 + F_2
\]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_1 = 1</td>
<td></td>
</tr>
<tr>
<td>F_2 = F_1 + F_0</td>
<td></td>
</tr>
<tr>
<td>F_3 = F_2 + F_1</td>
<td></td>
</tr>
<tr>
<td>F_4 = F_3 + F_2</td>
<td></td>
</tr>
</tbody>
</table>

Maxwell J Dunne
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_1 = 1</td>
<td></td>
</tr>
<tr>
<td>F_2 = F_1 + F_0</td>
<td></td>
</tr>
<tr>
<td>F_3 = F_2 + F_1</td>
<td></td>
</tr>
<tr>
<td>F_4 = F_3 + F_2</td>
<td></td>
</tr>
</tbody>
</table>

Maxwell J Dunne
CMPE-013/L: “C” Programming
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

- Partial results pushed on stack
- Function call replaced with result

\[
F_2 = 1 + F_0 \\
F_3 = F_2 + F_1 \\
F_4 = F_3 + F_2
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Fibonacci(0)
Fibonacci(2)
Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[
\begin{align*}
F_0 &= 1 \\
F_2 &= 1 + F_0 \\
F_3 &= F_2 + F_1 \\
F_4 &= F_3 + F_2 \\
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fibonacci(0)</td>
</tr>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(3)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

**Partial results pushed on stack**

- $F_0 = 1$
- $F_2 = 1 + F_0$
- $F_3 = F_2 + F_1$
- $F_4 = F_3 + F_2$

**Function call replaced with result**
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci(2)</td>
<td></td>
</tr>
<tr>
<td>Fibonacci(3)</td>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</td>
<td></td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

\[
F_2 = 1 + 1
\]

\[
F_3 = F_2 + F_1
\]

\[
F_4 = F_3 + F_2
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Partial results pushed on stack
Function call replaced with result

\[
F_3 = 2 + F_1
\]
\[
F_4 = F_3 + F_2
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Fibonacci(1)
Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[
\begin{align*}
F_1 &= 1 \\
F_3 &= 2 + F_1 \\
F_4 &= F_3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

| Partial results pushed on stack |
| Function call replaced with result |

Fibonacci(3)
Fibonacci(4)

F₃ = 2 + 1
F₄ = F₃ + F₂
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Fibonacci(4)</th>
</tr>
</thead>
</table>

Partial results pushed on stack
Function call replaced with result

$F_4 = 3 + F_2$
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Fibonacci(2)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[
F_2 = F_1 + F_0
\]

\[
F_4 = 3 + F_2
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Partial results pushed on stack
Function call replaced with result

Fibonacci(1)
Fibonacci(2)
Fibonacci(4)

F\_1 = 1
F\_2 = F\_1 + F\_0
F\_4 = 3 + F\_2
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

**Stack** (top)

<table>
<thead>
<tr>
<th>Partial results pushed on stack</th>
<th>Function call replaced with result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(2)</td>
<td></td>
</tr>
<tr>
<td>Fibonacci(4)</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
F_2 &= 1 + F_0 \\
F_4 &= 3 + F_2
\end{align*}
\]
Recursion
Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<table>
<thead>
<tr>
<th>Fibonacci(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(2)</td>
</tr>
<tr>
<td>Fibonacci(4)</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

\[
F_0 = 1 \\
F_2 = 1 + F_0 \\
F_4 = 3 + F_2
\]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Fibonacci(2)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[ F_2 = 1 + 1 \]
\[ F_4 = 3 + F_2 \]
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
(based on code from previous slide)

Stack (top)

Partial results pushed on stack

Function call replaced with result

F_{4} = 3 + 2
Recursion
Evaluation of Recursive Functions

• Evaluation of 5!
  (based on code from previous slide)

Stack (top)

Partial results pushed on stack

Function call replaced with result

\[ F_4 = 5 \]
Recursion
Self-similarity

• A structure that is similar to part of itself
  – Example: fractals

• Computation and data must be self-similar for recursion

• Previous examples only dealt with single integers

• But what about more complicated data?
Recursion
Complex data

- For example, operating on a string
  - How to do that in C?
- Passing a single string through functions is trivial.
- But what about splitting the string up?
Recursion
Complex data example

- Recursive word count
  - Count 1 word per function call

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

str

```
This is the end.
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

```
This is the end.
1
```
Recursion

Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

---

This is the end.

1
Recursion
Complex data example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

str
This is the end.
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

```
This is the end.
1 1
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

![Diagram showing recursion process](image)
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

```
This is the end.
```

```
1 1
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

1 1 1
Recursion
Complex data example

```
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

Example:
```
This is the end.
```

```
1 1 1
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

```
This is the end.
```

```
1 1 1
```
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This   is   the   end.

1      1      2
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

| 1 | 3 |
Recursion
Complex data example

Example

```c
int WordCount(char *str)
{
    char *p = strchr(str, ' ');
    if (!p) {
        return 0;
    }
    return 1 + WordCount(p + 1);
}
```

This is the end.

4
Binary trees
Binary trees

- ADT where nodes:
  - Have 0, 1, or 2 children
  - Have a single parent
- Generally nodes only know their children
- Generally traversal is top-down
  - From parents to children
Binary trees

- We will only discuss full binary trees
- Size is $2^n - 1$
Binary trees

Traversal

- Binary trees store data at each node
- So the tree must be traversed to access the node that has the data we want
Binary trees
Self-similarity
Binary trees
Self-similarity
Binary trees
Self-similarity
Binary trees

Self-similarity
Binary trees
Serialization

- A linear representation of a tree
Binary trees

Serialization

<table>
<thead>
<tr>
<th>Root</th>
<th>Left subtree</th>
<th>Right subtree</th>
</tr>
</thead>
</table>

Diagram of a binary tree with arrows pointing to the root node and subdivisions into left and right subtrees.
Binary trees
Serialization

<table>
<thead>
<tr>
<th>Root</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
</table>

Diagram of a binary tree with arrows indicating the serialization process.
Binary trees

Serialization

• Root node at the 1\textsuperscript{st} element
• Left tree at the 2\textsuperscript{nd} element
• Right tree at 2^{n-1} element
Binary trees
Serialization

A B C D E F G

Diagram of a binary tree with nodes A, B, C, D, E, F, G.
Binary trees
Serialization

A B C D
Binary trees

Serialization

A B C
Binary trees
Serialization
Binary trees

Serialization

A   B   C   D
Binary trees

Serialization

A B C D E F G
Binary trees

Serialization

A  B  C  D  E  F
Binary trees
Serialization
Binary trees

Serialization
Binary trees
Serialization

A B C D E F G
Binary trees
Serialization

```
|   |   |   |   |   |   |   |
```

```
| 8 | 2 | 7 | 0 | 4 | 2 | 5 |
```

Diagram of a binary tree with nodes labeled 0 to 8.
CMPE-013/L

Toaster Oven Lab

Maxwell James Dunne
**Integer Timing**

Free running counters and precision

\[
\text{time: 20 seconds; } \rightarrow \text{ 40 ticks} \quad \uparrow \quad > \geq 1
\]

\[
\text{event 1} \quad \text{Start time} = \text{FRT}
\]

\[
\text{event 2} \quad \text{End time} = \text{FRT}
\]

\[
\text{duration} = \text{End time} - \text{Start time}
\]

Input capture
OLED Display
Formatting and Update Cycles

store
print update

store
print update

mode: bake
mode: broil
mode: %s
state
mode
time
oven on/off