CMPE-013/L

Introduction to “C” Programming

Maxwell J Dunne
Recursion
Recursion

- 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)

Stack (top)

Partial results pushed on stack

Factorial term replaced with result

\[ 5! = 5 \times 4! \]
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

Factorial(4)

[3] 4! = 4 * 3!

Factorial(5)

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

<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>Factorial(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorial(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorial(4)</td>
<td></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>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>
</tr>
<tr>
<td>[1] 2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>[2] 3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td>[3] 4! = 4 * 3!</td>
<td></td>
</tr>
<tr>
<td>[4] 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>[0] 1! = 1</td>
<td>= 1</td>
</tr>
<tr>
<td></td>
<td>[1] 2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td></td>
<td>[2] 3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td></td>
<td>[4] 5! = 5 * 4!</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)

<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>[0]</td>
<td>2! = 2 * 1!</td>
<td>= 2 * 1 = 2</td>
</tr>
<tr>
<td>[1]</td>
<td>3! = 3 * 2!</td>
<td>= 3 * 2 = 6</td>
</tr>
<tr>
<td>[3]</td>
<td>5! = 5 * 4!</td>
<td>= 5 * 24 = 120</td>
</tr>
</tbody>
</table>

Maxwell J Dunne
CMPE-013/L: “C” Programming
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 $\text{Fibonacci}(4)$
  (based on code from previous slide)

$$F_4 = F_3 + F_2$$
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 = F_2 + F_1
F_4 = F_3 + F_2

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>
</tr>
</thead>
<tbody>
<tr>
<td>Function call replaced with result</td>
</tr>
</tbody>
</table>

F_2 = F_1 + F_0
F_3 = F_2 + F_1
F_4 = F_3 + F_2

Fibonacci(2)
Fibonacci(3)
Fibonacci(4)
Recursion
Evaluation of Recursive Functions

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

Stack (top)

Fibonacci(1)
Fibonacci(2)
Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

F_1 = 1
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)

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

Fibonacci(1)
Fibonacci(2)
Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

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

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

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

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)

Partial results pushed on stack
Function call replaced with result

F₀ = 1
F₂ = F₀ + F₀
F₃ = F₂ + F₁
F₄ = F₃ + F₂

Fibonacci(0)
Fibonacci(2)
Fibonacci(3)
Fibonacci(4)
Recursion
Evaluation of Recursive Functions

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

Stack (top)

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<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
Function call replaced with result

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

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

Stack (top)

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

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)

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

Fibonacci(3)
Fibonacci(4)

Partial results
pushed on stack
Function call replaced
with result

F₃ = 2 + F₁
F₄ = F₃ + F₂
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)

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

Fibonacci(3)
Fibonacci(4)

Partial results pushed on stack
Function call replaced with result

\[
F_3 = 2 + 1
\]
\[
F_4 = F_3 + F_2
\]
Recursion
Evaluation of Recursive Functions

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

Stack (top)

Fibonacci(4)

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)

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)

<table>
<thead>
<tr>
<th>Stack (top)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F_1 = 1</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F_2 = F_1 + F_0</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F_3 = F_2 + F_1</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F_4 = 3 + F_2</td>
</tr>
</tbody>
</table>

Partial results pushed on stack
Function call replaced with result

Fibonacci(1)  Fibonacci(2)  Fibonacci(4)
Recursion
Evaluation of Recursive Functions

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

Stack (top)

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

Partial results pushed on stack
Function call replaced with result

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

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

Stack (top)

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

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)

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)
- Fibonacci(4)

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

<table>
<thead>
<tr>
<th>This</th>
<th>is</th>
<th>the</th>
<th>end.</th>
</tr>
</thead>
</table>
Recursion
Complex data example

Example

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

str  p
↓  ↓
This X 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

Example

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

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);
}
```

```
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);
}
```

str

<table>
<thead>
<tr>
<th>This</th>
<th>is</th>
<th>the</th>
<th>end.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
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 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 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 a root node and child nodes illustrating the serialization process.
Binary trees

Serialization

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

- Red arrow pointing to the Root node.
- Green arrow pointing to the Left child.
- Blue arrow pointing to the Right child.
Binary trees
Serialization

- Root node at the $1^{st}$ element
- Left tree at the $2^{nd}$ element
- Right tree at $2^{n-1}$ element
Binary trees

Serialization

A  B  C  D  E  F  G
Binary trees
Serialization

A B C D
Binary trees
Serialization

A   B   C

A
  
  B
  |
  C
  |
  |
  |
  |
  |
Binary trees

Serialization

A  B  C  D  

A -> B -> C -> D
Binary trees
Serialization

A B C D
Binary trees
Serialization

\[ 2^{n-1} \]
\[ 2^{3-1} \]
\[ 2^2 \]

A     B     C     D     E     F     G

\[ 3 \]
Binary trees
Serialization

A B C D E F

\[ 2^{n-1} \]
Binary trees
Serialization

A  B  C  D  E  F  G

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

A   B   C   D   E   F   G

A
  B   E
  C   D   F   G
Binary trees
Serialization

min heap
Binary trees

Serialization
CMPE-013/L

Bounce (or Hardware)

Maxwell James Dunne
Bounce

- Digital I/O
- A/D
- Timers
- Debouncing
Leds. h

\[ \text{no.} < \]
AID 1023
0 - 100 %
8 samples
Average
Buttons

00101010

up to down

down and Last event is not down
down event
debouncing $\frac{1}{T}$
4 buttons

1 button
copy and paste
Button CEC
no blocking
no whiles
no for

UNO32
764
999
7022
1342
0 - 1023
Sprintf
BE and IUP

Arrot Hist[Buttons DP]
One button
copy
paste
buttons <

buttons

Leds.h