1 tree-search analysis:

1.1 Proof

2 Lexical scope, first look:

3 Enumerate:

3.1 Example: permutations (orderings)

3.2 permutations recursive diagram and I/O spec

3.3 insert-everywhere recursive diagram

3.4 Procedure definitions

4 OCaml

1 tree-search analysis:

- We can see that the runtime of a search on a BST is $O(n)$ (linear time) where $n$ is the depth of the tree.

- Now, let $g(n)$ be the worst-case runtime on a tree with depth $t$ and $n$ total items. We have the following:

$$
g(1) = 0$$

$$g(n) \leq 1 + g(n/2) \text{ for } n > 1$$

Solution: $g(n) \leq \log_2 n$

1.1 Proof

We can now prove this using induction:

Claim: $g(n) \leq \log_2 n$

Proof: By induction $n$
Base case: \( n = 1 \)

\[ g(1) = 0 \leq 0 = \log_2 1 \]

**Inductive step:** For \( n > 1 \):

\[ g(n) \leq 1 + g\left(\left\lceil \frac{n-1}{2} \right\rceil\right) \]

By inductive hypothesis we know that \( g\left(\left\lceil \frac{n-1}{2} \right\rceil\right) \leq \log_2 \left(\left\lceil \frac{n-1}{2} \right\rceil\right) \)

Now, we have the following:

\[
g(n) \leq 1 + g\left(\left\lceil \frac{n-1}{2} \right\rceil\right) \leq 1 + \log_2 \frac{n}{2} = 1 + \log_2 n - 1 \quad (1) \\
= \log_2 n \quad (2) \\
\]

QED

**Summary:** searching a BST with \( n \) items takes \( O(\log n) \) time Think of \( \log(n) \) as smaller than linear, but larger than constant time.

## 2 Lexical scope, first look:

- The body of a lambda expression can also be used as a top-level variable as we will see in the below function `increment`:

  ```scheme
  > (define increment 7)
  > ((lambda (x) (+ x increment)) 10 )
  17 
  > (define add-increment
  1    (lambda (x) (+ x increment)))
  2  > (add-increment 10)
  17
  ```

- Now, we can see how a lambda expression can be used as a formal argument:

  ```scheme
  > (define add-to-all
  1    (lambda (increment L)
  2      (map (lambda (x) (+ x increment)) L)))
  > (add-to-all 7 '(10 20 30))
  2  (17 27 37)
  ```

- The above refers back to the original procedure of `add-to-all` but still works.
• We say that the variables available are those in lexical scope when the lambda expression is evaluated.

• One procedure can return a second procedure whose body uses as a variable a formal argument of a first procedure:

```scheme
> (define adder
  (lambda (increment)
    (lambda (x) (+ x increment))))

> (map (adder 7) '(10 20 30))
(17 27 37)
```

3 Enumerate:

• Generating a set of all combinatorial objects of a certain kind.

3.1 Example: permutations (orderings)

• Consider the list (a, b, c)

• Re-ordering these elements, we have all of the below options:

  (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)

• We can see that the total number of permutations (including the original) is 3!

• We will write a procedure permutations to generator a list of all permutations of a given list of data objects.

3.2 permutations recursive diagram and I/O spec

I/O spec:

```scheme
(permutations '(a b c))
(((a b c) (a c b) (b a c) (b c a) (c a b) (c b a)))
```

Recursion diagram:

Original input: (a b c)
  Recursive input: (b c)
  Recursive output: ((b c) (c b))
Original output:((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))

• In this diagram, we see that we need another procedure to get from out recursive input to our recursive output. We will call this insert-everywhere.
3.3 **insert-everywhere recursive diagram**

original input: 1, (a, b, c)
recursive input: 1, (b, c)
recursive output: ((1, b, c), (b, 1, c), (b, c, 1))
original output: ((1, b, c, a), (a, 1, b, c), (a, b, 1, c), (a, b, c, 1))

3.4 **Procedure definitions**

```scheme
(define insert-everywhere
  (lambda (x L)
    (cond
      ((empty? L) empty)
      ((cons? L)
        (cons (cons x L)
          (map (lambda (perm) (cons (car L) perm))
              (insert-everywhere x (cdr L))))))))

(define all-insertions
  (lambda (x L)
    (my-flatten
      (map (lambda (perm) (insert-everywhere x perm))
           L))))
```

Now, we can implement the strategy from earlier:

```scheme
(define permutations
  (lambda (L)
    ...
    (all-insertions (car L) (permutations (cdr L)) ... )))
```

4 **OCaml**

OCaml is a typed language. To find out what this means and more, come to Friday’s lecture! Additionally, workshops held Wednesday, October 17 at 6pm and Saturday, October 20 at 4pm will go over basic OCaml concepts.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)