Lecture #34: Review of Scheme

List Tails

- The full Scheme library includes the function `list-tail`, which essentially performs the `cdr` operation a *given* number of times:

```
scm> (list-tail '(a b c d) 0)
(a b c d)
scm> (list-tail '(a b c d) 1)
(b c d)
scm> (list-tail '(a b c d) 2)
(c d)
scm> (list-tail '(a b c d) 4)
()
```

- Can you implement it?

```
(define (list-tail lst k)
)
```

- Solutions to problems in this lecture are in `34.scm`.

List Elements by Index

- Another Standard Scheme library operation is `list-ref`, which works like `lst[k]` in Python:

```
scm> (list-ref '(a b c d) 0)
a
scm> (list-ref '(a b c d) 3)
d
```

- What's the simplest implementation you can come up with?

```
(define (list-ref lst k)

)
```

A Faster List?

- Linked lists are problematic if your algorithm calls for performing lots of `list-ref` operations.
- In Python, indexed list access (like `A[i]`) takes constant ($\Theta(1)$) time.
- What about `list-ref`?

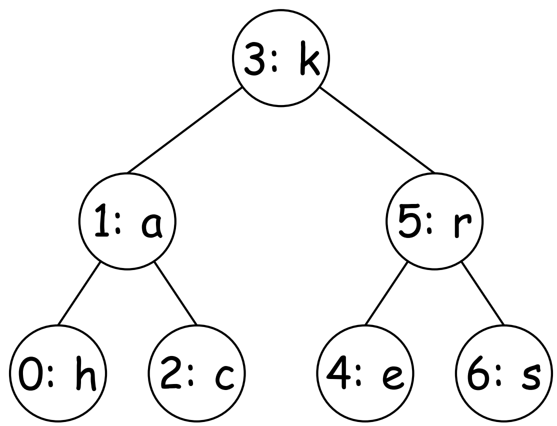
A Faster List?

- Linked lists are problematic if your algorithm calls for performing lots of `list-ref` operations.
- In Python, indexed list access (like `A[i]`) takes constant ($\Theta(1)$) time.
- What about `list-ref`?

  It takes $\Theta(N)$ time (worst case) for a list of length N.
- While we can't get to $\Theta(1)$ with a linked list, we can do better than $\Theta(N)$.

Lists as Trees

- One idea is to represent a list as a *binary search tree*.
- The labels of this tree contain integer indices (0–N) and the values at those indices.
- For example, `(h a c k e r s)` could be



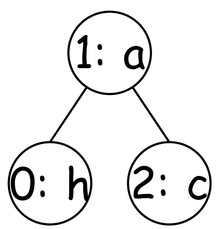
- So for any node in this tree, all the nodes in its left subtree have smaller indices, and all the nodes in the right subtree have larger indices.

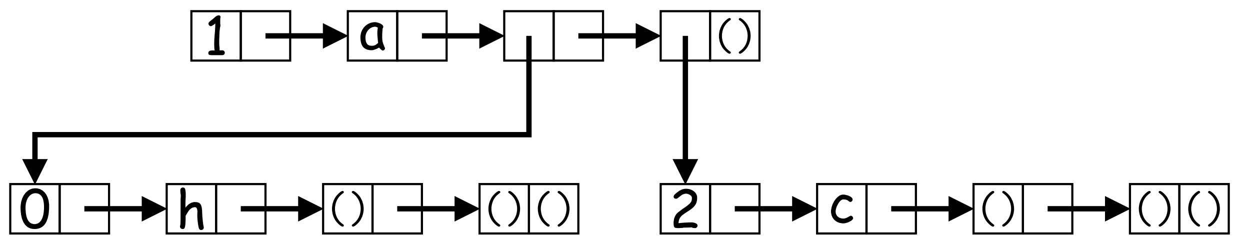
Lists as Trees as Lists

- First, let's define a suitable data structure to represent these trees. Suggestions?

Lists as Trees as Lists

- First, let's define a suitable data structure to represent these trees.
- Each node can be represented as a list containing the index, value, and two children, with () representing an empty tree (as well as an empty list).

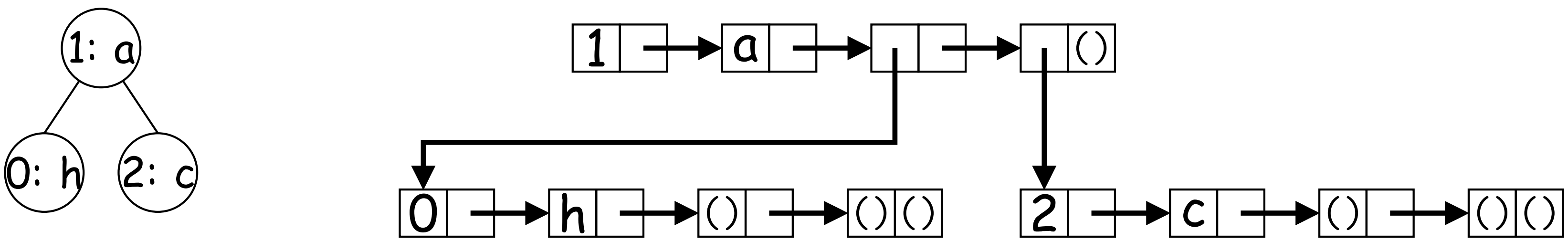




- What's a reasonable set of functions for accessing or creating these "array trees"?

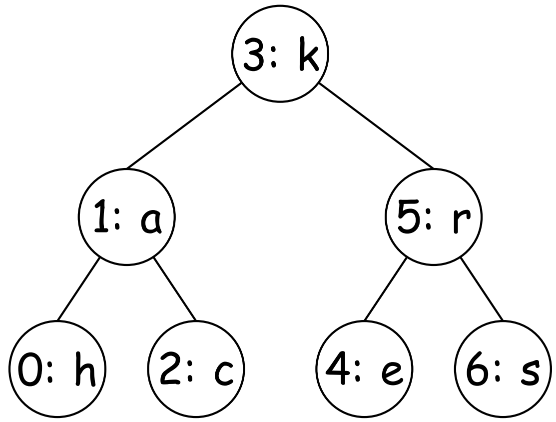
Lists as Trees as Lists

- First, let's define a suitable data structure to represent these trees.
- Each node can be represented as a list containing the index, value, and two children, with () representing an empty tree (as well as an empty list).



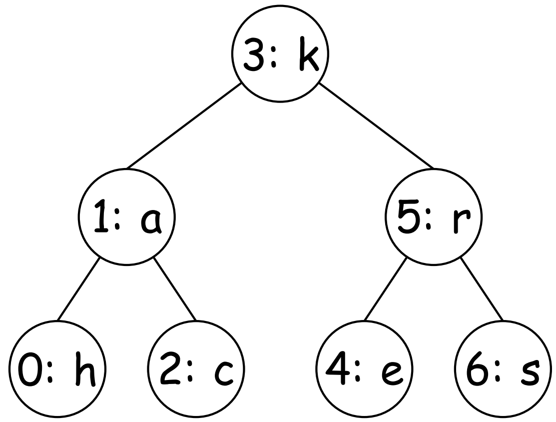

```
(define (arr-index arr) (car arr))
(define (arr-value arr) (car (cdr arr)))
(define (arr-left arr) (car (cdr (cdr arr))))
(define (arr-right arr) (car (cdr (cdr (cdr arr)))))
(define (arr-make index value left right)
  (list index value left right))
(define arr-empty nil)
```

Fetching from an Array Tree



- To fetch item #j, we start at the root and compare j to the index we find there. If equal, we return the value in that node.
- If j is less than the node's index, we search for item j in the left subtree. Otherwise, we search the right.
- How long does this process take if there are N elements in the represented array?

Fetching from an Array Tree

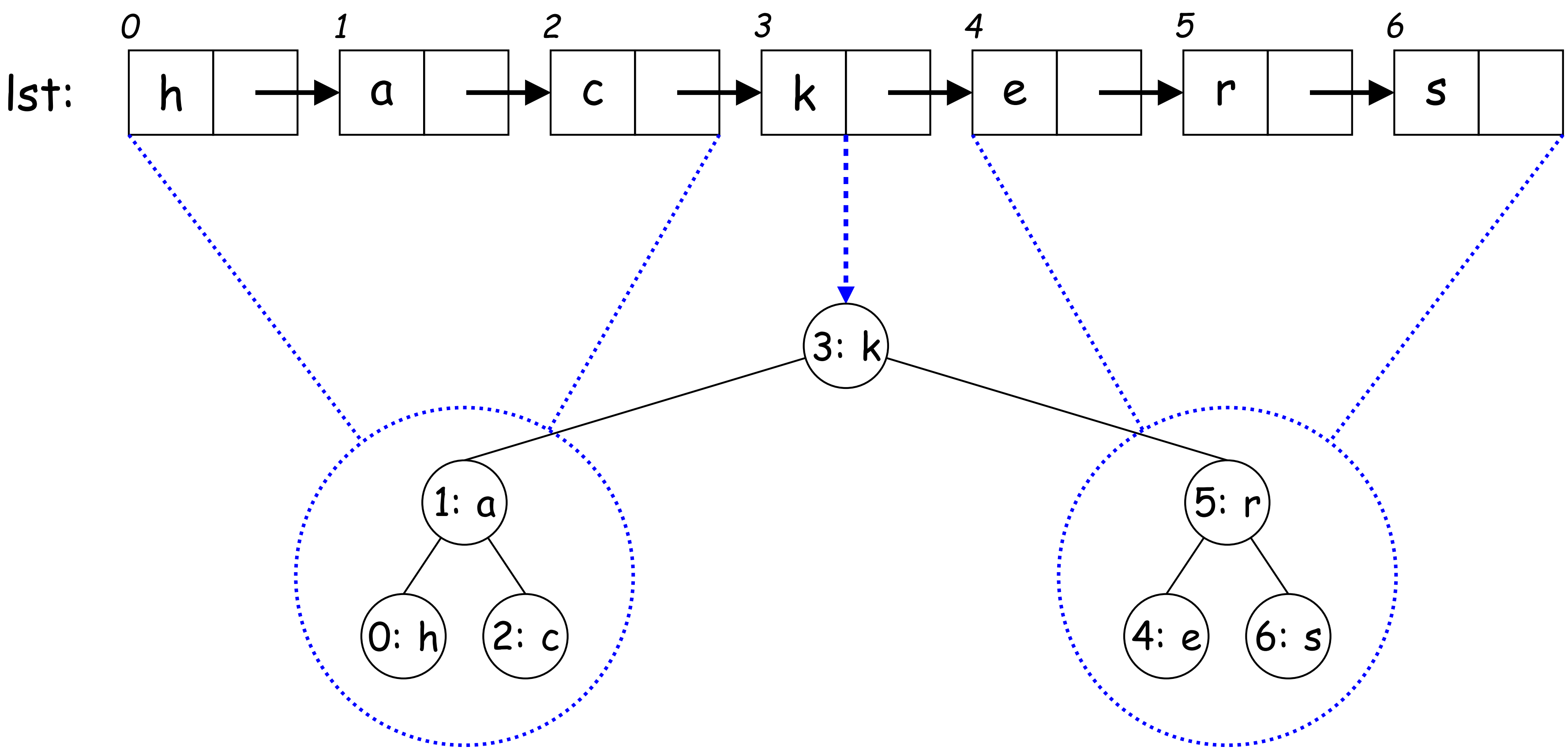


- To fetch item #j, we start at the root and compare j to the index we find there. If equal, we return the value in that node.
- If j is less than the node's index, we search for item j in the left subtree. Otherwise, we search the right.
- How long does this process take if there are N elements in the represented array?

  $\Theta(\lg N)$ (assuming that the tree is evenly balanced.)
- How would we code it in Scheme?

Building an Array Tree

- Given an arbitrary Scheme list, what is a function that will return an array tree such as we've been discussing?
- Suggested approach: find list's length, then build tree from middle element of list, with subtrees found recursively on either side. How do we do this in Scheme?



Further Encapsulation

- Now that we have a function for creating this data structure, how do we use it to get the following effects?

```
scm> (define arr (make-array-tree '(h a c k e r s)))
arr
scm> (arr 0)
h
scm> (arr 4)
e
```

- In other words, create a function with a single (index) argument that returns the indexed item of the original list.

Let*

- Let's consider a macro.
- You implemented the `let` form, which should have the following behavior:

```
scm> (define x 4)
x
scm> (let ((x 5) (y x)) (list x y))
(5 4)
```

- That is, `x` in `(y x)` still refers to the outer `x`, not to its redefinition.
- There is another form in standard Scheme: `let*`, which is incremental:

```
scm> (let* ((x 5) (y x)) (list x y))
(5 5)
```

- ...as if we had written

```
scm> (let ((x 5))
...>    (let ((y x))
...>       (list x y)))
(5 5)
```

- Assuming that `let*` bodies have a single expression, how can we define `let*` to get this effect?