8: Trees

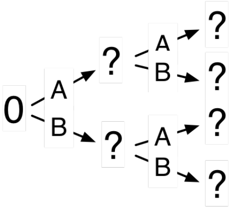
Lessons learned from unary and binary

Our unary representation of Nats was inefficient because we only had a choice of a single data constructor.

This conceptual diagram showed the relationships between different natural numbers in terms of the relationships between their representations.

$$0 \xrightarrow{S} 1 \xrightarrow{S} 2 \xrightarrow{S}$$

By using two data constructors instead of one, we could do better.



Using two data constructors allowed us to create small representations of many different natural numbers.

We'll adapt this idea to create a data structure holding many different values that allows quicker access to each of them.

Inefficiencies of lists

A list exhibits a linear structure similar to our conceptual diagram of our unary representation of Nats.

```
(cons 'A (cons 'B (cons 'C (cons 'D empty))))
```

Getting the value of the first element of a list is easy and quick.

A list exhibits a linear structure similar to our conceptual diagram of our unary representation of Nats.

```
(cons 'A (cons 'B (cons 'C (cons 'D empty))))
```

Getting the value of the first element of a list is easy and quick.

Getting the value of the last element of a list takes
more code and more time.

A list exhibits a linear structure similar to our conceptual diagram of our unary representation of Nats.

```
(cons 'A (cons 'B (cons 'C (cons 'D empty))))
```

Getting the value of the first element of a list is easy and quick.

Getting the value of the last element of a list takes
more code and more time.

To be clear about what we are looking for, let's return to the mathematical concept of sequences, as discussed in lecture module 05.

Our implementations of sequences in lecture module 05 (S-lists and Racket's built-in lists) resulted in implementations of the following operations:

- **extend** consumed an element e and a sequence S and produced the new sequence e, S.
- **head** consumed a non-empty sequence S and produced the element e, where $S = e, S'$.
- **tail** consumed a non-empty sequence S and produced the sequence S', where $S = e, S'$.

Our implementations of sequences in lecture module 05 (S-lists and Racket's built-in lists) resulted in implementations of the following operations:

- **extend** consumed an element e and a sequence S and produced the new sequence e, S.
- **head** consumed a non-empty sequence S and produced the element e, where $S = e, S'$.
- **tail** consumed a non-empty sequence S and produced the sequence S', where $S = e, S'$.

For Racket lists, the implementations of these operations are `cons`, `first`, and `rest`.

We now add a new operation, **index**, which consumes a natural number i and a sequence S and produces the i^{th} element of S (counting from zero).

We now add a new operation, **index**, which consumes a natural number i and a sequence S and produces the i^{th} element of S (counting from zero).

We say this element has index i in S.
S must have length greater than i.

The index operation

We now add a new operation, **index**, which consumes a natural number i and a sequence S and produces the i^{th} element of S (counting from zero).

We say this element has index i in S.
S must have length greater than i.

Here is an implementation for Racket lists.

```
(define (list-ref i lst)
  (cond
    [(empty? lst) (error "no element of that index")]
    [(zero? i) (first lst)]
    [else (list-ref (sub1 i) (rest lst))]))
```

We now add a new operation, **index**, which consumes a natural number i and a sequence S and produces the i^{th} element of S (counting from zero).

We say this element has index i in S.
S must have length greater than i.

Here is an implementation for Racket lists.

```
(define (list-ref i lst)
  (cond
    [(empty? lst) (error "no element of that index")]
    [(zero? i) (first lst)]
    [else (list-ref (sub1 i) (rest lst))]))
```

Example: `(list-ref 2 '(A B C D))` is `'C`.

We now add a new operation, **index**, which consumes a natural number i and a sequence S and produces the i^{th} element of S (counting from zero).

We say this element has index i in S.
S must have length greater than i.

Here is an implementation for Racket lists.

```
(define (list-ref i lst)
  (cond
    [(empty? lst) (error "no element of that index")]
    [(zero? i) (first lst)]
    [else (list-ref (sub1 i) (rest lst))]))
```

Example: `(list-ref 2 '(A B C D))` is `'C`.

`list-ref` is a built-in function in ISL+.

Our code for `list-ref` does structural recursion on the natural-number index and the list representing the sequence.

Our code for `list-ref` does structural recursion on the natural-number index and the list representing the sequence.

The total number of recursive applications of `list-ref` is the minimum of the index and the length of the list.

Efficiency of the index operation

Our code for `list-ref` does structural recursion on the natural-number index and the list representing the sequence.

The total number of recursive applications of `list-ref` is the minimum of the index and the length of the list.

Can we do better than this?

Efficiency of the index operation

Our code for `list-ref` does structural recursion on the natural-number index and the list representing the sequence.

The total number of recursive applications of `list-ref` is the minimum of the index and the length of the list.

Can we do better than this?

Our goal is to create a representation of sequences that supports the four operations (extend, head, tail, index) but improves the running time for index.

Our code for `list-ref` does structural recursion on the natural-number index and the list representing the sequence.

The total number of recursive applications of `list-ref` is the minimum of the index and the length of the list.

Can we do better than this?

Our goal is to create a representation of sequences that supports the four operations (extend, head, tail, index) but improves the running time for index.

We'll take as a starting point our development of S-lists from lecture module 05, which was made obsolete by Racket's built-in list operations.

```
(define-struct Empty ())
(define-struct Cons (fst rst))
```

Data definition: an S-list is either a `(make-Empty)` or it is `(make-Cons v slist)`, where `v` is a Racket value and `slist` is an S-list.

```
(define-struct Empty ())
(define-struct Cons (fst rst))
```

Data definition: an S-list is either a `(make-Empty)` or it is `(make-Cons v slist)`, where `v` is a Racket value and `slist` is an S-list.

```
(define-struct Empty ())
(define-struct Node (val left right))
```

Data definition: a **binary tree** is either a `(make-Empty)` or it is `(make-Node v lft rgt)`, where `v` is a Racket value and `lft` and `rgt` are binary trees.

```
(define-struct Empty ())
(define-struct Cons (fst rst))
```

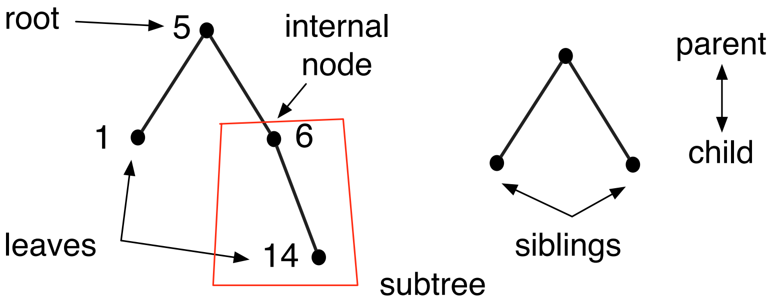
Data definition: an S-list is either a `(make-Empty)` or it is `(make-Cons v slist)`, where `v` is a Racket value and `slist` is an S-list.

```
(define-struct Empty ())
(define-struct Node (val left right))
```

Data definition: a **binary tree** is either a `(make-Empty)` or it is `(make-Node v lft rgt)`, where `v` is a Racket value and `lft` and `rgt` are binary trees.

Examples:

```
(make-Node 14 (make-Empty) (make-Empty))
(make-Node 6
           (make-Empty)
           (make-Node 14 (make-Empty) (make-Empty)))
```


root
5
internal
node
1
6
leaves
14
subtree
parent
child
siblings

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree.**

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree**.

The element of index zero will be at the root of the tree.

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree**.

The element of index zero will be at the root of the tree.

Example: representing the sequence A,B,C,D,E,F.

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree**.

The element of index zero will be at the root of the tree.

Example: representing the sequence A,B,C,D,E,F.

Our representation will be `(make-Node 'A ... ...)`.

Representing sequences using Braun trees

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree**.

The element of index zero will be at the root of the tree.

Example: representing the sequence A,B,C,D,E,F.

Our representation will be `(make-Node 'A ... ...)`.

We will distribute the rest of the sequence among the left and right subtrees of the root.

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree**.

The element of index zero will be at the root of the tree.

Example: representing the sequence A,B,C,D,E,F.

Our representation will be `(make-Node 'A ... ...)`.

We will distribute the rest of the sequence among the left and right subtrees of the root.

The elements whose indices are odd numbers will go in the left subtree. These are B, D, and F.

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree**.

The element of index zero will be at the root of the tree.

Example: representing the sequence A,B,C,D,E,F.

Our representation will be `(make-Node 'A ... ...)`.

We will distribute the rest of the sequence among the left and right subtrees of the root.

The elements whose indices are odd numbers will go in the left subtree. These are B, D, and F.

The elements whose indices are even numbers (but not zero) will go in the right subtree. These are C and E.

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree**.

The element of index zero will be at the root of the tree.

Example: representing the sequence A,B,C,D,E,F.

Our representation will be `(make-Node 'A ... ...)`.

We will distribute the rest of the sequence among the left and right subtrees of the root.

The elements whose indices are odd numbers will go in the left subtree. These are B, D, and F.

The elements whose indices are even numbers (but not zero) will go in the right subtree. These are C and E.

We will store them in the subtree in the order in which they appear in the odd- or even-indexed subsequence, recursively.

We are going to place the elements of a sequence in a binary tree in a specific manner that results in a **Braun tree**.

The element of index zero will be at the root of the tree.

Example: representing the sequence A,B,C,D,E,F.

Our representation will be `(make-Node 'A ... ...)`.

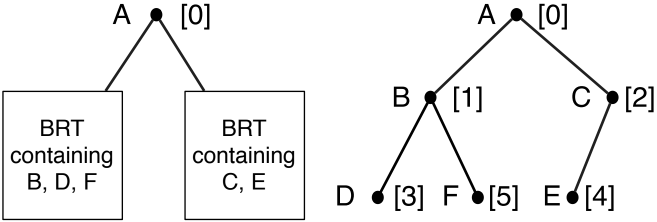
We will distribute the rest of the sequence among the left and right subtrees of the root.

The elements whose indices are odd numbers will go in the left subtree. These are B, D, and F.

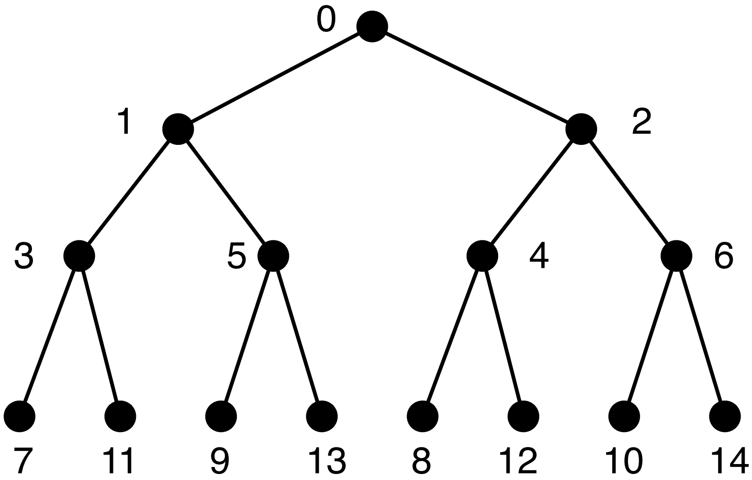
The elements whose indices are even numbers (but not zero) will go in the right subtree. These are C and E.

We will store them in the subtree in the order in which they appear in the odd- or even-indexed subsequence, recursively.

That is: the left and right subtrees will also be Braun trees.




```
(make-Node 'A
  (make-Node 'B
    (make-Node 'D (make-Empty) (make-Empty))
    (make-Node 'F (make-Empty) (make-Empty)))
  (make-Node 'C
    (make-Node 'E (make-Empty) (make-Empty))
    (make-Empty)))
```


0
1
2
3
5
4
6
7
11
9
13
8
12
10
14

The index operation in a Braun tree

How do we find the element of index i?

How do we find the element of index i?

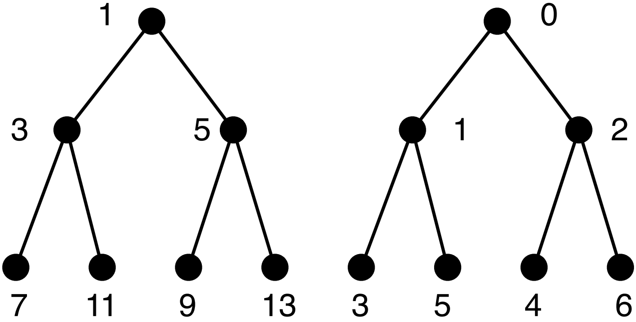
```
(define (brt-ref i brt)
  (cond
    [(Empty? brt) (error "index too large")]
    [(zero? i) (Node-val brt)]
    [(odd? i)  (brt-ref ... (Node-left brt))]
    [(even? i) (brt-ref ... (Node-right brt))]))
```

How do we find the element of index i?

```
(define (brt-ref i brt)
  (cond
    [(Empty? brt) (error "index too large")]
    [(zero? i) (Node-val brt)]
    [(odd? i)  (brt-ref ... (Node-left brt))]
    [(even? i) (brt-ref ... (Node-right brt))]))
```

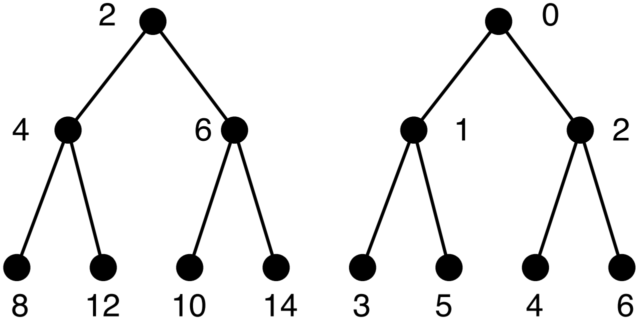
We must compute, from the index of the desired element in the whole sequence, its index in the odd-index or even-index subsequences that are stored in the left or right subtree respectively.

Here is the left subtree.



The element at index i in the whole sequence (for odd i) is found at index $(i-1)/2$ in the odd-indexed sequence (stored in the left subtree).

Here is the right subtree.



The element at index i in the whole sequence (for nonzero even i) is found at index $(i/2) - 1$ in the even-indexed sequence (stored in the right subtree).

The index operation in a Braun tree, completed

```
(define (brt-ref i brt)
  (cond
    [(Empty? brt) (error "index too large")]
    [(zero? i) (Node-val brt)]
    [(odd? i)  (brt-ref (/ (sub1 i) 2) (Node-left brt))]
    [(even? i) (brt-ref (sub1 (/ i 2)) (Node-right brt))]))
```

The index operation in a Braun tree, completed

```
(define (brt-ref i brt)
  (cond
    [(Empty? brt) (error "index too large")]
    [(zero? i) (Node-val brt)]
    [(odd? i)  (brt-ref (/ (sub1 i) 2) (Node-left brt))]
    [(even? i) (brt-ref (sub1 (/ i 2)) (Node-right brt))]))
```

Let $T_1(i)$ be the number of applications of `brt-ref` on a Braun tree and a valid index `i`.

The index operation in a Braun tree, completed

```
(define (brt-ref i brt)
  (cond
    [(Empty? brt) (error "index too large")]
    [(zero? i) (Node-val brt)]
    [(odd? i)  (brt-ref (/ (sub1 i) 2) (Node-left brt))]
    [(even? i) (brt-ref (sub1 (/ i 2)) (Node-right brt))]))
```

Let $T_1(i)$ be the number of applications of `brt-ref` on a Braun tree and a valid index `i`.

$$
\begin{aligned}
T_1(0) &= 1 \\
T_1(2k+1) &= 1 + T_1(k) \\
T_1(2k+2) &= 1 + T_1(k)
\end{aligned}
$$

n	$T_1(n)$
0	1
1	2
2	2
3	3
4	3
5	3
6	3
7	4

Running time of brt-ref on a valid index

n	$T_1(n)$
0	1
1	2
2	2
3	3
4	3
5	3
6	3
7	4

Conjecture: For $2^k \leqslant i + 1 < 2^{k+1}$, $T_1(n) = k + 1$.

n	$T_1(n)$
0	1
1	2
2	2
3	3
4	3
5	3
6	3
7	4

Conjecture: For $2^k \leqslant i+1 < 2^{k+1}$, $T_1(n) = k+1$.

Or: For $k \leqslant \log_2(i+1) < k+1$, $T_1(n) = i+1$.

Running time of brt-ref on a valid index

n	$T_1(n)$
0	1
1	2
2	2
3	3
4	3
5	3
6	3
7	4

Conjecture: For $2^k \leqslant i + 1 < 2^{k+1}$, $T_1(n) = k + 1$.

Or: For $k \leqslant \log_2(i + 1) < k + 1$, $T_1(n) = i + 1$.

Or: For $i \geqslant 0$, $T_1(i) = \lfloor \log_2(i + 1) \rfloor + 1$.

Let $T_2(n)$ be the number of applications of `brt-ref` on a Braun tree representing a sequence of length n and an invalid index.

Let $T_2(n)$ be the number of applications of `brt-ref` on a Braun tree representing a sequence of length n and an invalid index.

$$\begin{aligned} T_2(0) &= 1 \\ T_2(2k+1) &= 1 + T_2(k) \\ T_2(2k+2) &= 1 + T_2(k) \end{aligned}$$

Running time of brt-ref, concluded

Let $T_2(n)$ be the number of applications of `brt-ref` on a Braun tree representing a sequence of length n and an invalid index.

$$\begin{aligned} T_2(0) &= 1 \\ T_2(2k+1) &= 1 + T_2(k) \\ T_2(2k+2) &= 1 + T_2(k) \end{aligned}$$

Clearly $T_2(n) = T_1(n)$.

Let $T_2(n)$ be the number of applications of `brt-ref` on a Braun tree representing a sequence of length n and an invalid index.

$$\begin{aligned} T_2(0) &= 1 \\ T_2(2k+1) &= 1 + T_2(k) \\ T_2(2k+2) &= 1 + T_2(k) \end{aligned}$$

Clearly $T_2(n) = T_1(n)$.

We conclude that the number of applications of `brt-ref` on a Braun tree representing a sequence of length n and an index i is $\lfloor \log_2(\min\{i+1, n+1\}) \rfloor + 1$.

Another way of looking at the running time on an invalid index is to note that each recursive application of `brt-ref` is done on only one subtree.

Another way of looking at the running time on an invalid index is to note that each recursive application of `brt-ref` is done on only one subtree.

We define the **height** of a Braun tree to be the maximum number of applications of `Node-left` or `Node-right` before reaching `(make-Empty)`.

Another way of looking at the running time on an invalid index is to note that each recursive application of `brt-ref` is done on only one subtree.

We define the **height** of a Braun tree to be the maximum number of applications of `Node-left` or `Node-right` before reaching `(make-Empty)`.

The total number of applications of `brt-ref` on a Braun tree is bounded above by the height plus 1.

Another way of looking at the running time on an invalid index is to note that each recursive application of `brt-ref` is done on only one subtree.

We define the **height** of a Braun tree to be the maximum number of applications of `Node-left` or `Node-right` before reaching `(make-Empty)`.

The total number of applications of `brt-ref` on a Braun tree is bounded above by the height plus 1.

A Braun tree of size n has height at most $\lfloor \log_2 n + 1 \rfloor$.

Another way of looking at the running time on an invalid index is to note that each recursive application of `brt-ref` is done on only one subtree.

We define the **height** of a Braun tree to be the maximum number of applications of `Node-left` or `Node-right` before reaching `(make-Empty)`.

The total number of applications of `brt-ref` on a Braun tree is bounded above by the height plus 1.

A Braun tree of size n has height at most $\lfloor \log_2 n + 1 \rfloor$.

Reasoning about height works because the recursion is done on only one subtree. For a task where recursion may be performed on both subtrees (for example, an "element of" computation), the number of recursive applications can be as big as the size.

What about the other operations?

We have been focussing on the index operation, and Braun trees perform quite well, but perhaps the other operations are much harder on a Braun tree.

What about the other operations?

We have been focussing on the index operation, and Braun trees perform quite well, but perhaps the other operations are much harder on a Braun tree.

As you probably guessed, this is not the case.

What about the other operations?

We have been focussing on the index operation, and Braun trees perform quite well, but perhaps the other operations are much harder on a Braun tree.

As you probably guessed, this is not the case.

The head operation is implemented by `Node-val` and so takes constant time.

What about the other operations?

We have been focussing on the index operation, and Braun trees perform quite well, but perhaps the other operations are much harder on a Braun tree.

As you probably guessed, this is not the case.

The head operation is implemented by `Node-val` and so takes constant time.

We can ensure running times for implementations of extend and tail that are comparable to the running time of index, that is, roughly logarithmic in the length of the sequence.

What about the other operations?

We have been focussing on the index operation, and Braun trees perform quite well, but perhaps the other operations are much harder on a Braun tree.

As you probably guessed, this is not the case.

The head operation is implemented by `Node-val` and so takes constant time.

We can ensure running times for implementations of extend and tail that are comparable to the running time of index, that is, roughly logarithmic in the length of the sequence.

This is not as good as with lists (where these operations take constant time) but it is still pretty good.

What about the other operations?

We have been focussing on the index operation, and Braun trees perform quite well, but perhaps the other operations are much harder on a Braun tree.

As you probably guessed, this is not the case.

The head operation is implemented by `Node-val` and so takes constant time.

We can ensure running times for implementations of extend and tail that are comparable to the running time of index, that is, roughly logarithmic in the length of the sequence.

This is not as good as with lists (where these operations take constant time) but it is still pretty good.

The key is to ensure that recursion is done on at most one subtree.

Suppose we are trying to extend A,B,C,D,E,F with Q, which goes at the front of the new sequence. Clearly Q is at the root of the new tree.

Implementing extend and tail

Suppose we are trying to extend A,B,C,D,E,F with Q, which goes at the front of the new sequence. Clearly Q is at the root of the new tree.

Everything in the old sequence has index one greater in the new sequence.

Suppose we are trying to extend A,B,C,D,E,F with Q, which goes at the front of the new sequence. Clearly Q is at the root of the new tree.

Everything in the old sequence has index one greater in the new sequence.

That means the old odd-indexed sequence is the new even-indexed subsequence. In other words, the new right subtree is the old left subtree, unchanged.

Suppose we are trying to extend A,B,C,D,E,F with Q, which goes at the front of the new sequence. Clearly Q is at the root of the new tree.

Everything in the old sequence has index one greater in the new sequence.

That means the old odd-indexed sequence is the new even-indexed subsequence. In other words, the new right subtree is the old left subtree, unchanged.

What is the new odd-indexed subsequence?

Implementing extend and tail

Suppose we are trying to extend A,B,C,D,E,F with Q, which goes at the front of the new sequence. Clearly Q is at the root of the new tree.

Everything in the old sequence has index one greater in the new sequence.

That means the old odd-indexed sequence is the new even-indexed subsequence. In other words, the new right subtree is the old left subtree, unchanged.

What is the new odd-indexed subsequence?

It is the old even-indexed subsequence with the former root value A extending it. This is a recursive application on the old right subtree.

Implementing extend and tail

Suppose we are trying to extend A,B,C,D,E,F with Q, which goes at the front of the new sequence. Clearly Q is at the root of the new tree.

Everything in the old sequence has index one greater in the new sequence.

That means the old odd-indexed sequence is the new even-indexed subsequence. In other words, the new right subtree is the old left subtree, unchanged.

What is the new odd-indexed subsequence?

It is the old even-indexed subsequence with the former root value A extending it. This is a recursive application on the old right subtree.

Similar reasoning works for the tail operation. The details are left as an exercise.

Braun trees are a specific type of binary tree that allow us to store a sequence in a manner that permits logarithmic-time index operations.

Braun trees in the larger context

Braun trees are a specific type of binary tree that allow us to store a sequence in a manner that permits logarithmic-time index operations.

We have sped up indexing relative to the list implementation, but for lists, the extend and tail operations take constant time, while they take logarithmic time for Braun trees.

Braun trees are a specific type of binary tree that allow us to store a sequence in a manner that permits logarithmic-time index operations.

We have sped up indexing relative to the list implementation, but for lists, the extend and tail operations take constant time, while they take logarithmic time for Braun trees.

There is a more clever (and more complicated) data structure that achieves constant time for extend and tail, while maintaining logarithmic time for index.

Braun trees are a specific type of binary tree that allow us to store a sequence in a manner that permits logarithmic-time index operations.

We have sped up indexing relative to the list implementation, but for lists, the extend and tail operations take constant time, while they take logarithmic time for Braun trees.

There is a more clever (and more complicated) data structure that achieves constant time for extend and tail, while maintaining logarithmic time for index.

Braun trees are an excellent introduction to the study of trees, but they are not used much in practice. Instead, more general data structures that implement “finite maps” are used.

Braun trees in the larger context

Braun trees are a specific type of binary tree that allow us to store a sequence in a manner that permits logarithmic-time index operations.

We have sped up indexing relative to the list implementation, but for lists, the extend and tail operations take constant time, while they take logarithmic time for Braun trees.

There is a more clever (and more complicated) data structure that achieves constant time for extend and tail, while maintaining logarithmic time for index.

Braun trees are an excellent introduction to the study of trees, but they are not used much in practice. Instead, more general data structures that implement “finite maps” are used.

Braun trees have logarithmic height, but it is more difficult or even impossible to ensure this for trees used for other purposes.

With Braun trees, we took a mathematical concept (sequences) that had no inherent tree structure, and imposed structure on it to improve efficiency.

With Braun trees, we took a mathematical concept (sequences) that had no inherent tree structure, and imposed structure on it to improve efficiency.

Many forms of data already have structure that lends itself naturally to a tree representation.

With Braun trees, we took a mathematical concept (sequences) that had no inherent tree structure, and imposed structure on it to improve efficiency.

Many forms of data already have structure that lends itself naturally to a tree representation.

A book may be divided into chapters, each chapter may be divided into sections and subsections, which contain paragraphs and words.

With Braun trees, we took a mathematical concept (sequences) that had no inherent tree structure, and imposed structure on it to improve efficiency.

Many forms of data already have structure that lends itself naturally to a tree representation.

A book may be divided into chapters, each chapter may be divided into sections and subsections, which contain paragraphs and words.

Web pages are specified in a language (HTML) which is tree-structured.

With Braun trees, we took a mathematical concept (sequences) that had no inherent tree structure, and imposed structure on it to improve efficiency.

Many forms of data already have structure that lends itself naturally to a tree representation.

A book may be divided into chapters, each chapter may be divided into sections and subsections, which contain paragraphs and words.

Web pages are specified in a language (HTML) which is tree-structured.

In fact, most languages created for processing by computers (programming or specification) are best handled by treating uses as trees.

With Braun trees, we took a mathematical concept (sequences) that had no inherent tree structure, and imposed structure on it to improve efficiency.

Many forms of data already have structure that lends itself naturally to a tree representation.

A book may be divided into chapters, each chapter may be divided into sections and subsections, which contain paragraphs and words.

Web pages are specified in a language (HTML) which is tree-structured.

In fact, most languages created for processing by computers (programming or specification) are best handled by treating uses as trees.

We will take a look at one line of development for which Racket is well-suited.

Consider a Racket expression using only numbers, and addition and multiplication with two arguments.

```
(* (+ 3 4) (+ 1 6))
```

Consider a Racket expression using only numbers, and addition and multiplication with two arguments.

```
(* (+ 3 4) (+ 1 6))
```

We can represent this by a variation on binary trees.

Consider a Racket expression using only numbers, and addition and multiplication with two arguments.

```
(* (+ 3 4) (+ 1 6))
```

We can represent this by a variation on binary trees.

The `val` field in a `Node` structure can hold the operation, and the `left` and `right` fields can hold the operands.

Consider a Racket expression using only numbers, and addition and multiplication with two arguments.

```
(* (+ 3 4) (+ 1 6))
```

We can represent this by a variation on binary trees.

The `val` field in a `Node` structure can hold the operation, and the `left` and `right` fields can hold the operands.

The `Empty` structure is replaced by a number.

Consider a Racket expression using only numbers, and addition and multiplication with two arguments.

```
(* (+ 3 4) (+ 1 6))
```

We can represent this by a variation on binary trees.

The `val` field in a `Node` structure can hold the operation, and the `left` and `right` fields can hold the operands.

The `Empty` structure is replaced by a number.

While this works, we can also use lists, which are made more convenient for this purpose because of quote notation.

```
'(* (+ 3 4) (+ 1 6))
```

```
(list '* (list '+ 3 4) (list '+ 1 6))
```

Data definition: a **binary arithmetic S-expression** (BAS-exp) is either a number or a `(list op lft rgt)`, where `op` is either `'+` or `'*`, and `lft` and `rgt` are BAS-exps.

Data definition: a **binary arithmetic S-expression** (BAS-exp) is either a number or a `(list op lft rgt)`, where `op` is either `'+` or `'*`, and `lft` and `rgt` are BAS-exps.

As before, the data definition leads to a template for writing functions that consume BAS-exps. For clarity, we define synonyms for list accessor functions that we use to retrieve components of an expression.

Data definition: a **binary arithmetic S-expression** (BAS-exp) is either a number or a `(list op lft rgt)`, where `op` is either `'+` or `'*`, and `lft` and `rgt` are BAS-exps.

As before, the data definition leads to a template for writing functions that consume BAS-exps. For clarity, we define synonyms for list accessor functions that we use to retrieve components of an expression.

```
(define op first)
(define left second)
(define right third)

(define (my-base-fn base)
  (cond
    [(number? base) ...]
    [(list? base) ... (op base) ...
                  ... (my-base-fn (left base)) ...
                  ... (my-base-fn (right base)) ...]))
```

```
(define (my-base-fn base)
  (cond
    [(number? base) ...]
    [(list? base) ... (op base) ...
                  ... (my-base-fn (left base)) ...
                  ... (my-base-fn (right base)) ...]))

(define (eval base)
  (cond
    [(number? base) base]
    [(list? base) (apply (op base)
                         (eval (left base))
                         (eval (right base)))]))

(define (apply op val1 val2)
  (cond
    [(symbol=? op '+) (+ val1 val2)]
    [(symbol=? op '*) (* val1 val2)]))
```

Stepping with expression trees

In DrRacket's Stepper, `'(* (+ 3 4) (+ 1 6))` steps to `'(* 7 (+ 1 6))`.

In DrRacket's Stepper, `'(* (+ 3 4) (+ 1 6))` steps to `'(* 7 (+ 1 6))`.

```
(define (reducible? base) (not (number? base)))

(define (one-step base)
  (cond
    [(and (number? (left base)) (number? (right base)))
       (apply (op base) (left base) (right base))]
    [(number? (left base))
       (list (op base) (left base) (one-step (right base)))]
    [else
       (list (op base) (one-step (left base)) (right base))]))

(define (eval2 base)
  (cond
    [(reducible? base) (eval2 (one-step base))]
    [else base]))
```

We can generalize to operations with more than two operands (exercise).

Generalizing binary arithmetic S-expressions

We can generalize to operations with more than two operands (exercise).

We can also allow symbols as well as numbers, letting us represent algebraic expressions with variables.

We can generalize to operations with more than two operands (exercise).

We can also allow symbols as well as numbers, letting us represent algebraic expressions with variables.

If we add binding forms like `let`, function creation like `lambda`, and conditionals like `cond`, we will have an interpreter for a Racket-like language written in Racket.

We can generalize to operations with more than two operands (exercise).

We can also allow symbols as well as numbers, letting us represent algebraic expressions with variables.

If we add binding forms like `let`, function creation like `lambda`, and conditionals like `cond`, we will have an interpreter for a Racket-like language written in Racket.

DrRacket is such an interpreter. Doing some of this work ourselves provides insight into its design, and lets us experiment with alternate syntax and semantics.

We can generalize to operations with more than two operands (exercise).

We can also allow symbols as well as numbers, letting us represent algebraic expressions with variables.

If we add binding forms like `let`, function creation like `lambda`, and conditionals like `cond`, we will have an interpreter for a Racket-like language written in Racket.

DrRacket is such an interpreter. Doing some of this work ourselves provides insight into its design, and lets us experiment with alternate syntax and semantics.

Putting a quote in front of a Racket expression turns it into data (an S-expression) that can be manipulated by a Racket program.

We can generalize to operations with more than two operands (exercise).

We can also allow symbols as well as numbers, letting us represent algebraic expressions with variables.

If we add binding forms like `let`, function creation like `lambda`, and conditionals like `cond`, we will have an interpreter for a Racket-like language written in Racket.

DrRacket is such an interpreter. Doing some of this work ourselves provides insight into its design, and lets us experiment with alternate syntax and semantics.

Putting a quote in front of a Racket expression turns it into data (an S-expression) that can be manipulated by a Racket program.

This opens up many possibilities for programs that write or compute with other programs.

We can generalize to operations with more than two operands (exercise).

We can also allow symbols as well as numbers, letting us represent algebraic expressions with variables.

If we add binding forms like `let`, function creation like `lambda`, and conditionals like `cond`, we will have an interpreter for a Racket-like language written in Racket.

DrRacket is such an interpreter. Doing some of this work ourselves provides insight into its design, and lets us experiment with alternate syntax and semantics.

Putting a quote in front of a Racket expression turns it into data (an S-expression) that can be manipulated by a Racket program.

This opens up many possibilities for programs that write or compute with other programs.

It also lets us prove a surprising result about the limits of computation.

Programs that don't terminate

```scheme
(define (forever n)
  (forever n))

(forever 1)
```

When this program is run, it does not produce a value, but runs forever.

Programs that don't terminate

```
(define (forever n)
  (forever n))

(forever 1)
```

When this program is run, it does not produce a value, but runs forever.

Can we detect behaviour like this?

Programs that don't terminate

```
(define (forever n)
  (forever n))

(forever 1)
```

When this program is run, it does not produce a value, but runs forever.

Can we detect behaviour like this?

It turns out that we cannot, and we can prove that we cannot.

Programs that don't terminate

```
(define (forever n)
  (forever n))
```

```
(forever 1)
```

When this program is run, it does not produce a value, but runs forever.

Can we detect behaviour like this?

It turns out that we cannot, and we can prove that we cannot.

Let’s start by specifying clearly what we are going to prove cannot be computed.

```
; halting?: (listof sexp) any -> boolean

(define (halting? sxl arg) ...)
```

The halting function

```
; halting?: (listof sexp) any -> boolean

(define (halting? sxl arg) ...)
```

The function `halting?` we are specifying consumes a list of S-expressions `sxl` (representing a list of definitions in a program) and a value `arg`.

```
; halting?: (listof sexp) any -> boolean

(define (halting? sxl arg) ...)
```

The function `halting?` we are specifying consumes a list of S-expressions `sxl` (representing a list of definitions in a program) and a value `arg`.

It produces `true` if and only if the last function in `sxl` (assumed to be a function with one parameter) **halts** when applied argument `arg`.

The halting function

```
; halting?: (listof sexp) any -> boolean

(define (halting? sxl arg) ...)
```

The function `halting?` we are specifying consumes a list of S-expressions `sxl` (representing a list of definitions in a program) and a value `arg`.

It produces `true` if and only if the last function in `sxl` (assumed to be a function with one parameter) **halts** when applied argument `arg`.

Our goal is to show that any attempt to complete the definition of `halting?` will be incorrect.

```
; halting?: (listof sexp) any -> boolean

(define (halting? sxl arg) ...)
```

The function `halting?` we are specifying consumes a list of S-expressions `sxl` (representing a list of definitions in a program) and a value `arg`.

It produces `true` if and only if the last function in `sxl` (assumed to be a function with one parameter) **halts** when applied argument `arg`.

Our goal is to show that any attempt to complete the definition of `halting?` will be incorrect.

Suppose someone gave us code for `halting?`. We can then write code that uses it.

```
; perverse: (listof sexp) -> boolean

(define (perverse x)
  (cond
    [(halting? x x) (forever 1)]
    [else true]))
```

```
; perverse: (listof sexp) -> boolean

(define (perverse x)
  (cond
    [(halting? x x) (forever 1)]
    [else true]))
```

Let `p` be the list of S-expressions with the definitions of `halting?`, `forever`, and `perverse`.

The halting problem is uncomputable

```
; perverse: (listof sexp) -> boolean

(define (perverse x)
  (cond
    [(halting? x x) (forever 1)]
    [else true]))
```

Let `p` be the list of S-expressions with the definitions of `halting?`, `forever`, and `perverse`.

Does `(perverse p)` halt or not?

```
; perverse: (listof sexp) -> boolean

(define (perverse x)
  (cond
    [(halting? x x) (forever 1)]
    [else true]))
```

Let `p` be the list of S-expressions with the definitions of `halting?`, `forever`, and `perverse`.

Does `(perverse p)` halt or not?

If `(perverse p)` halts...

```
; perverse: (listof sexp) -> boolean

(define (perverse x)
  (cond
    [(halting? x x) (forever 1)]
    [else true]))
```

Let `p` be the list of S-expressions with the definitions of `halting?`, `forever`, and `perverse`.

Does `(perverse p)` halt or not?

If `(perverse p)` halts...

If `(perverse p)` does not halt...

The halting problem is uncomputable

```
; perverse: (listof sexp) -> boolean

(define (perverse x)
  (cond
    [(halting? x x) (forever 1)]
    [else true]))
```

Let `p` be the list of S-expressions with the definitions of `halting?`, `forever`, and `perverse`.

Does `(perverse p)` halt or not?

If `(perverse p)` halts...

If `(perverse p)` does not halt...

In either case, we can show that `(halting? p p)` must produce the wrong answer.

```
; perverse: (listof sexp) -> boolean

(define (perverse x)
  (cond
    [(halting? x x) (forever 1)]
    [else true]))
```

Let `p` be the list of S-expressions with the definitions of `halting?`, `forever`, and `perverse`.

Does `(perverse p)` halt or not?

If `(perverse p)` halts...

If `(perverse p)` does not halt...

In either case, we can show that `(halting? p p)` must produce the wrong answer.

The idea in this proof is due to Alan Turing, from 1936, a few months after Alonzo Church published his work on the lambda calculus.

The significance of the halting problem

The idea in this proof is due to Alan Turing, from 1936, a few months after Alonzo Church published his work on the lambda calculus.

Turing's model of computation is very different from Church's, but they are equivalent in power. Both have been influential in the development of computer science.

The significance of the halting problem

The idea in this proof is due to Alan Turing, from 1936, a few months after Alonzo Church published his work on the lambda calculus.

Turing's model of computation is very different from Church's, but they are equivalent in power. Both have been influential in the development of computer science.

The halting problem can be used to show that most questions asked about programs are uncomputable.

The significance of the halting problem

The idea in this proof is due to Alan Turing, from 1936, a few months after Alonzo Church published his work on the lambda calculus.

Turing's model of computation is very different from Church's, but they are equivalent in power. Both have been influential in the development of computer science.

The halting problem can be used to show that most questions asked about programs are uncomputable.

It can even be used to show that questions that don't seem to involve programs at all are uncomputable.

The idea in this proof is due to Alan Turing, from 1936, a few months after Alonzo Church published his work on the lambda calculus.

Turing's model of computation is very different from Church's, but they are equivalent in power. Both have been influential in the development of computer science.

The halting problem can be used to show that most questions asked about programs are uncomputable.

It can even be used to show that questions that don't seem to involve programs at all are uncomputable.

For example, whether or not a set of polynomial equations has an integer solution is uncomputable.

The idea in this proof is due to Alan Turing, from 1936, a few months after Alonzo Church published his work on the lambda calculus.

Turing's model of computation is very different from Church's, but they are equivalent in power. Both have been influential in the development of computer science.

The halting problem can be used to show that most questions asked about programs are uncomputable.

It can even be used to show that questions that don't seem to involve programs at all are uncomputable.

For example, whether or not a set of polynomial equations has an integer solution is uncomputable.

We are still waiting for a similar proof to show that some "natural" problem cannot be computed *efficiently*.