CSE1729: Introduction to Programming

Trees, Heaps, Stacks & Queues… in SCHEME

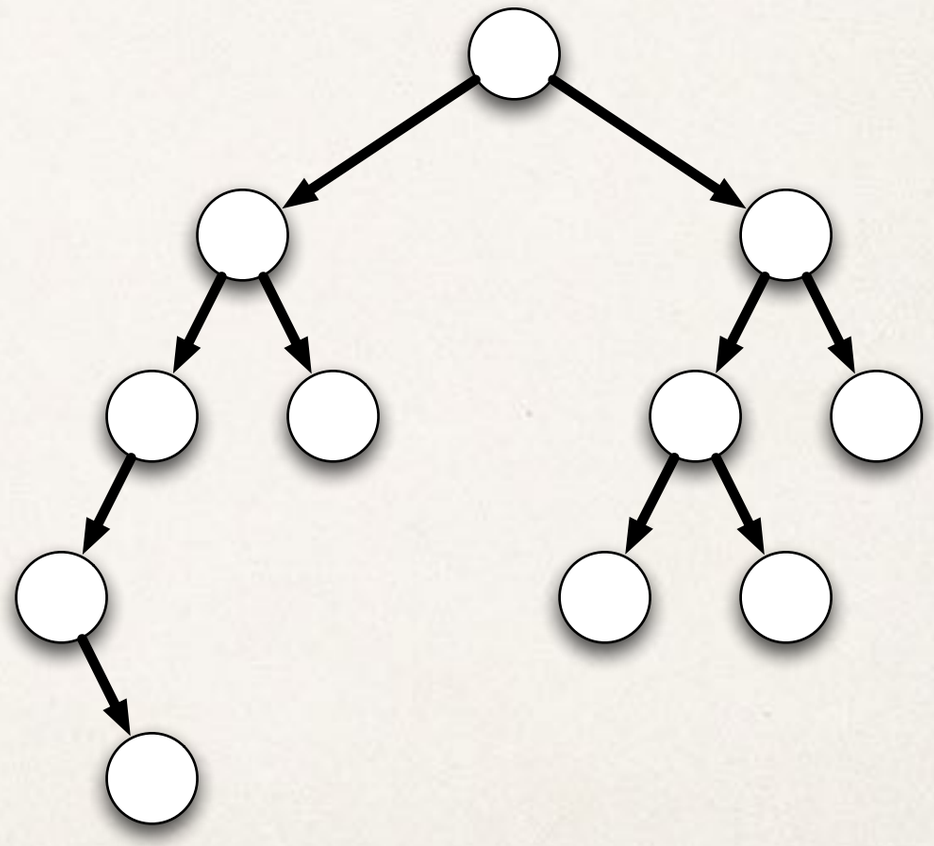
Alexander Russell

Trees

- The basic pair structure we have introduced is extremely flexible. A natural extension: *trees*.

- A *tree* is a natural hierarchical data structure. We'll focus on a variant called *binary trees*.

A binary tree

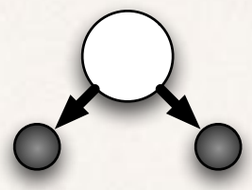


Trees, a definition

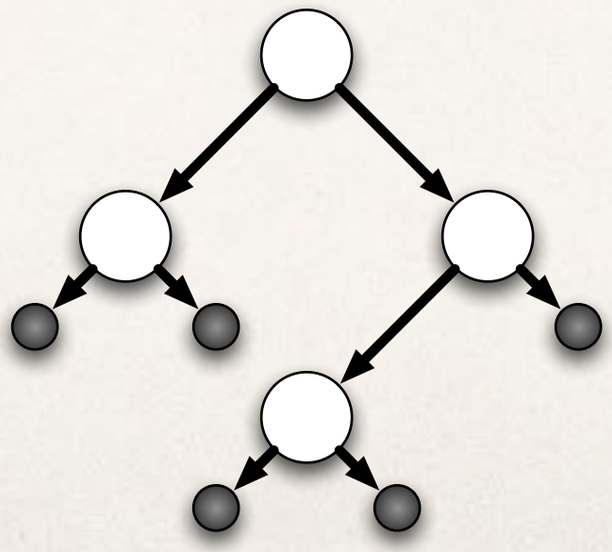
- The notion of tree can be defined recursively:

- A tree is either:
  - The *empty* tree.
  - A *node, with arrows pointing to two trees.*

The empty tree



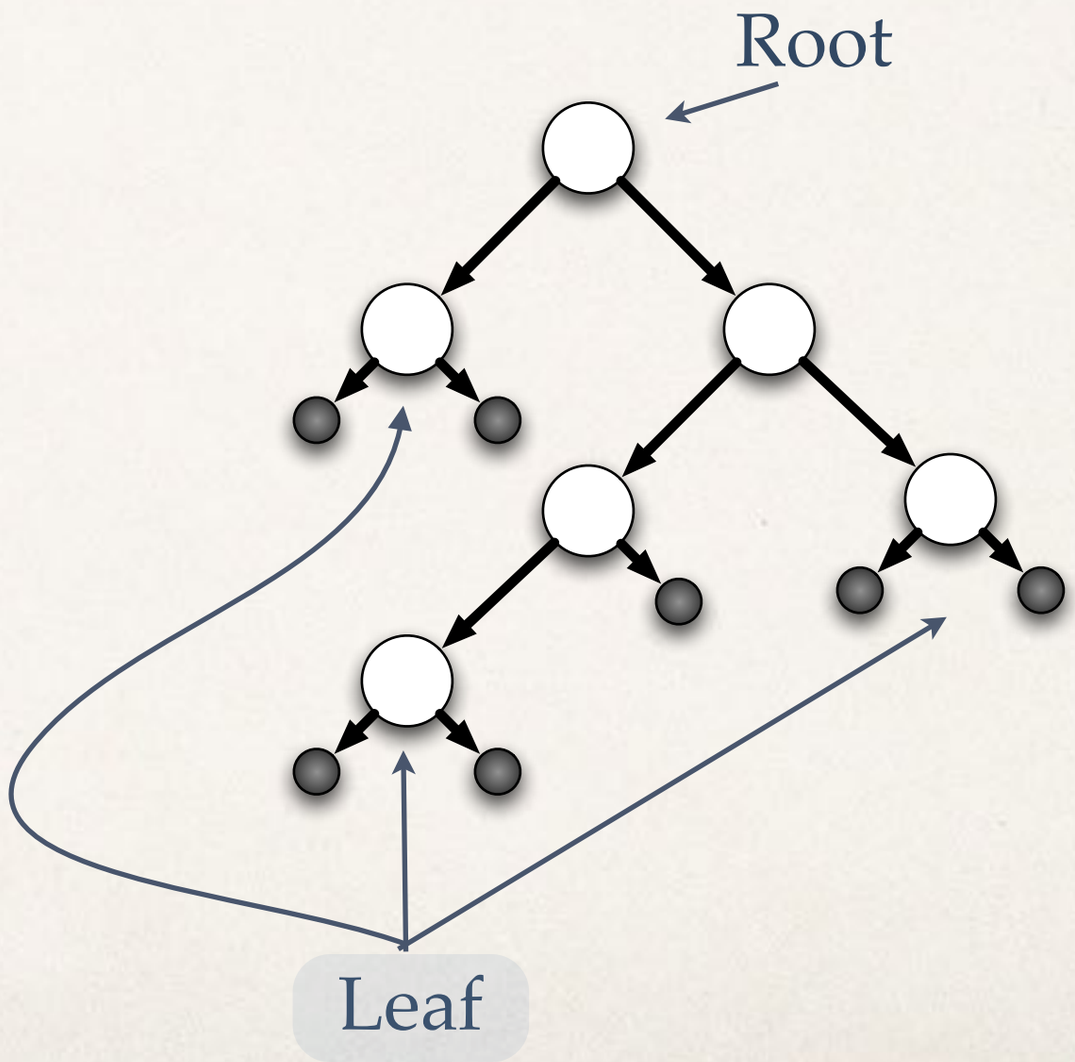
A larger tree: A node with arrows pointing to two empty trees.



A more complicated tree with four nodes

Terminology

- The top of the tree is the *root*.
- The *children* of a node are the roots of the trees to which it points.
- A *leaf* is a node with no children (so it points to two empty trees).

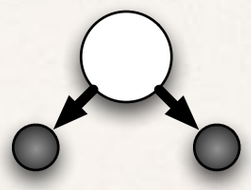


Depth

- The depth of a tree is length of the longest path from the root to a leaf.

Depth undefined

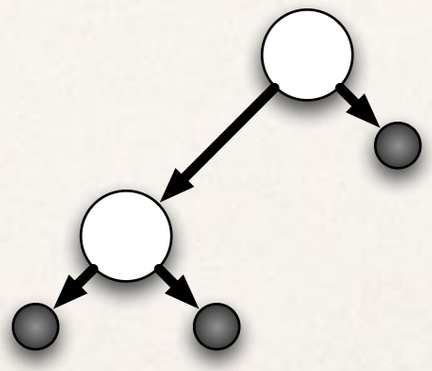
- We do not define the depth of the empty tree.



Depth 0

- A tree with a single node has depth 0.

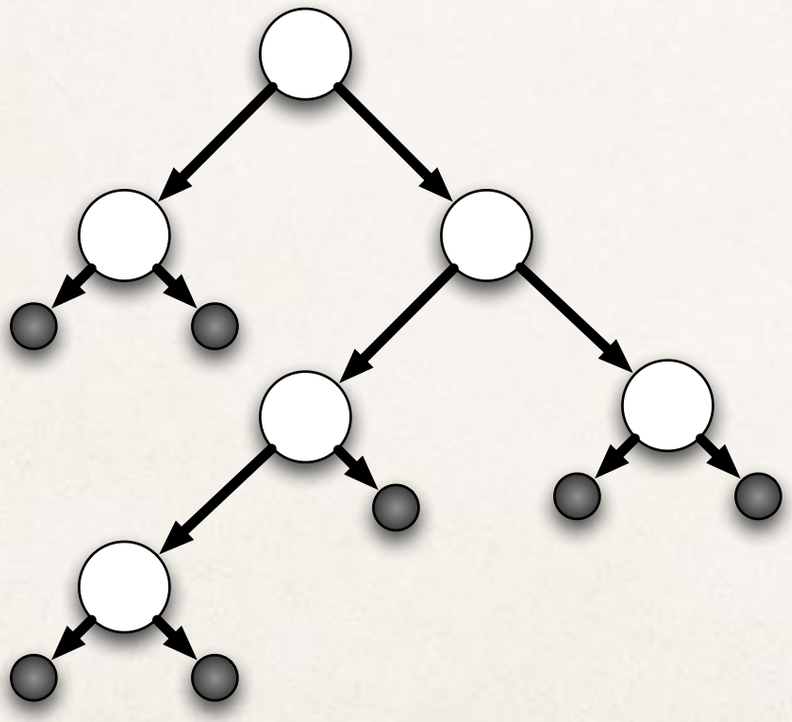
- Otherwise, notice that the depth of a tree is one more than the maximum depth of the trees rooted at its children.



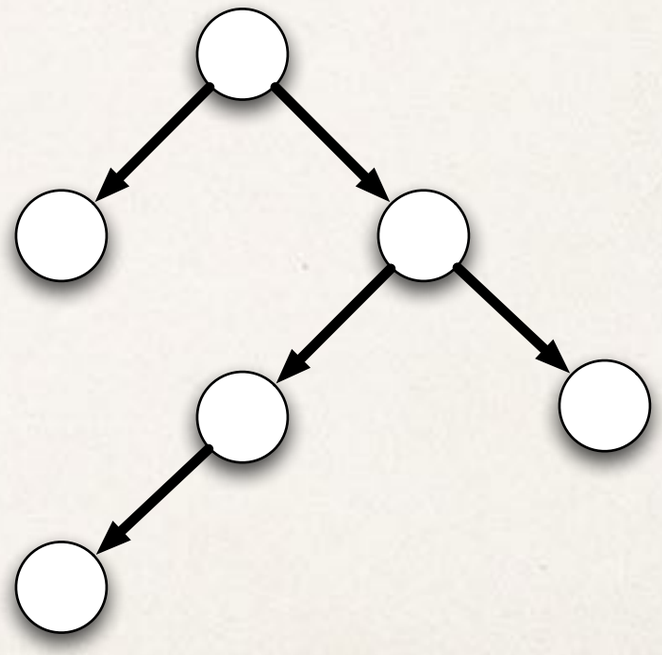
Depth 1

Conventions

- When we draw trees, we typically suppress the empty trees.
- Thus:



is drawn...



Motivating trees: Maintaining a Set

- Why might you wish to store data this way?

- Consider the basic task of maintaining a set of numbers. We'd like to be able to

  - **INSERT** a number into our set, and

  - **TEST MEMBERSHIP**: Determine if a given number is an element of our set.

Maintaining a set with a list

- What's the problem? We have a data structure we can use for this purpose: a list.
  - To **INSERT** a number: Easy! Just add it at the beginning of the list.
  - To **TEST MEMBERSHIP**: Easy! Just scan the list from left to right.

```
(define (insert element set)
  (cons element set))
```

Note..this returns the new set

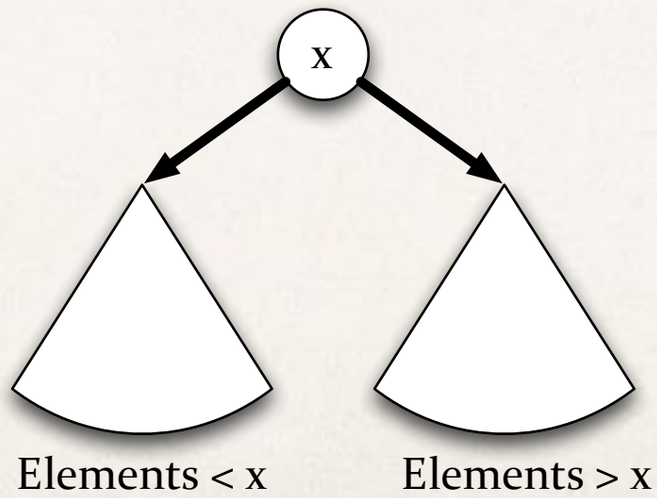
```
(define (ismember? element set)
  (cond ((null? set) #f)
        ((equal? element (car set)) #t)
        (else (ismember? element (cdr set)))))
```

What's the problem?

- So...
  - Insertion is *FAST*. A single function call to `cons`.
    - But...testing membership is *SLOW*. We may have to scan the entire list each time.
- Unfortunately, in practice, most processing with sets tends to be membership queries.
- We'd like a way to maintain sets so that both insertion and membership are fast.

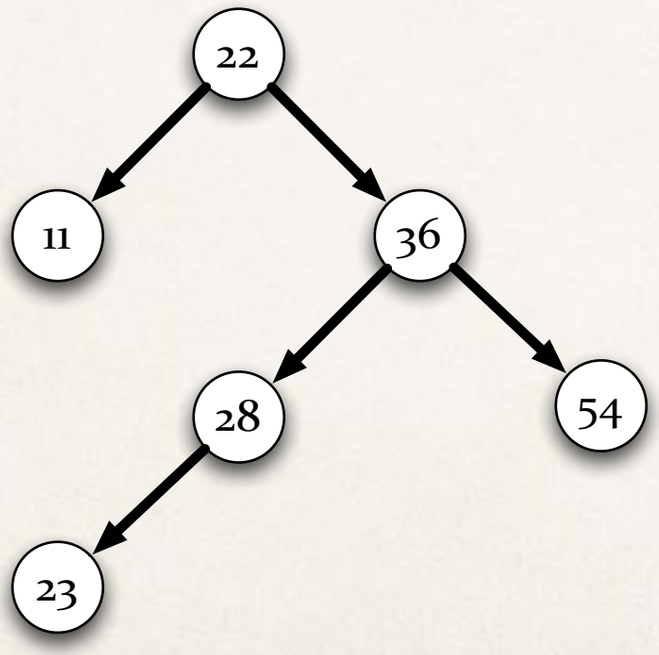
Idea: Maintain the set in a **Binary Search Tree**

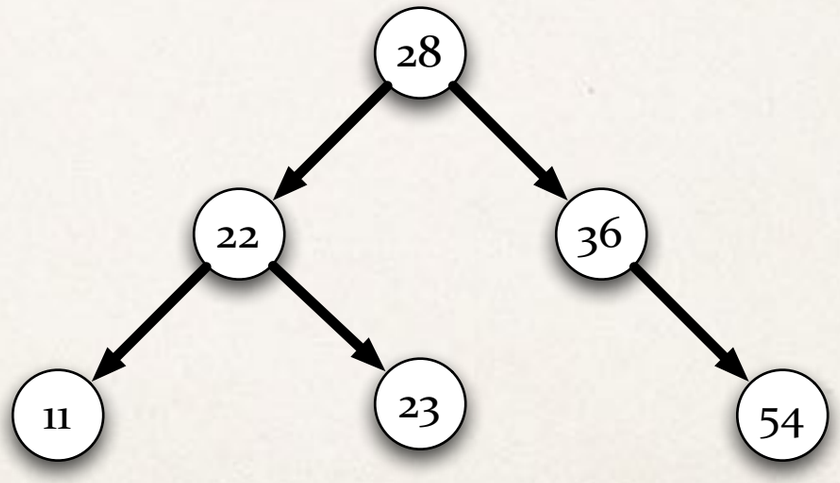
- A binary search tree for the set S:
  - Elements of S are placed on the nodes in such a way that...
  - **Rule**: elements in the left subtree of x are smaller than x; elements in the right subtree are larger than x.



Binary search trees

- Note that a given set can have many binary search trees.
- Consider the set {11, 22, 23, 28, 36, 54}. Two binary search trees are shown below:





Searching for an element in a binary search tree

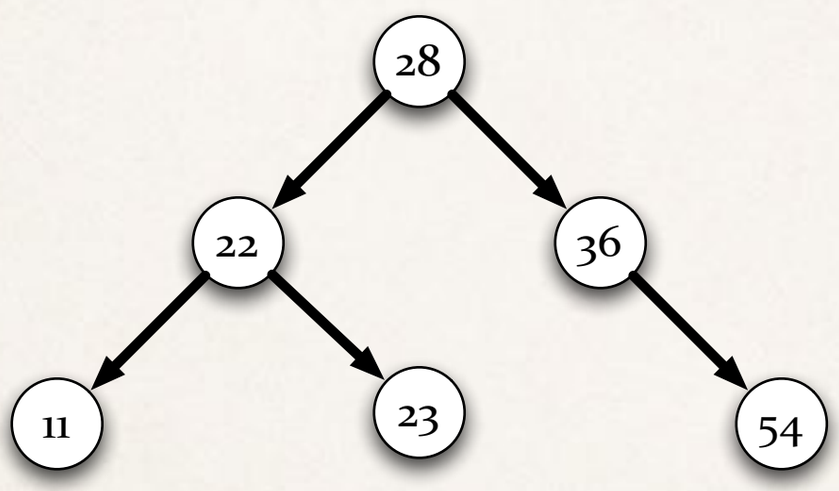
- Element?(x,T)
  - If T is empty, return `#f`.
  - If x = root(T), return `#t`.
  - Otherwise:
    - if x > root(T) return Element?(x, RightChild(T));
    - if x < root(T) return Element?(x, LeftChild(T)).

Two examples

- Element?(23,T)
- Element?(31,T)

Two examples

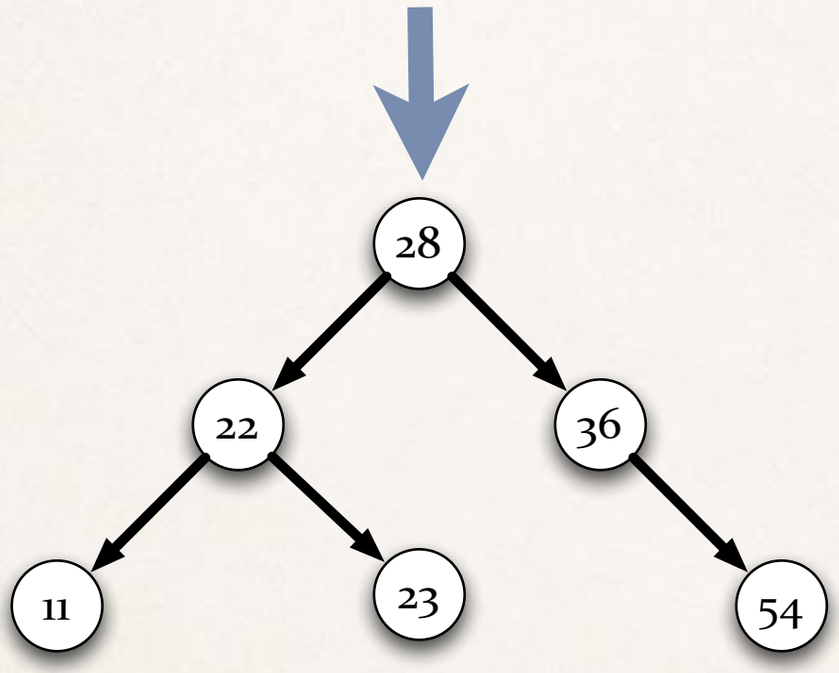
- Element?(23,T)



- Element?(31,T)

Two examples

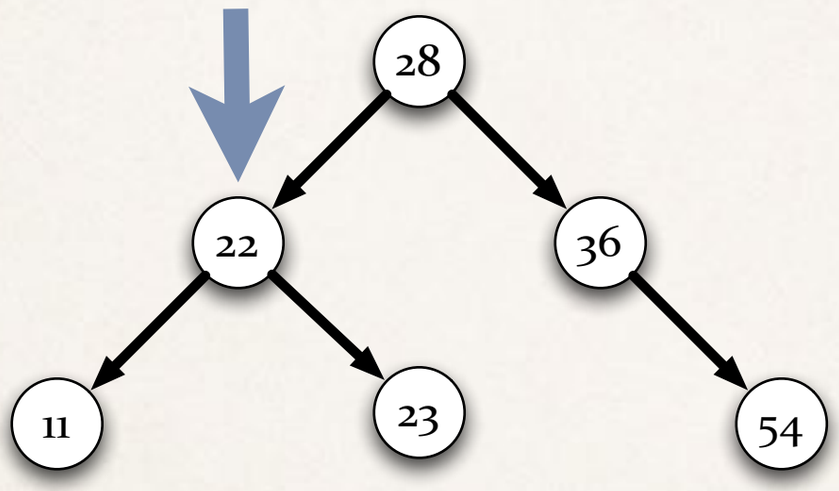
- Element?(23,T)



- Element?(31,T)

Two examples

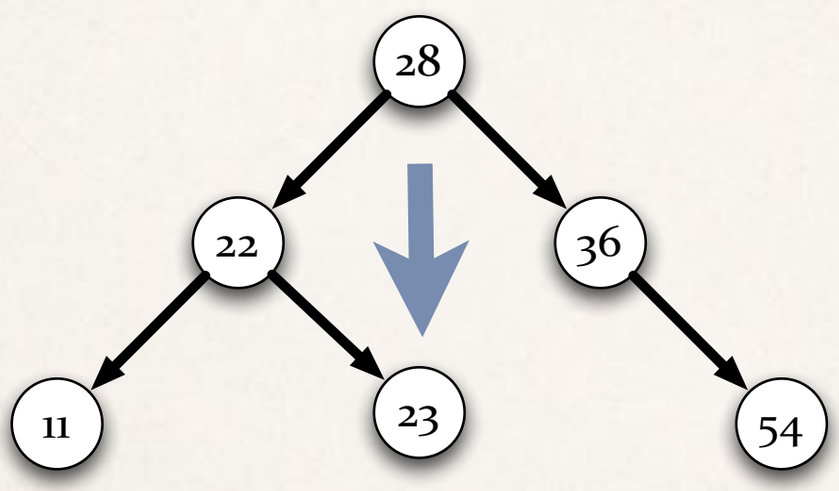
- Element?(23,T)



- Element?(31,T)

Two examples

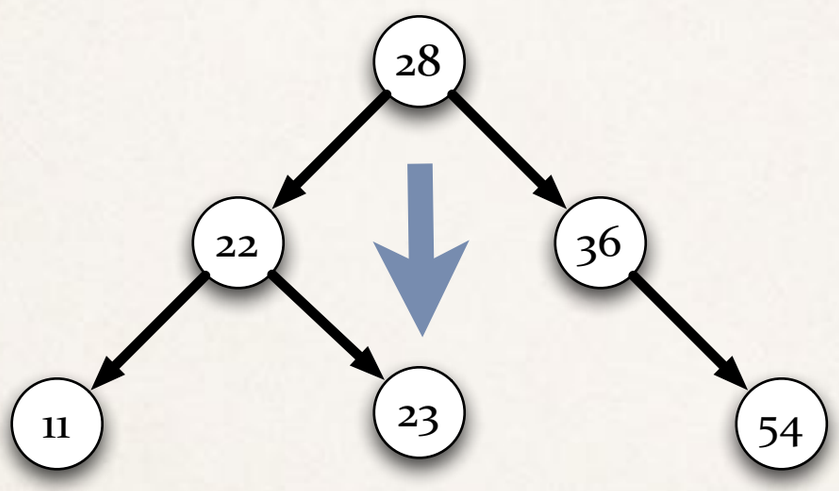
- Element?(23,T)



- Element?(31,T)

Two examples

- Element?(23,T)

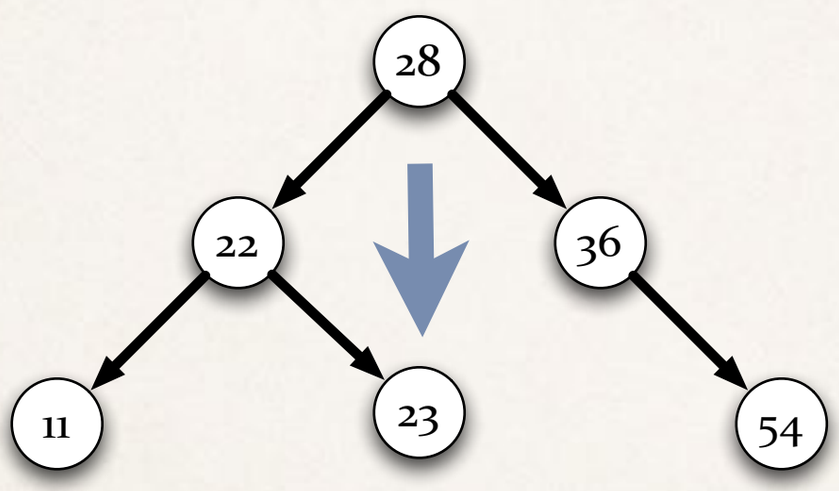


Found it!

- Element?(31,T)

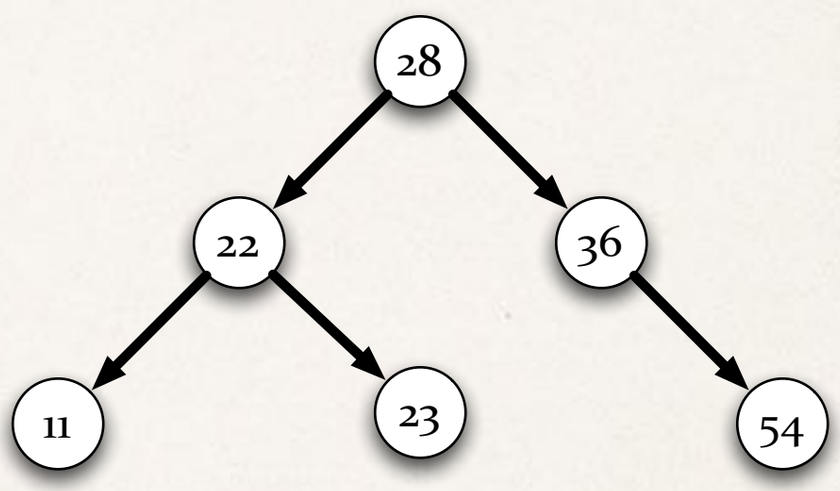
Two examples

- Element?(23,T)



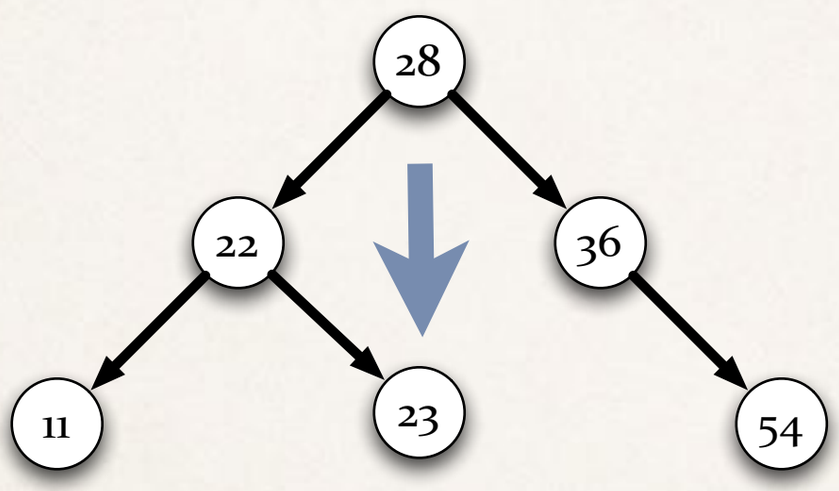
Found it!

- Element?(31,T)



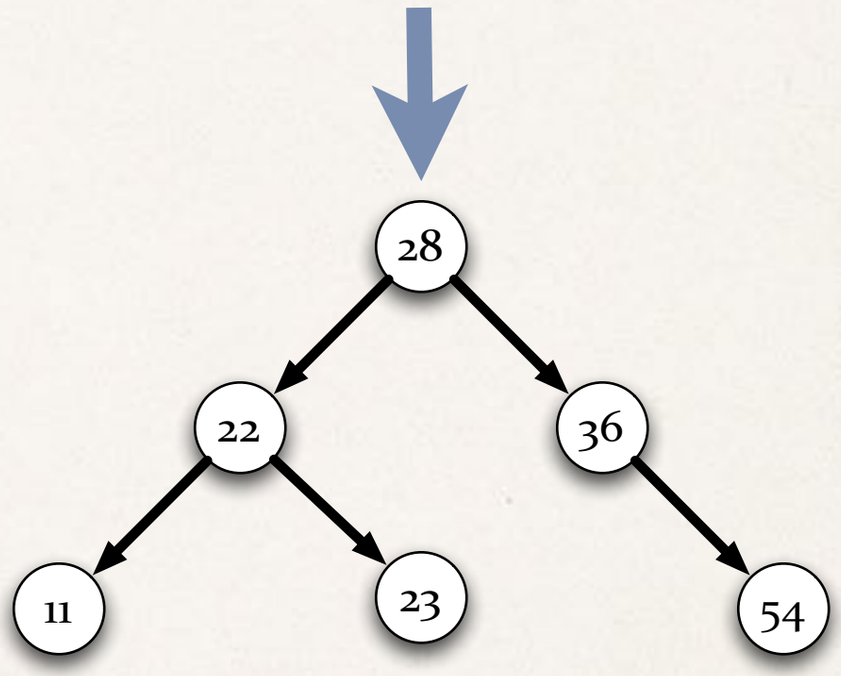
Two examples

- Element?(23,T)



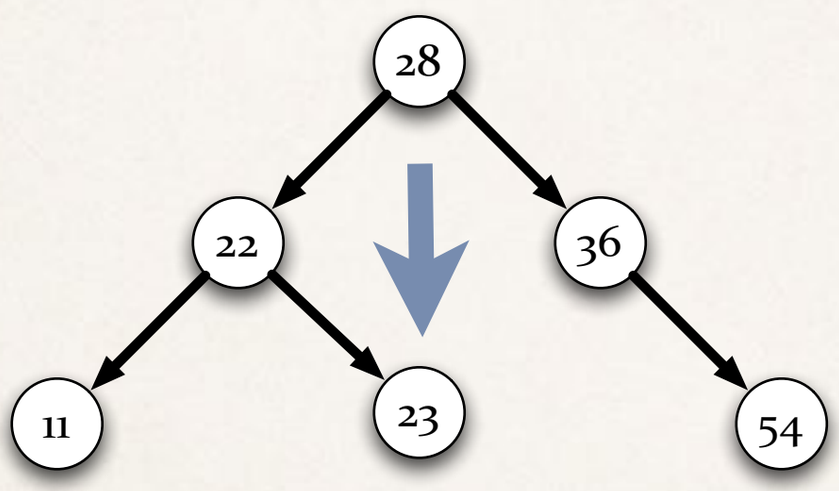
Found it!

- Element?(31,T)



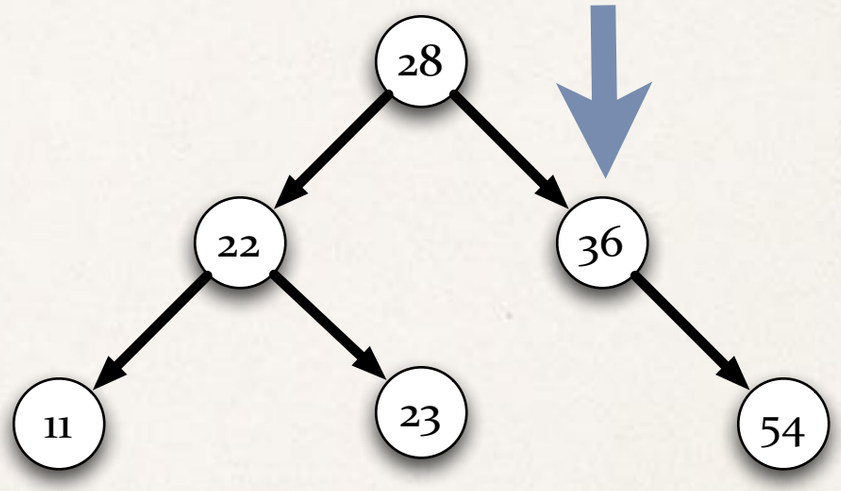
Two examples

- Element?(23,T)



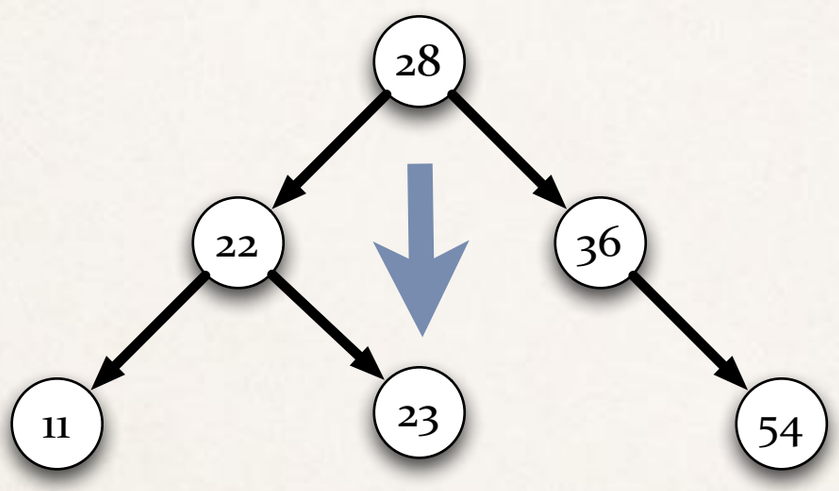
Found it!

- Element?(31,T)



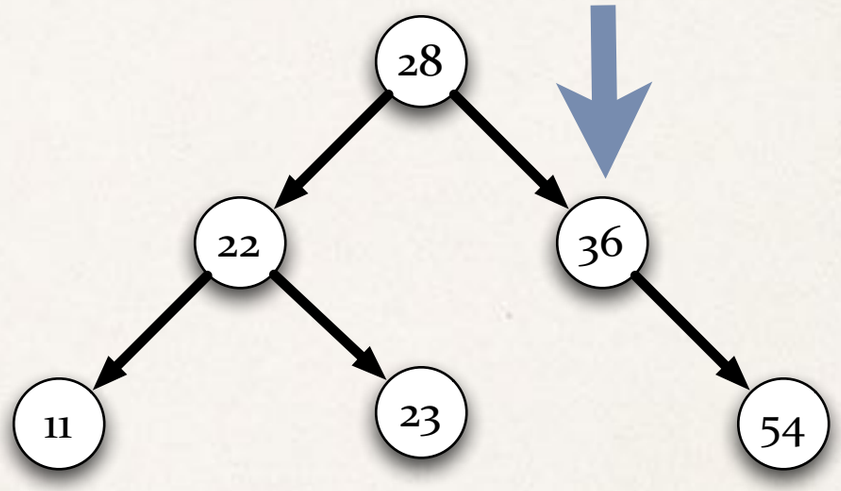
Two examples

- Element?(23,T)



Found it!

- Element?(31,T)



It’s not here!

Inserting an element into a binary search tree

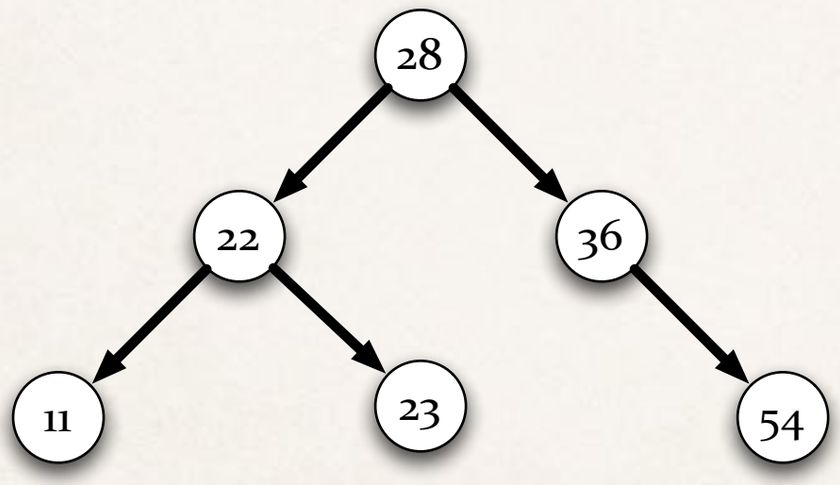
- `Insert`(x, T). To insert an element x into a tree T:
  - If T is empty, return a new node containing x.
  - Otherwise, if x < root(T), insert into the left subtree of T; if x > root(T), insert into the right subtree of T. Return the resulting T.
- Note: The tree T is "traversed" in the same way by both `Insertion` and `Element?`.

An example

- Insert(38,T)

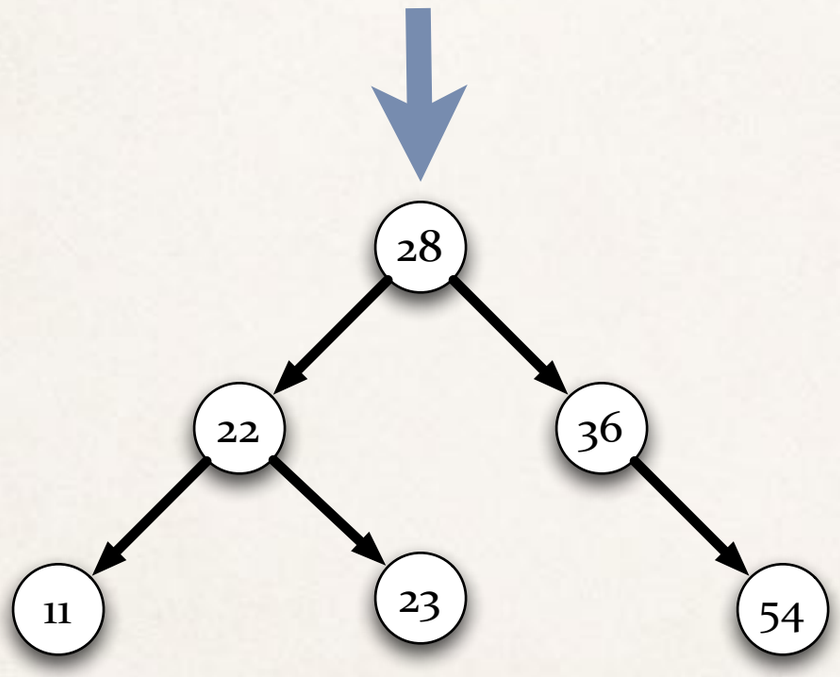
An example

- Insert(38,T)



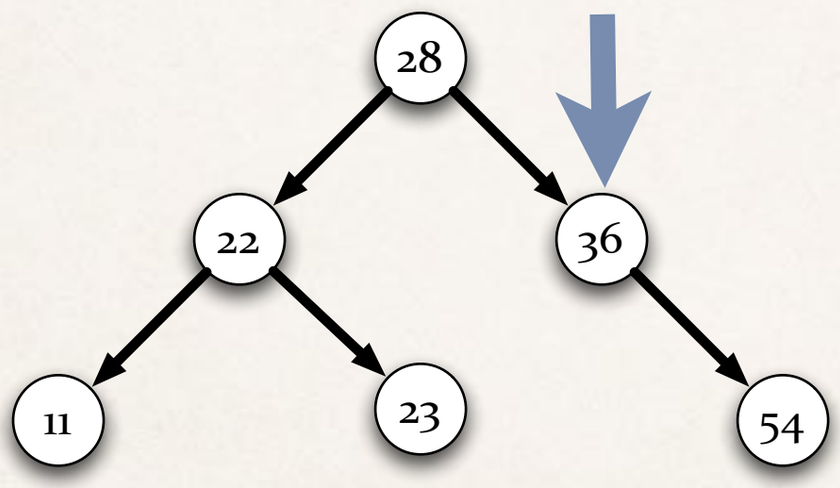
An example

- Insert(38,T)



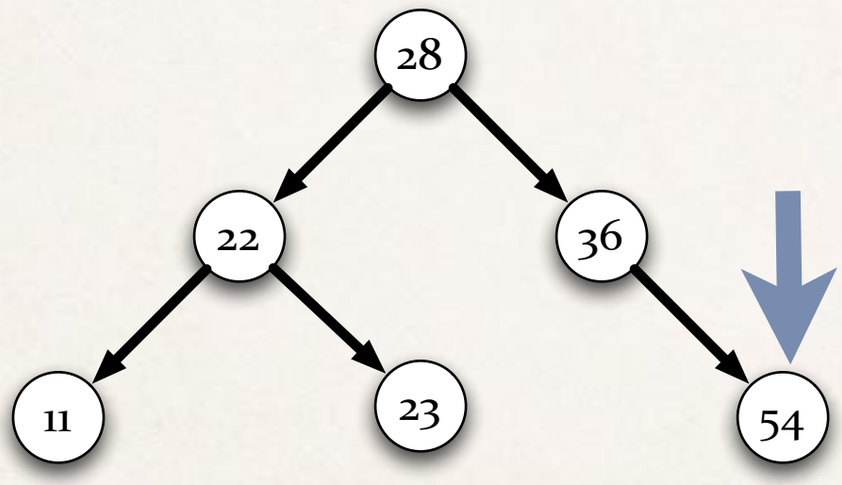
An example

- Insert(38,T)



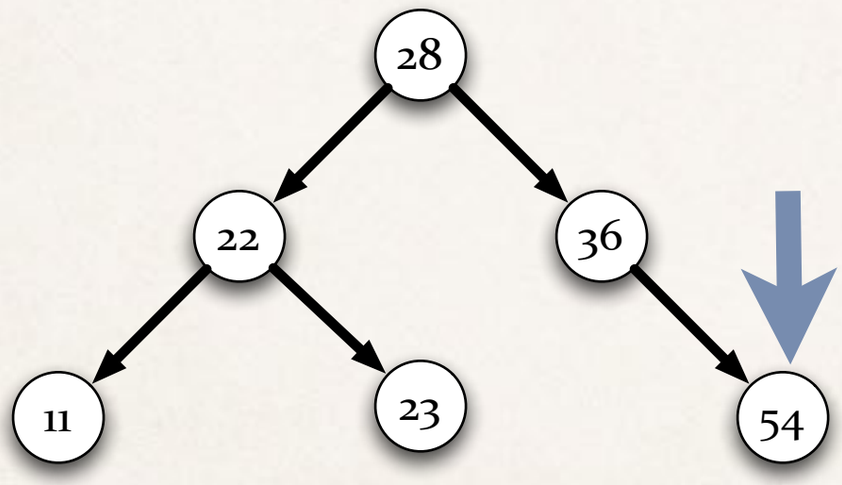
An example

- Insert(38,T)



An example

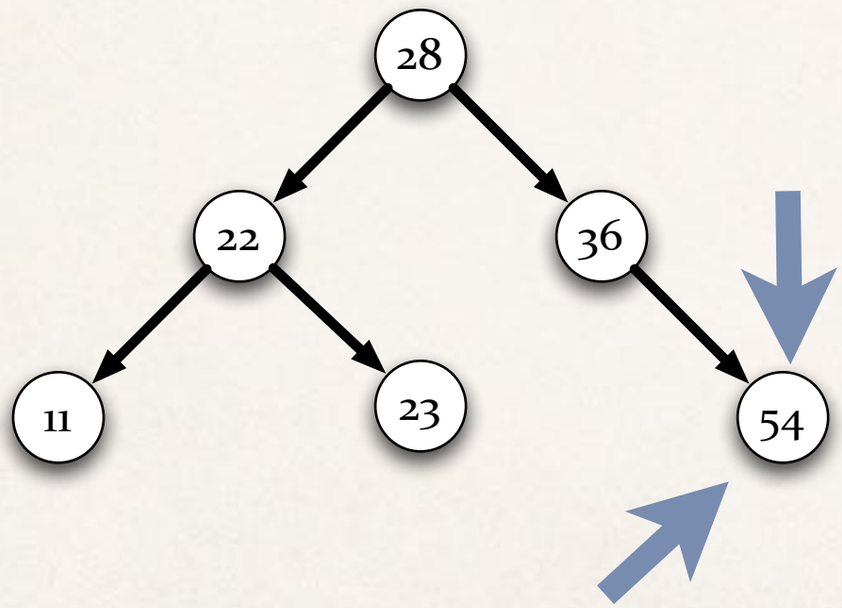
- Insert(38,T)



So…insert here

An example

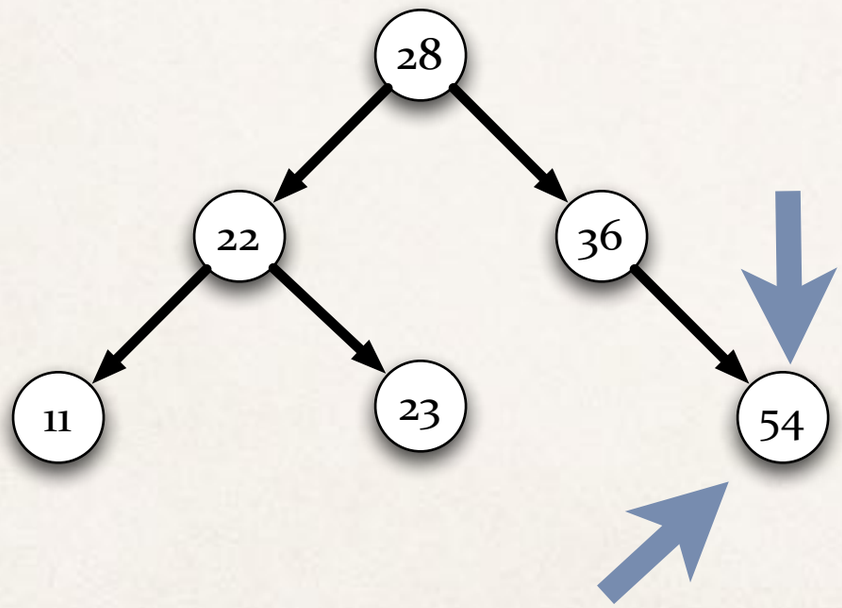
- Insert(38,T)



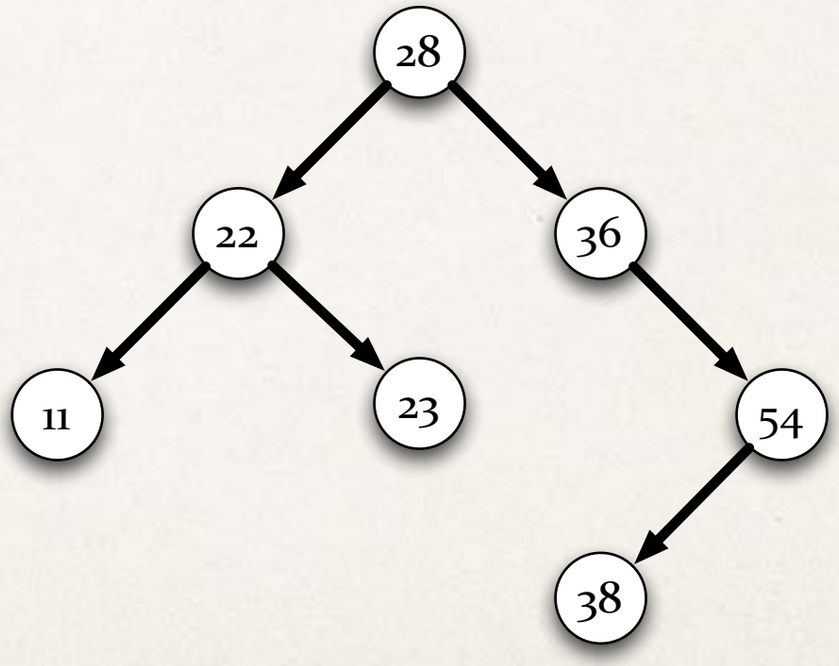
So…insert here

An example

- Insert(38,T)



So…insert here



How expensive is a membership query?

- In general, to determine membership (or insert an element), we may have to traverse the tree from the root to a leaf. In the worst case, this will involve a number of function calls proportional to

The DEPTH of the tree.

- Question: How can we expect depth to scale with the number of elements?

Depth versus size

- **Depth**: longest path in a tree.

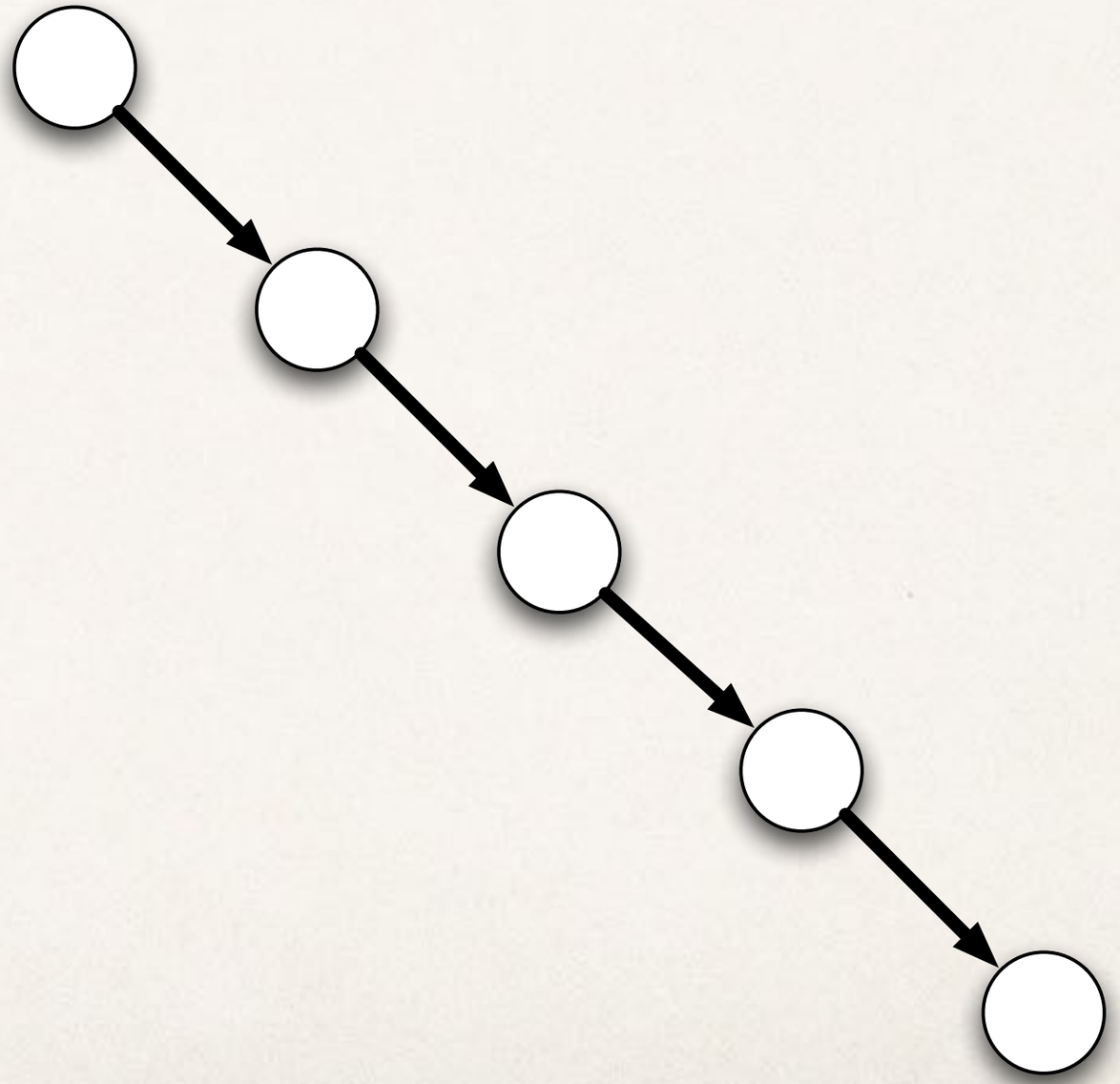
- **Size**: total number of nodes in a tree.

- These could be comparable: a long skinny tree.

- Size could be much larger: a bushy tree.

Depth versus size

- **Depth**: longest path in a tree.
- **Size**: total number of nodes in a tree.
- These could be comparable: a long skinny tree.
- Size could be much larger: a bushy tree.



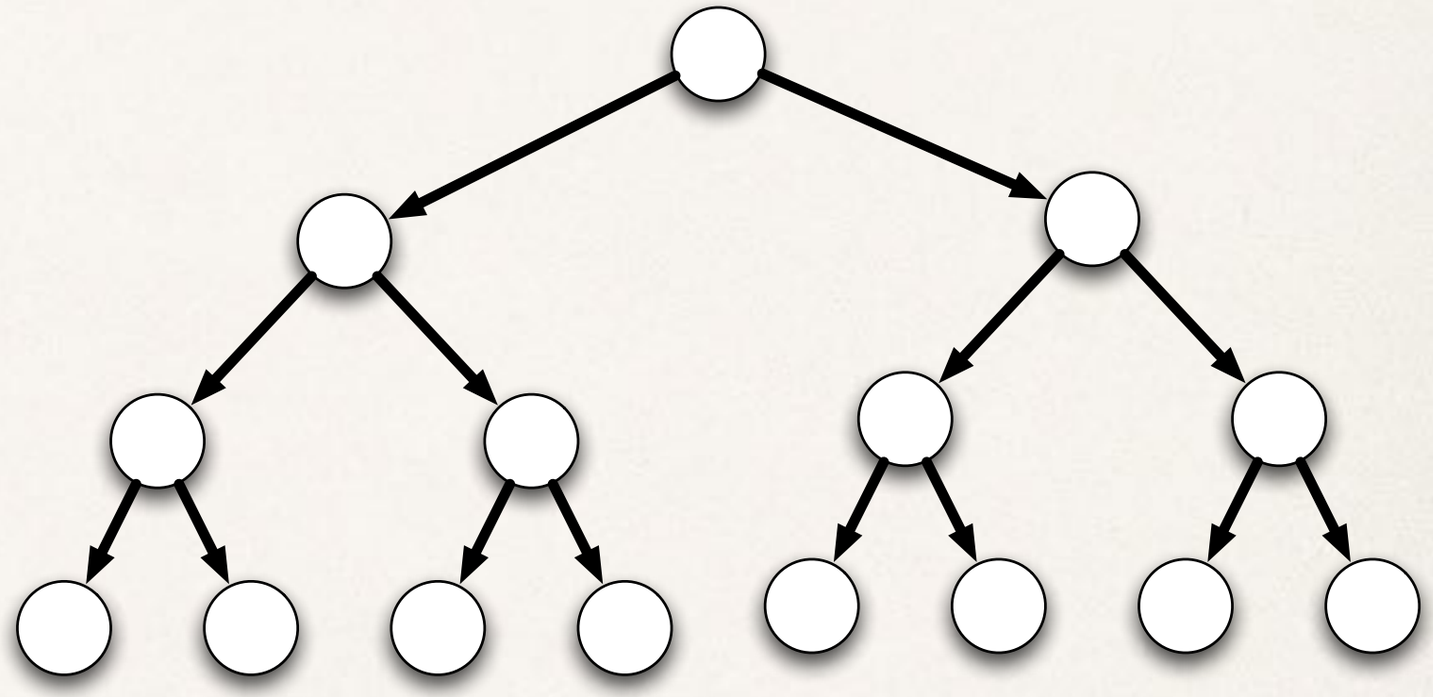
Depth versus size

- **Depth**: longest path in a tree.

- **Size**: total number of nodes in a tree.

- These could be comparable: a long skinny tree.

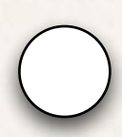
- Size could be much larger: a bushy tree.



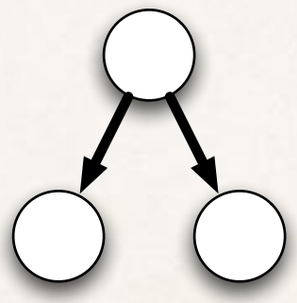
The size of a bushy tree

- Consider "complete" trees of various depths. How many elements do they have?

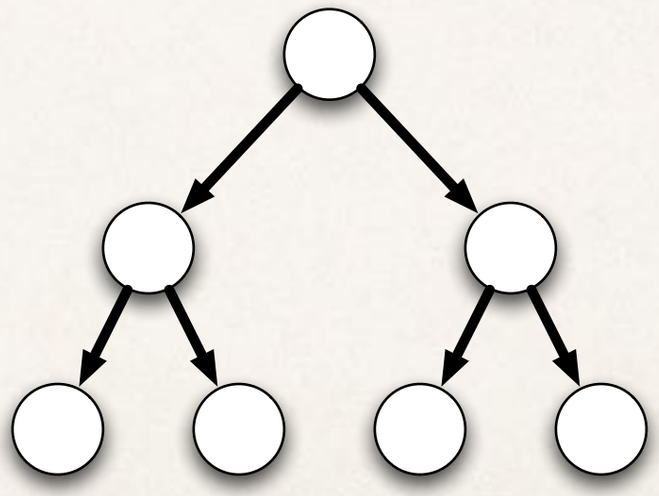
depth: 0
size: 1



depth: 1
size: 3

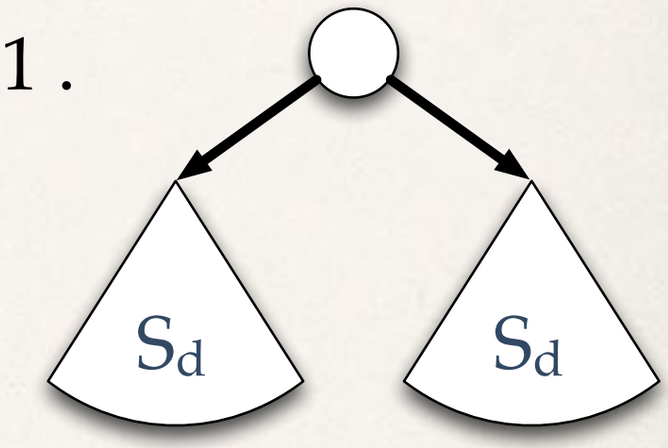


depth: 2
size: 7



In general…

- Let S_d be the size for depth d. Then $S_{d+1} = 2 * S_d + 1$.



- The sequence: 1, 3, 7, 15, 31, 63, ... may be familiar. It is the sequence 2, 4, 8, 16, 32, 64, less one. Or:

$$S_d = 2^{d+1} - 1 \quad \text{(approximately } 2^{d+1}\text{)}$$

- It follows that we can pack an n element set into a very shallow tree. If $2^d > n$, we can do it with depth d. We can have d about $\log_2 n$.

- $\log_2 n$ grows very slowly. If $n = 100{,}000{,}000$ then $\log_2 n \sim 27$.

Log grows slowly; 2^x grows quickly.

- S = # students at UCONN. Log(S) = 14.
- G = # base pairs in the human genome. Log(G) = 31.
- H = # humans on Earth. Log(H) = 33.
- C = # cells in your body. Log (C) = 45.
- M_E = mass of Earth in kg. Log(M_E) = 83.
- M_S = mass of our sun in kg. Log(M_S) = 101.
- M_U = mass of observable universe in kg. Log(M_U) = 180.

...and the universe is a big place.

The Hubble ultra deep field shot

So...for practical purposes... Is Log(X) always less than 200?

- C = number of different possible chess matches. Log(C) = 402.
- L = number of ways to list the numbers {1,...,1000}. Log(L) = 8529.
- G = number of different possible Go matches. Log(G) = 10^{48}.
- F_n = the nth Fibonacci number. Log(F_n) = n/7 .
- B_n = number of binary search trees for n numbers. Log(B_n) = 4 n .

Implementing binary search trees in SCHEME

- To represent a node of a tree, we need to maintain 3 data items: the **value** at the node, the **left subtree**, and the **right subtree**.

- We choose to represent these as a list:

```
(value left-subtree right-subtree)
```

- Define helper functions to create trees and extract these fields:

```
(define (make-tree value left right)
  (list value left right))

(define (value T) (car T))
(define (right T) (caddr T))
(define (left T)  (cadr T))
```

cadr, the car of the cdr

Tree insertion & membership

```
(define (element? x T)
  (cond ((null? T) #f)
        ((eq? x (value T)) #t)
        ((< x (value T)) (element? x (left T)))
        ((> x (value T)) (element? x (right T)))))
```

```
(define (insert x T)
  (cond ((null? T) (make-tree x '() '()))
        ((eq? x (value T)) T)
        ((< x (value T)) (make-tree (value T)
                                    (insert x (left T))
                                    (right T)))
        ((> x (value T)) (make-tree (value T)
                                    (left T)
                                    (insert x (right T))))))
```

Returns the tree resulting from insertion of x into T

Maintaining sets with trees: Performance

- What happens if you insert elements into a tree in sorted order? You get a skinny tree.

- However, one can prove that if elements are inserted into a tree in random order, the tree is near-bushy with high probability.

- There are also fancier ways of **insuring** that trees remain balanced. 2-3 trees, splay trees, AVL trees, etc. You'll learn about some of them in your algorithms course.

Extracting a sorted list from a binary search tree

- Note that it is easy to extract the a sorted list of element from a binary search tree:

```
(define (extract-sorted T)
  (if (null? T)
      '()
      (append (extract-sorted (left T))
              (list (value T))
              (extract-sorted (right T)))))
```

`(append list1 list2 ...)` appends the lists together

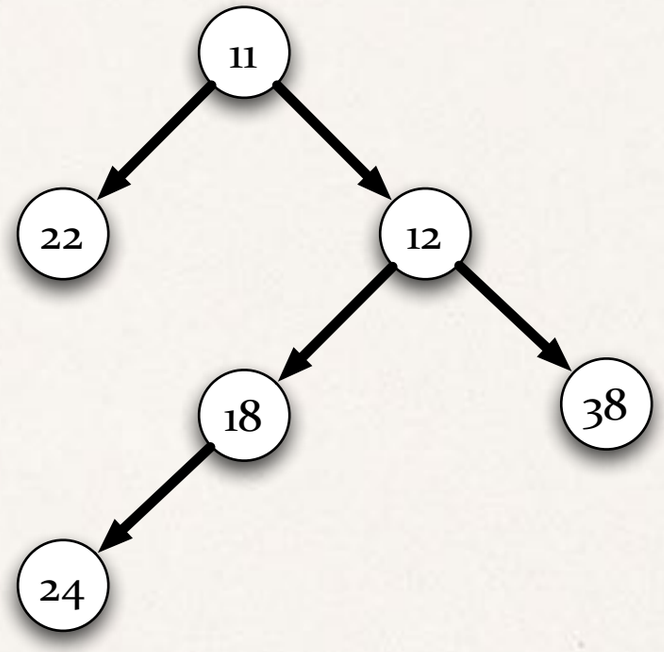
- One can then use this machinery to sort by inserting the elements into the search tree, and then extracting the sorted list.

Another natural tree structure

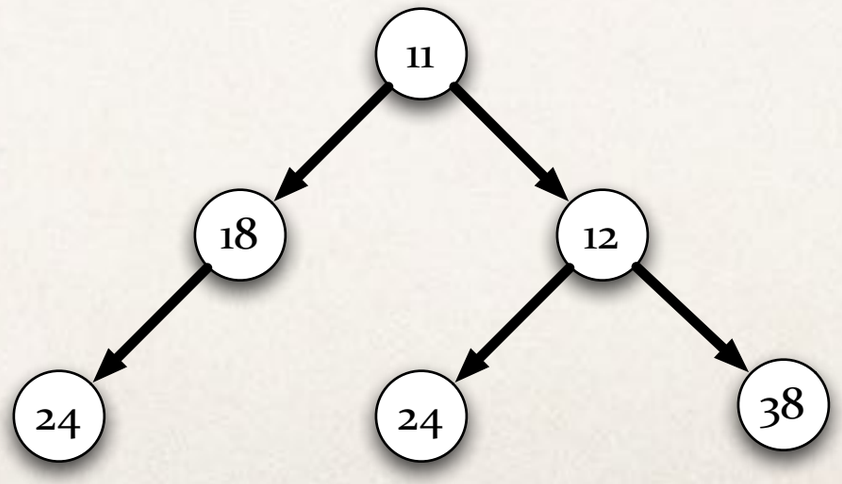
- A *heap* is a tree, with numbers stored at the nodes, with a different property:
  - HEAP PROPERTY: The value of any node is smaller than that of its children.
- Notice that it is easy to determine the minimum element of a heap: it's always the root!
- If we can find a way to remove the minimum element and retain the heap property, we could use a heap for sorting. How?
- I. Build a heap, II. Repeatedly extract the minimum element.

An example

- Consider the following heap containing the set {11, 12, 18, 22, 24, 38}



- Note that as with binary search trees, there are many different heaps containing the same set.



Building balanced heaps

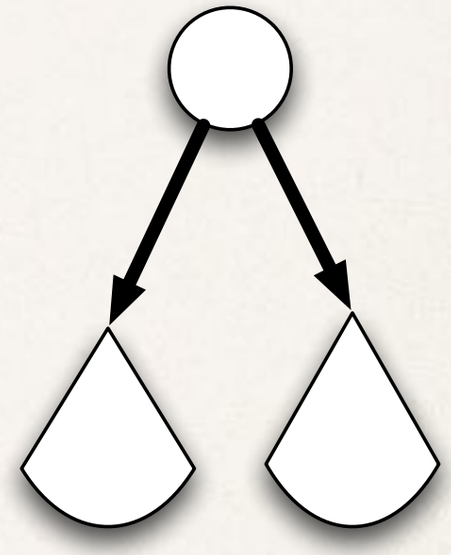
- Forcing a binary search tree to be balanced requires delicate algorithmic work.

- Heaps guarantee a weaker property: it's easy to construct a balanced heap.

- Idea: *Take turns choosing the insertion subtree*.

Building balanced heaps

- Forcing a binary search tree to be balanced requires delicate algorithmic work.



- Heaps guarantee a weaker property: it's easy to construct a balanced heap.

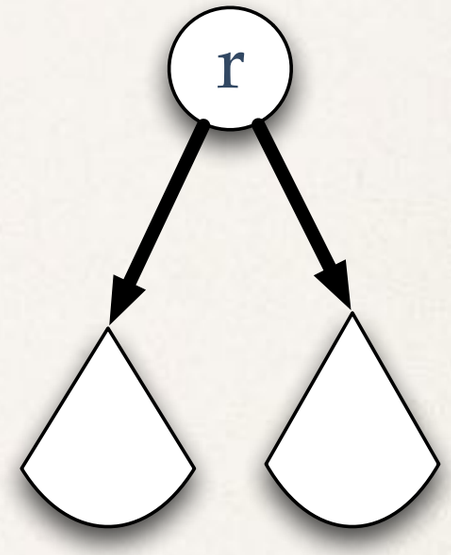
- Idea: *Take turns choosing the insertion subtree.*

Building balanced heaps

- Forcing a binary search tree to be balanced requires delicate algorithmic work.

- Heaps guarantee a weaker property: it's easy to construct a balanced heap.

- Idea: *Take turns choosing the insertion subtree.*



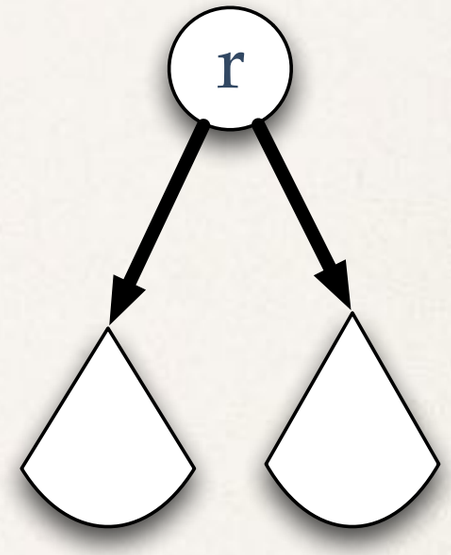
Building balanced heaps

- Forcing a binary search tree to be balanced requires delicate algorithmic work.

- Heaps guarantee a weaker property: it's easy to construct a balanced heap.

- Idea: *Take turns choosing the insertion subtree.*

Insert(s)



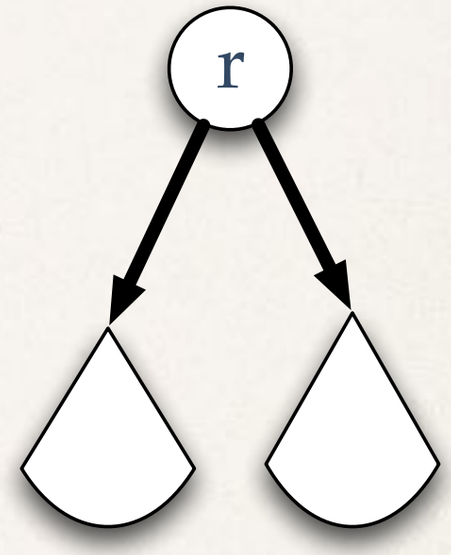
Building balanced heaps

- Forcing a binary search tree to be balanced requires delicate algorithmic work.

- Heaps guarantee a weaker property: it's easy to construct a balanced heap.

- Idea: *Take turns choosing the insertion subtree.*

Insert(s)



If s < r, the value s should occupy the root

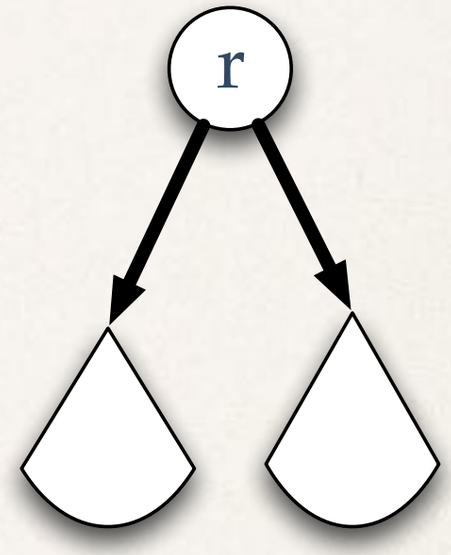
Building balanced heaps

- Forcing a binary search tree to be balanced requires delicate algorithmic work.

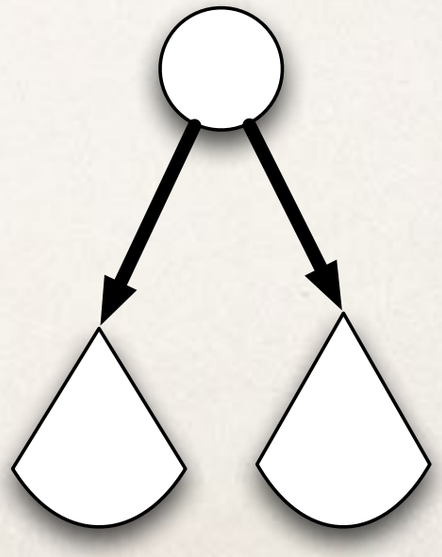
- Heaps guarantee a weaker property: it's easy to construct a balanced heap.

- Idea: *Take turns choosing the insertion subtree.*

Insert(s)



If s < r, the value s should occupy the root



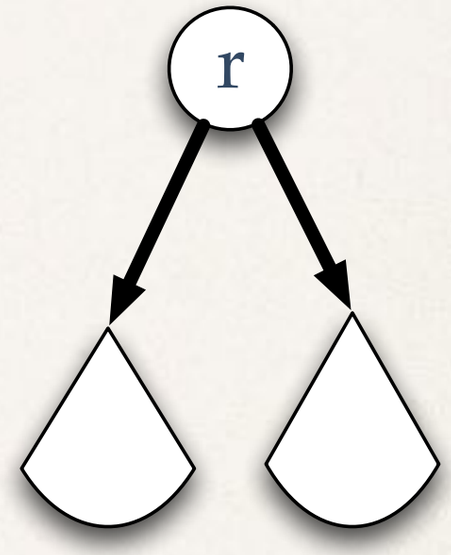
Building balanced heaps

- Forcing a binary search tree to be balanced requires delicate algorithmic work.

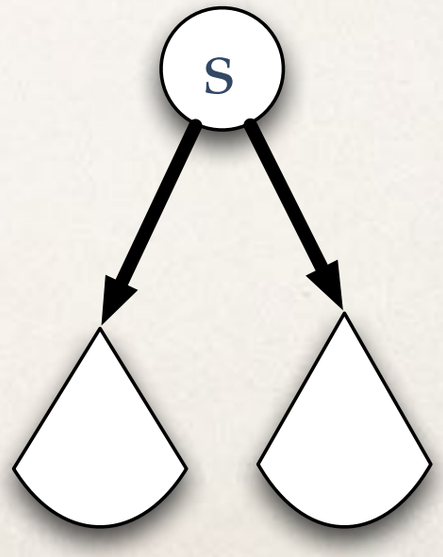
- Heaps guarantee a weaker property: it's easy to construct a balanced heap.

- Idea: *Take turns choosing the insertion subtree.*

Insert(s)



If s < r, the value s should occupy the root



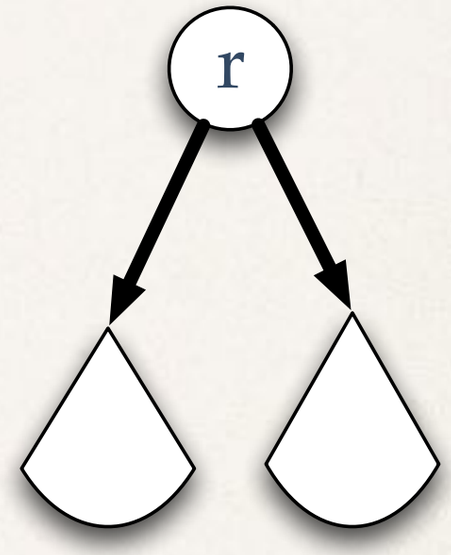
Building balanced heaps

- Forcing a binary search tree to be balanced requires delicate algorithmic work.

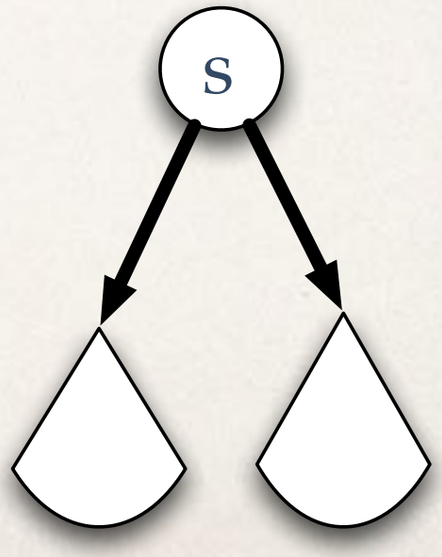
- Heaps guarantee a weaker property: it's easy to construct a balanced heap.

- Idea: *Take turns choosing the insertion subtree.*

Insert(s)



If s < r, the value s should occupy the root



Now, must insert r into (any) one of the subtrees...which one?

Alternating insertion

- In general, an insertion will require insertion into a subtree. However, *we have the luxury of choosing which tree*. If we alternate, half of the elements will be handed down to each child, and the result will be balanced!

- Implementation: Insert into the left tree; exchange them! Why does this maintain the heap property?

Problem: Insert s into a subtree

Insert into left tree

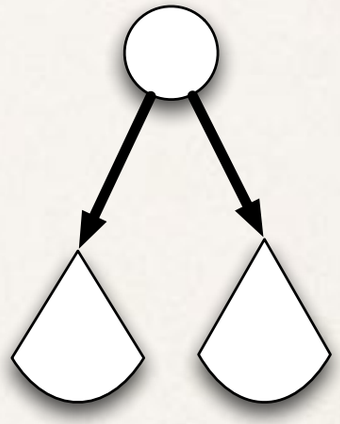
Swap

Alternating insertion

- In general, an insertion will require insertion into a subtree. However, *we have the luxury of choosing which tree*. If we alternate, half of the elements will be handed down to each child, and the result will be balanced!

- Implementation: Insert into the left tree; exchange them! Why does this maintain the heap property?

Problem: Insert s into a subtree



Insert into left tree

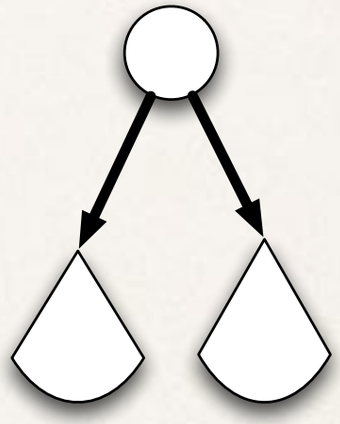
Swap

Alternating insertion

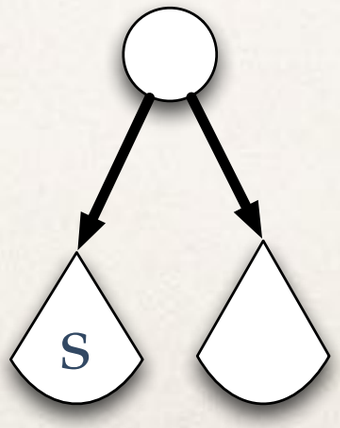
- In general, an insertion will require insertion into a subtree. However, *we have the luxury of choosing which tree*. If we alternate, half of the elements will be handed down to each child, and the result will be balanced!

- Implementation: Insert into the left tree; exchange them! Why does this maintain the heap property?

Problem: Insert s into a subtree



Insert into left tree



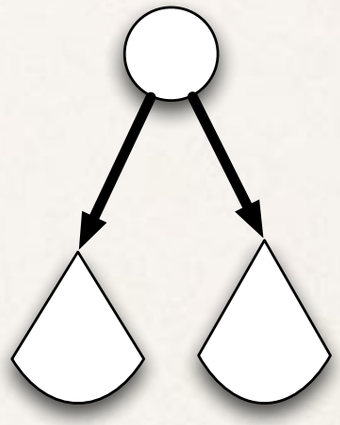
Swap

Alternating insertion

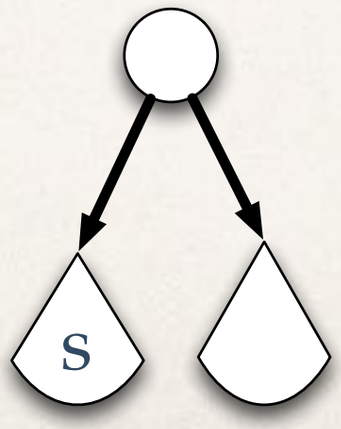
- In general, an insertion will require insertion into a subtree. However, *we have the luxury of choosing which tree*. If we alternate, half of the elements will be handed down to each child, and the result will be balanced!

- Implementation: Insert into the left tree; exchange them! Why does this maintain the heap property?

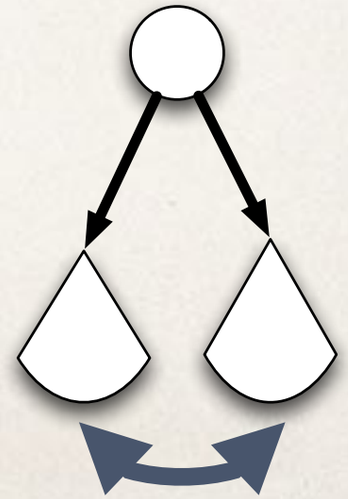
Problem: Insert s into a subtree



Insert into left tree



Swap

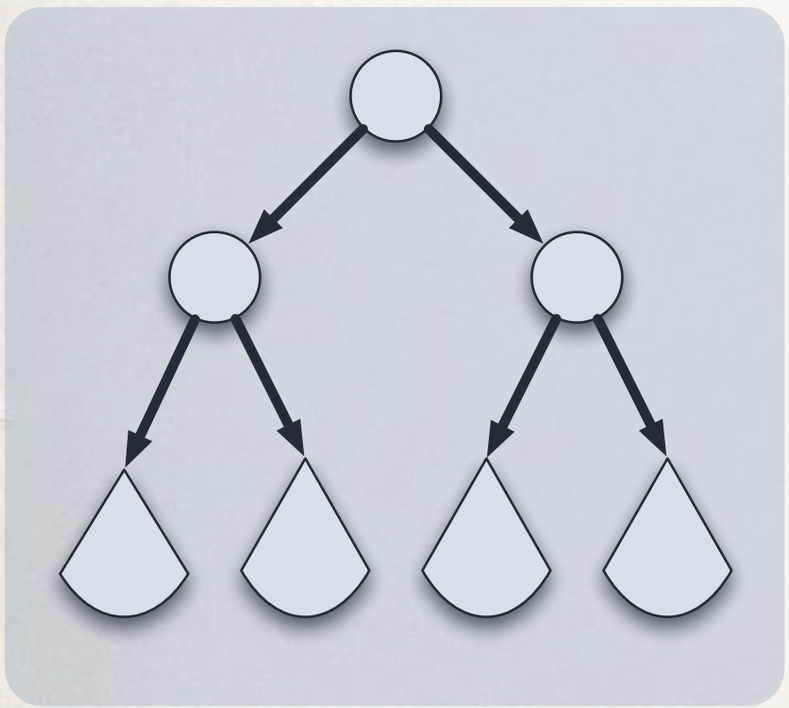


Removing the minimum element from a heap

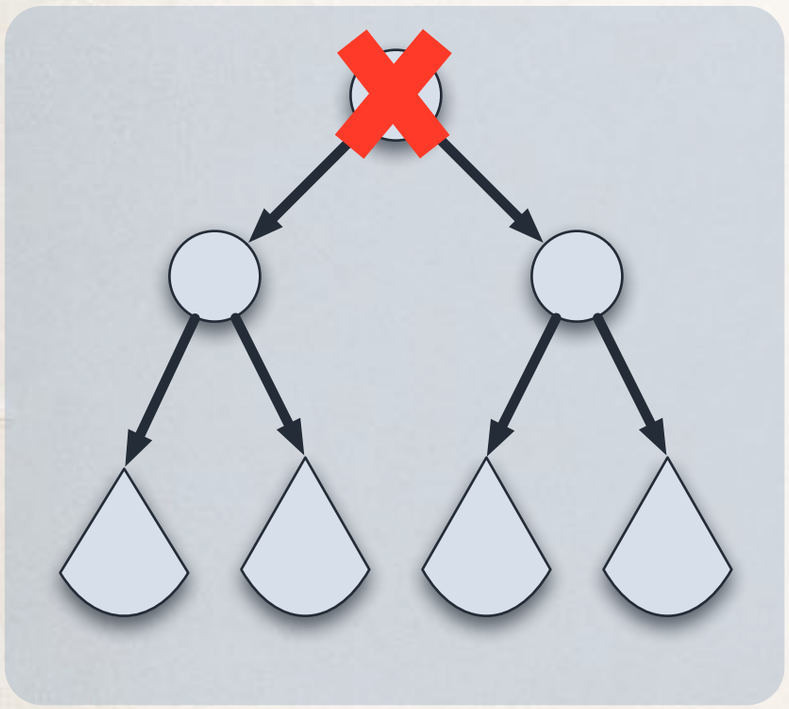
- Let min(H) denote the minimum element of a heap H.
- Find minimum value: **Findmin**(H). Easy! Return the value of the root.
- Remove minimum value: **Removemin**(H): Trickier.
  - Remove the top element. Two heaps result: H_1, H_2.
    Problem: How do you combine them into a single heap?
  - Suppose min(H_1) < min(H_2). Place min(H_1) at top; its children will be H_2 and **Removemin**(H_1). A nice recursive rule.
  - (Other cases? What if one of H_i is empty?)

Removemin, in pictures

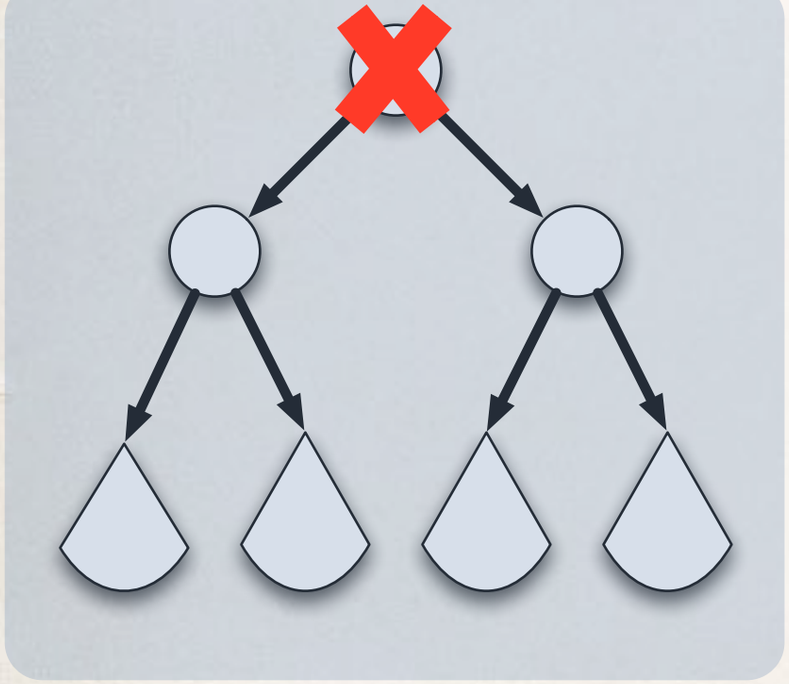
Removemin, in pictures

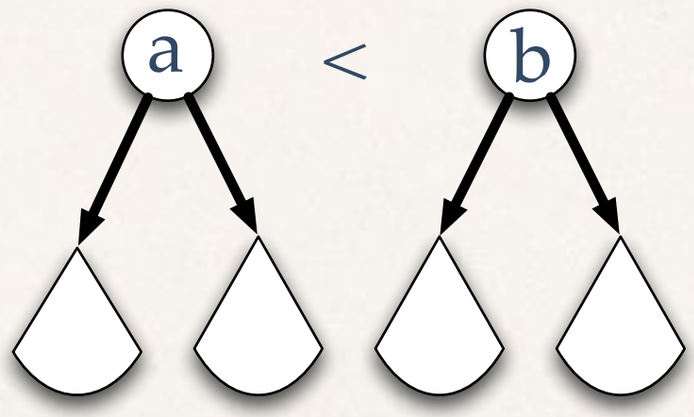


Removemin, in pictures

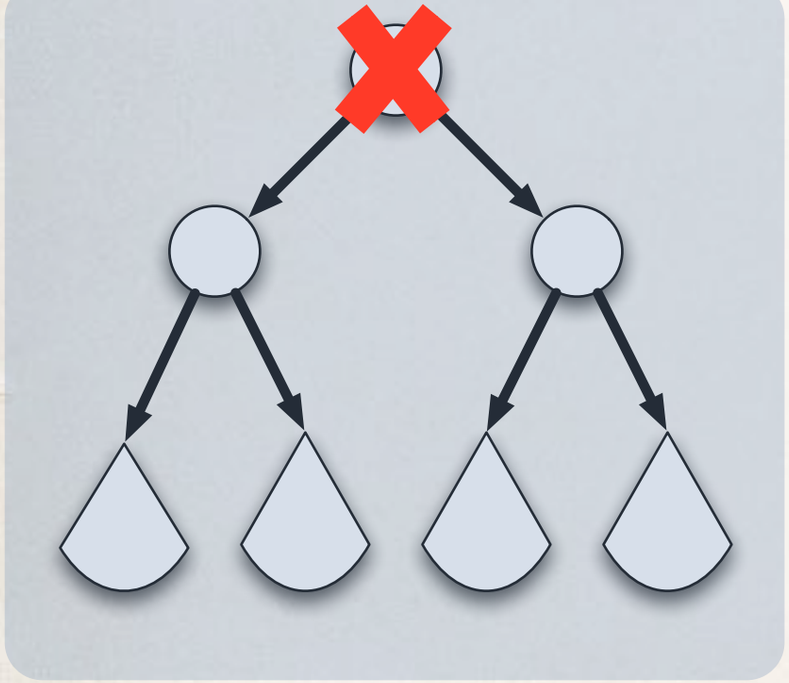


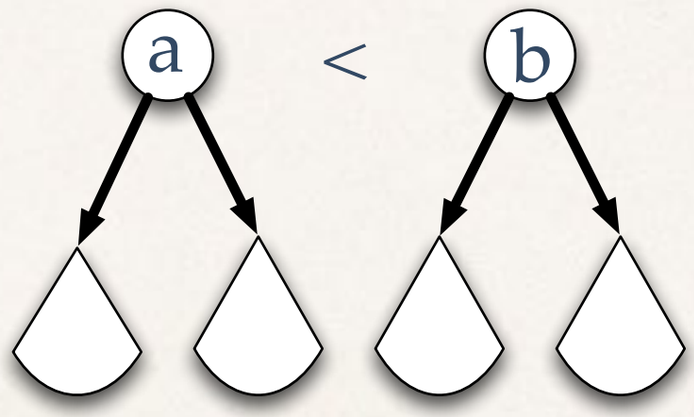
Removemin, in pictures

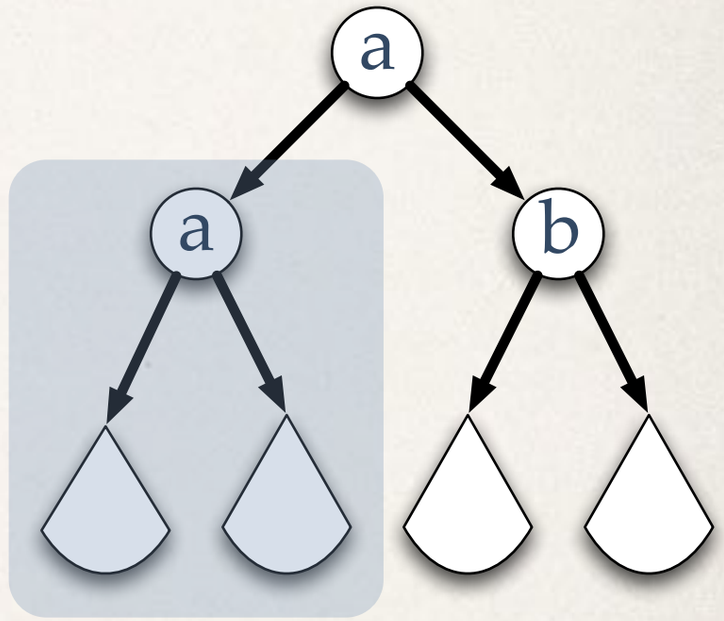




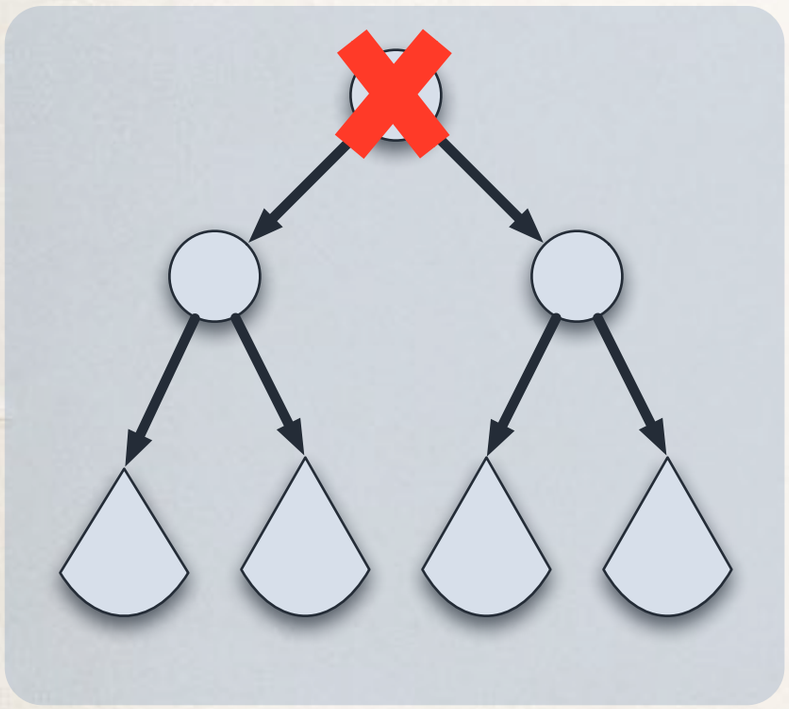
Removemin, in pictures

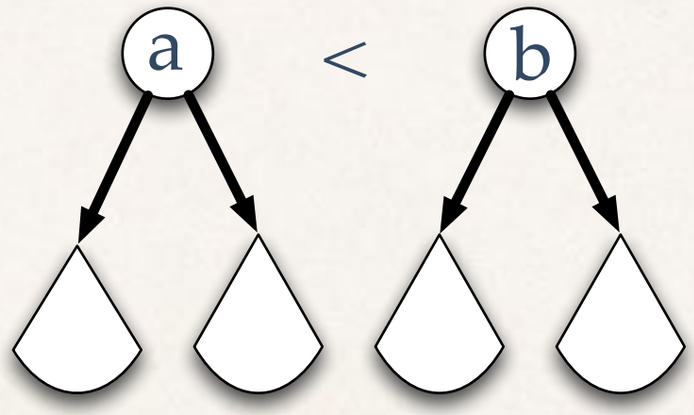


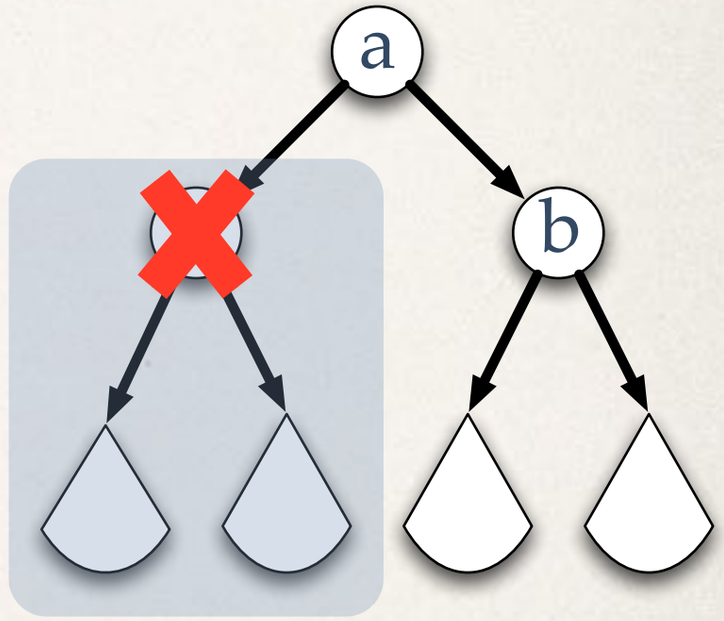




Removemin, in pictures





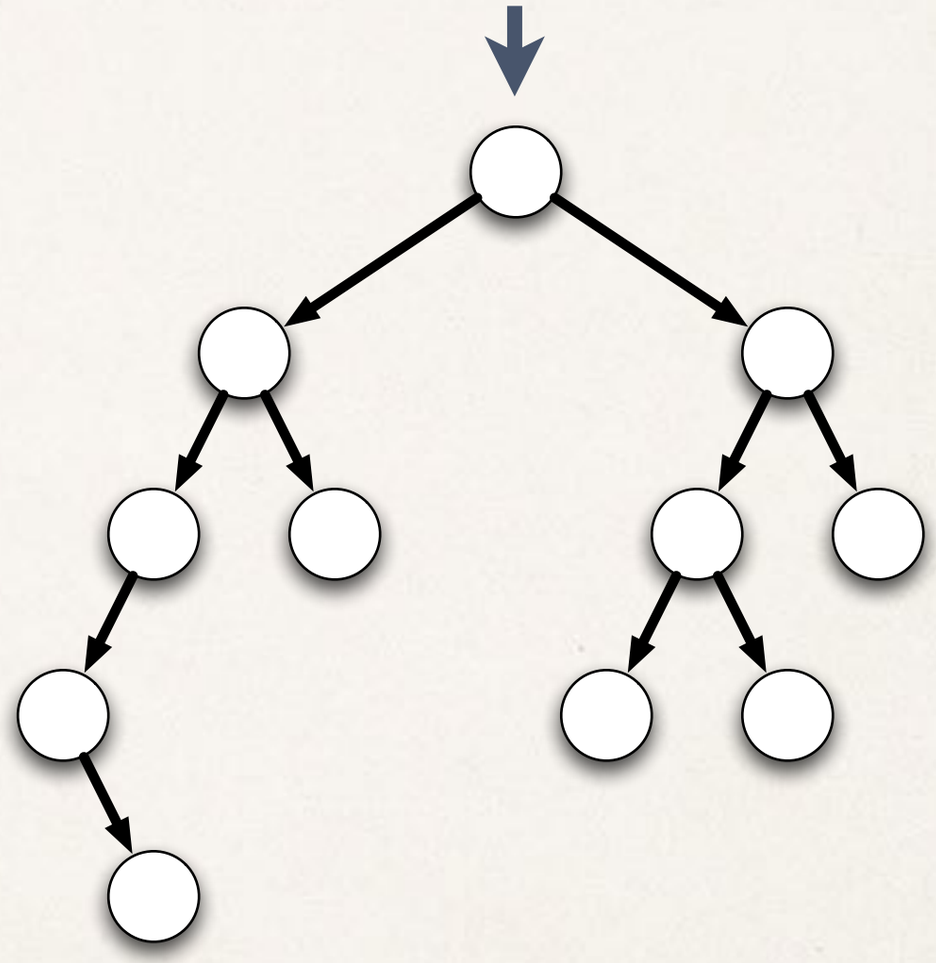


Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.

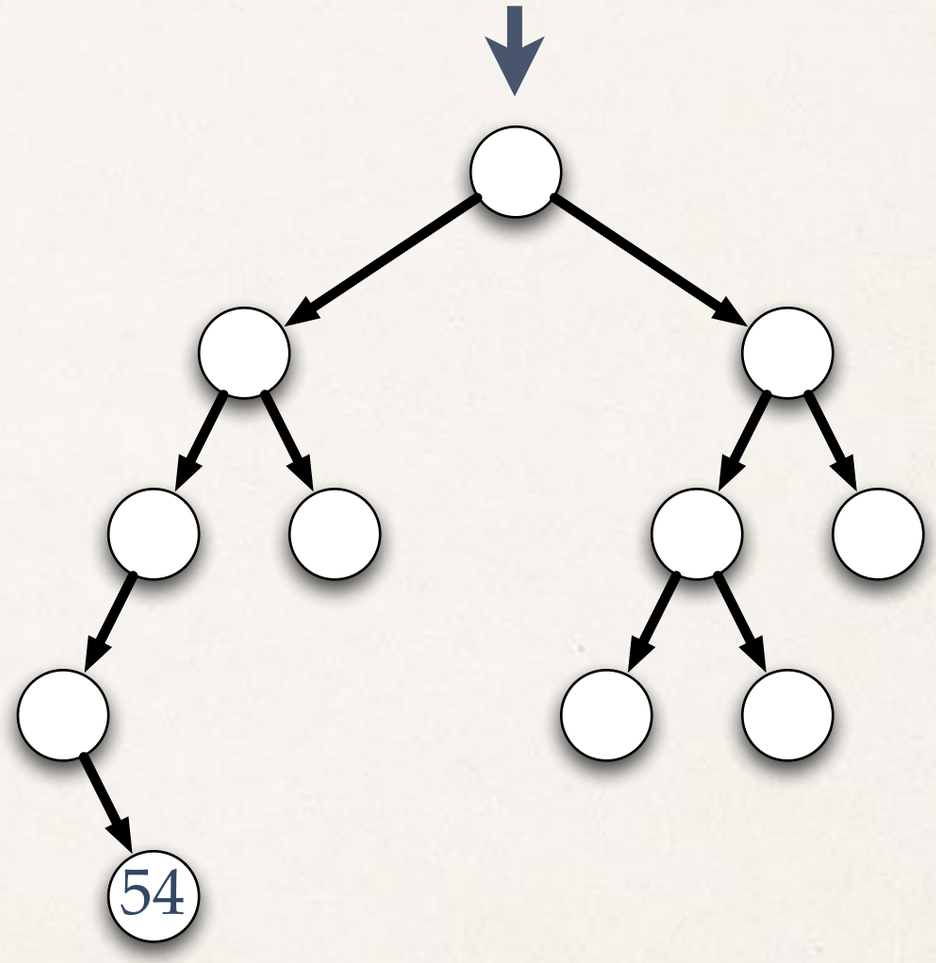
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



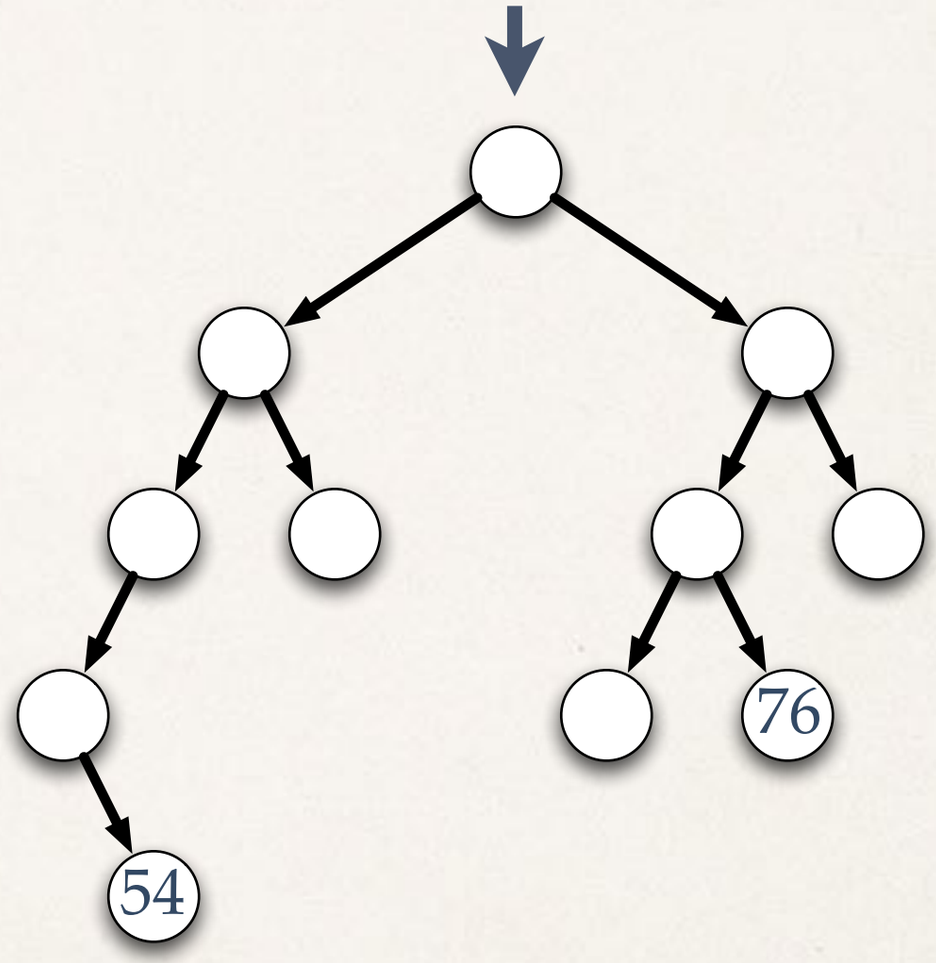
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



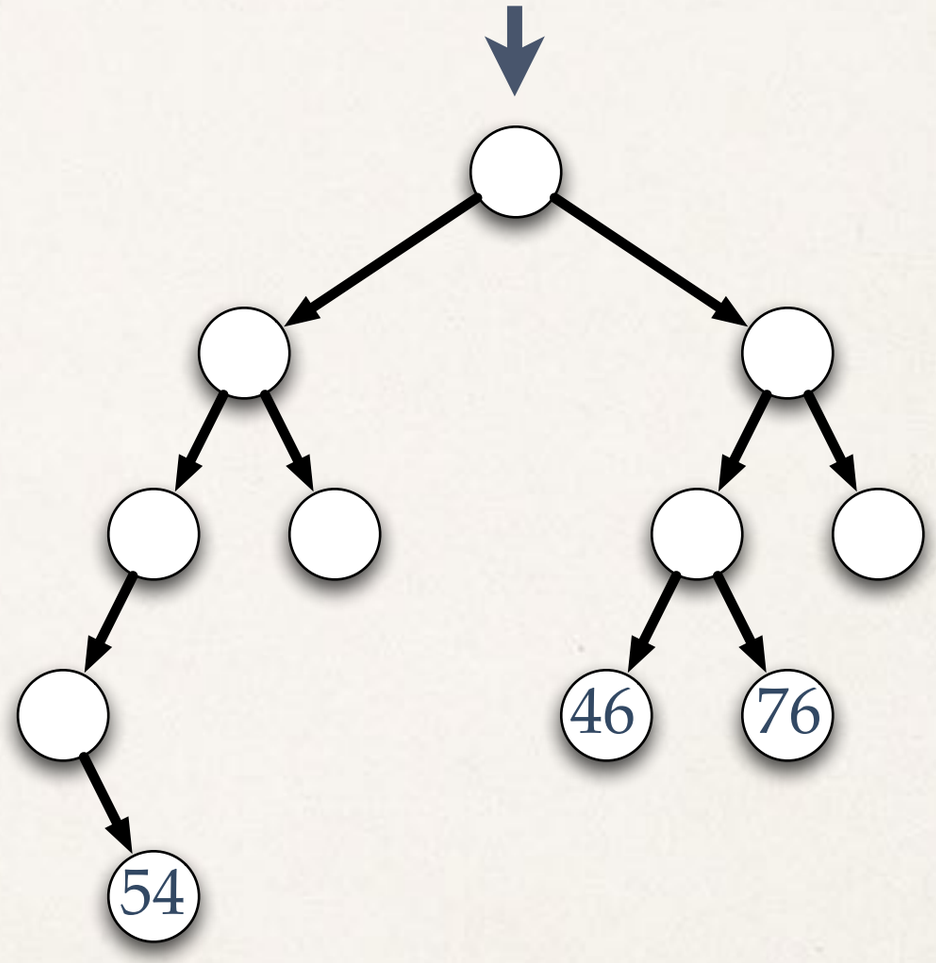
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



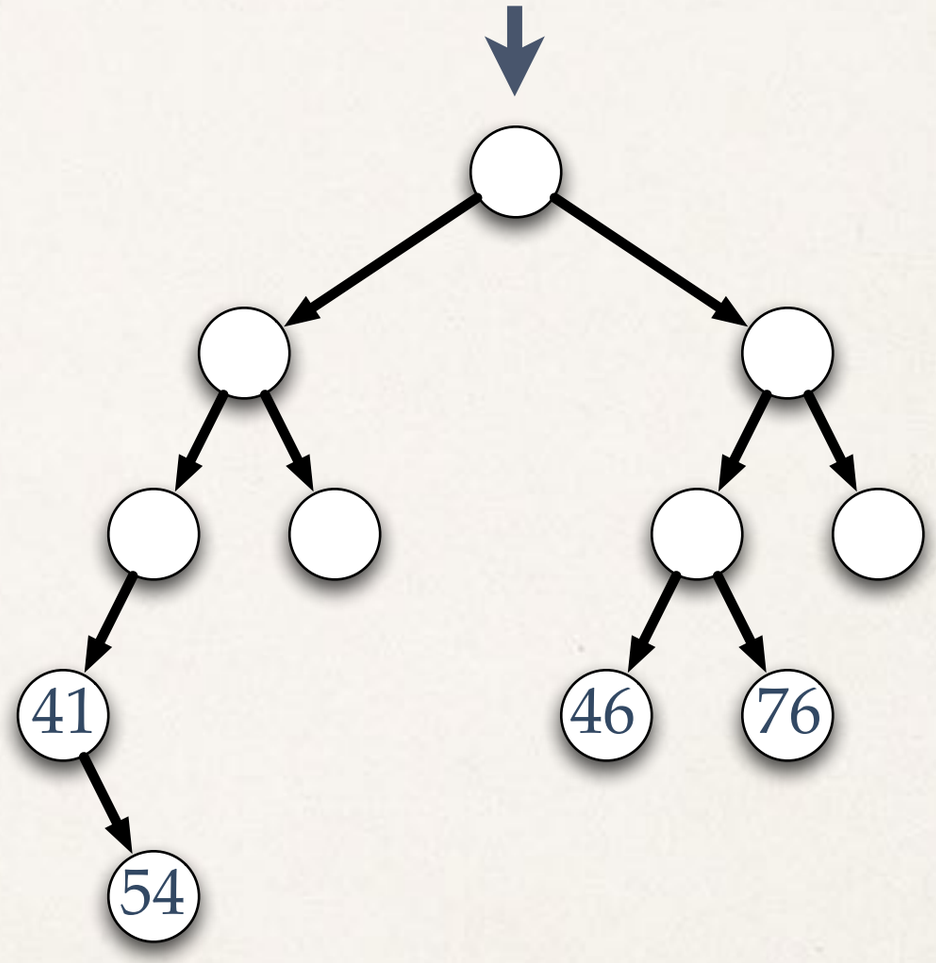
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



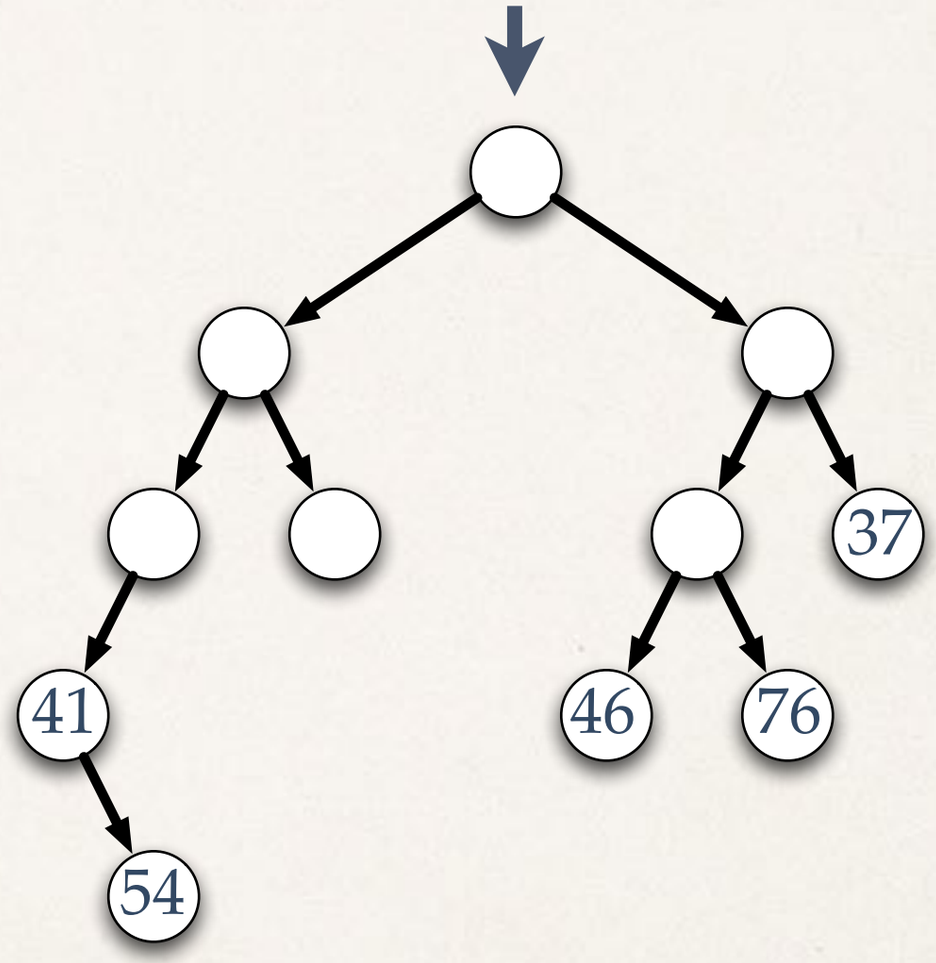
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



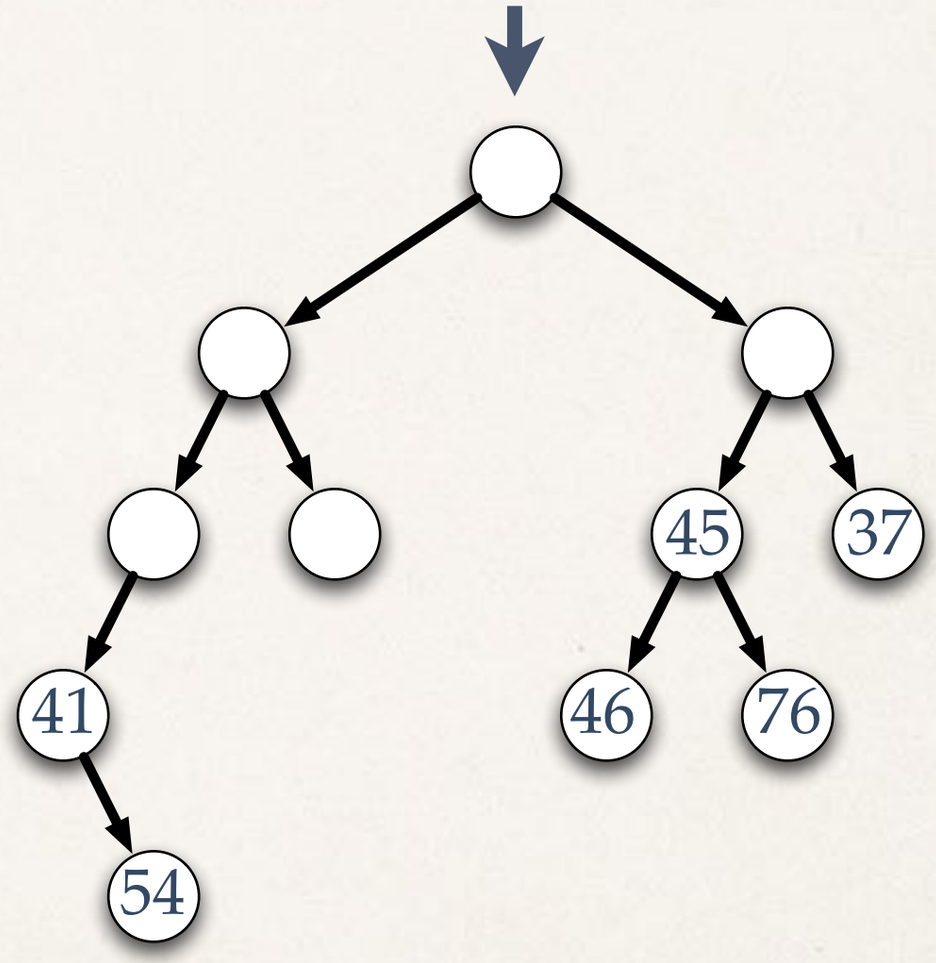
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



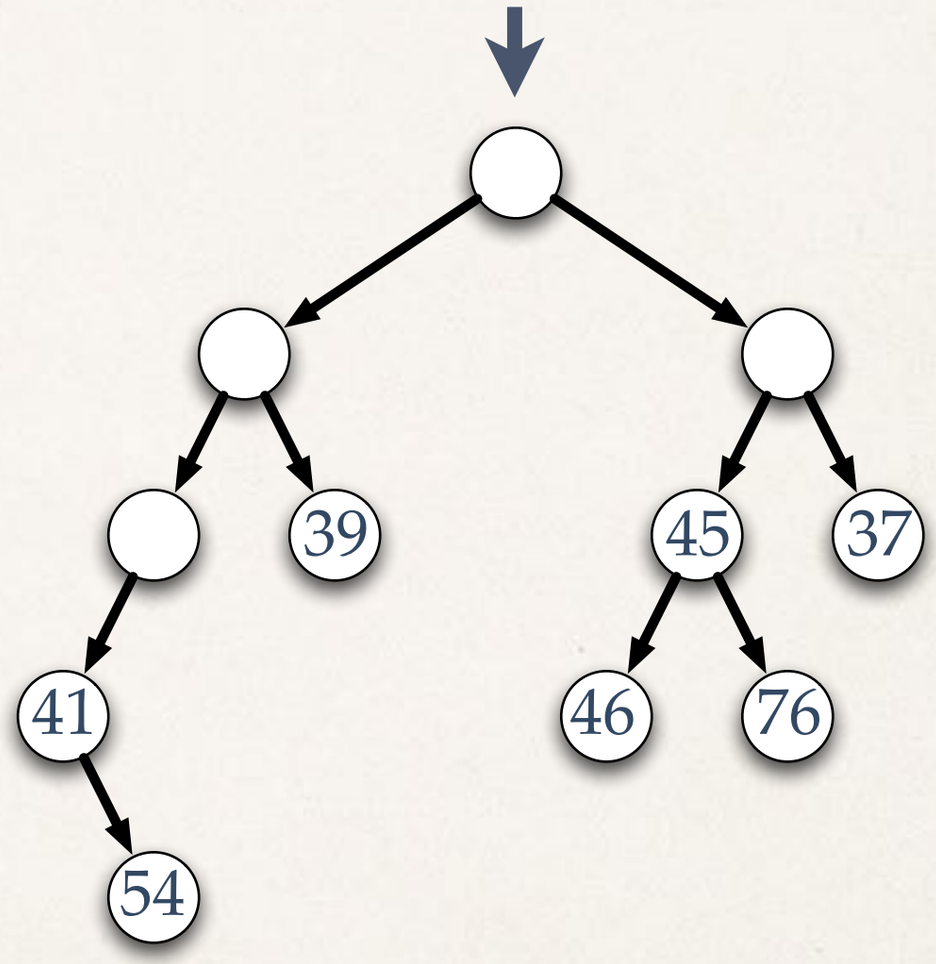
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



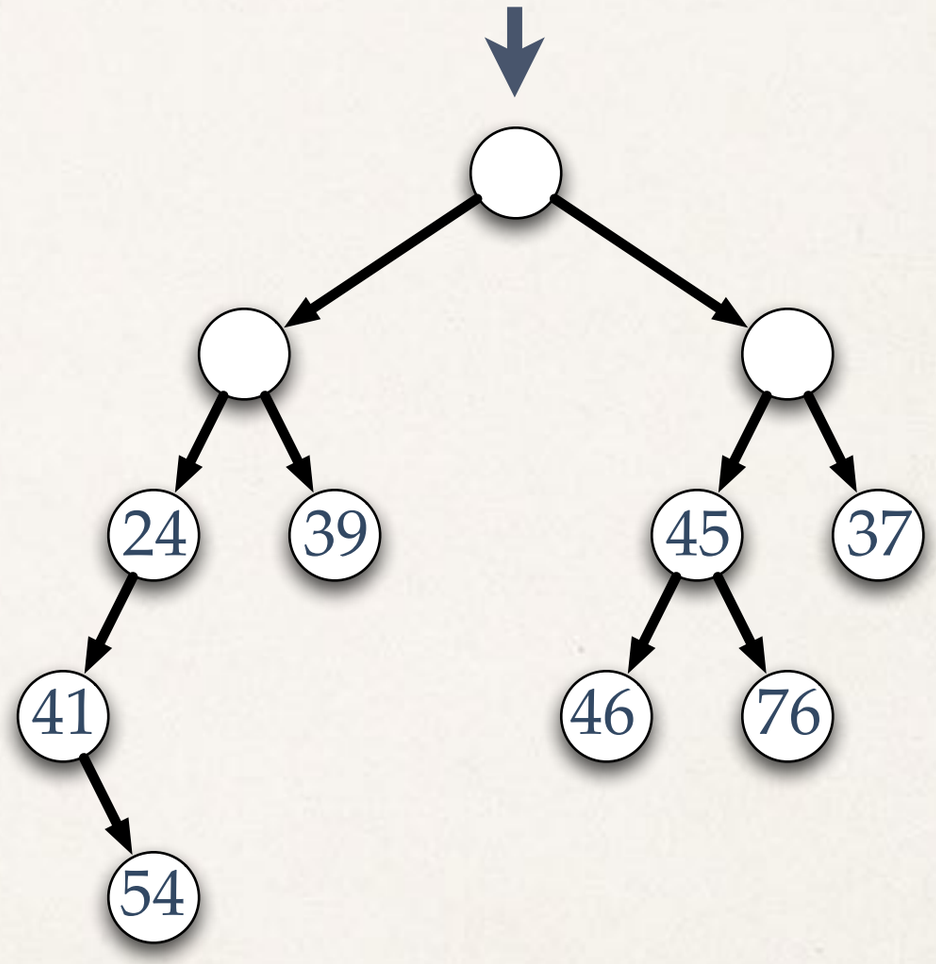
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



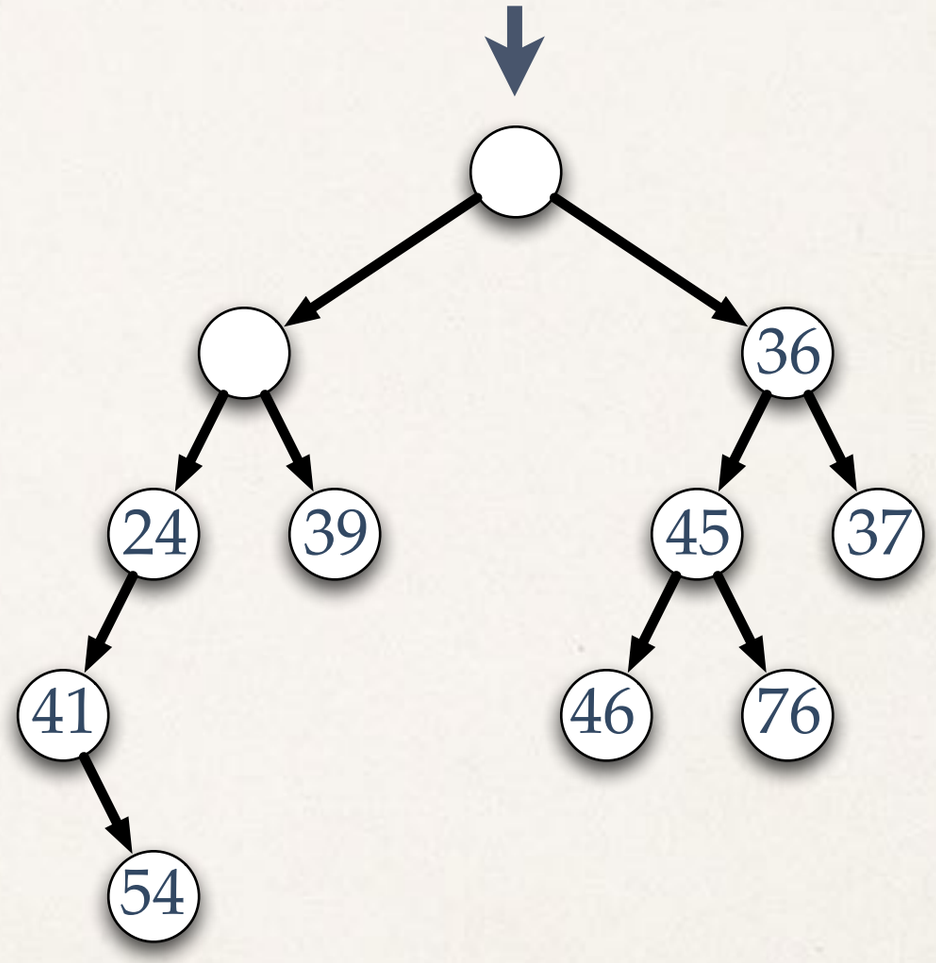
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



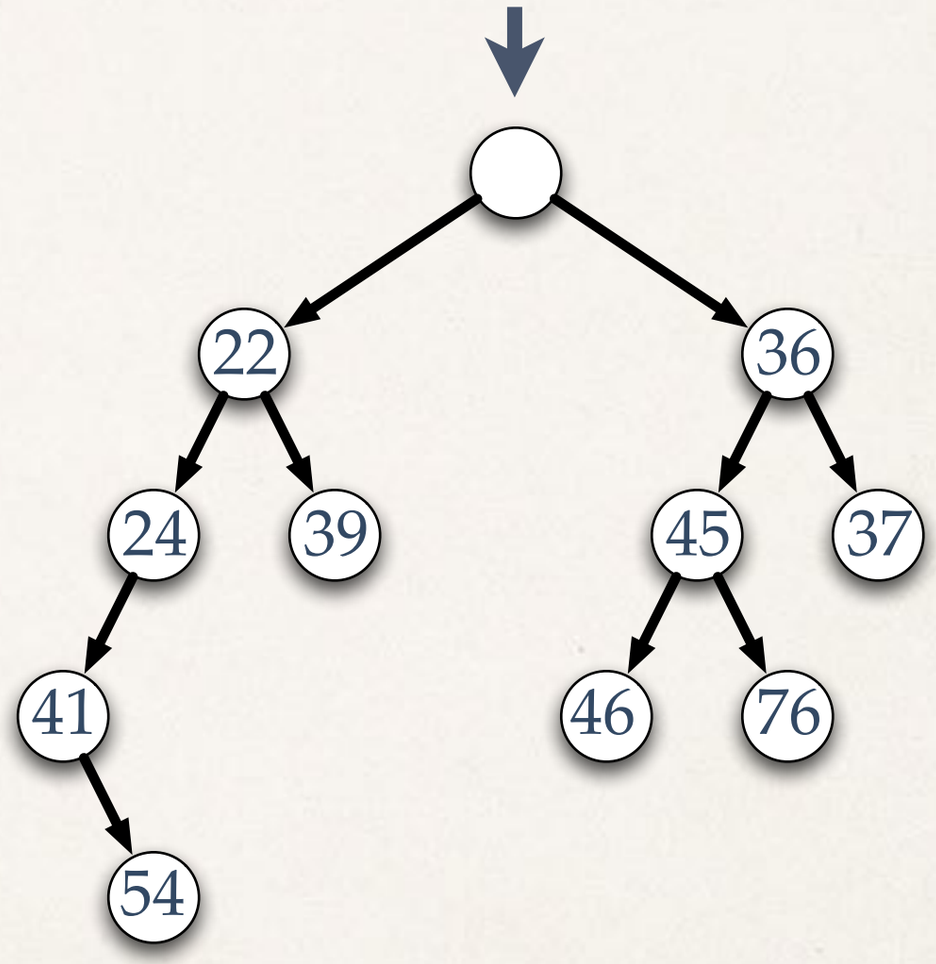
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



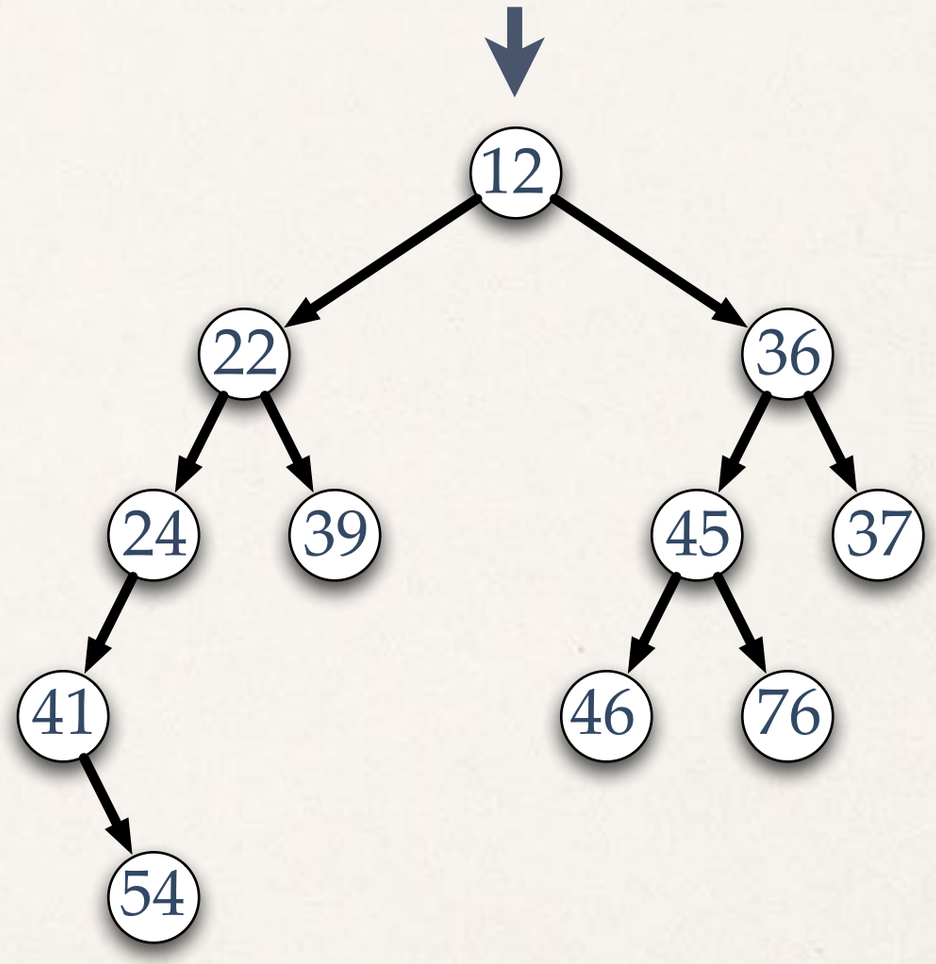
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



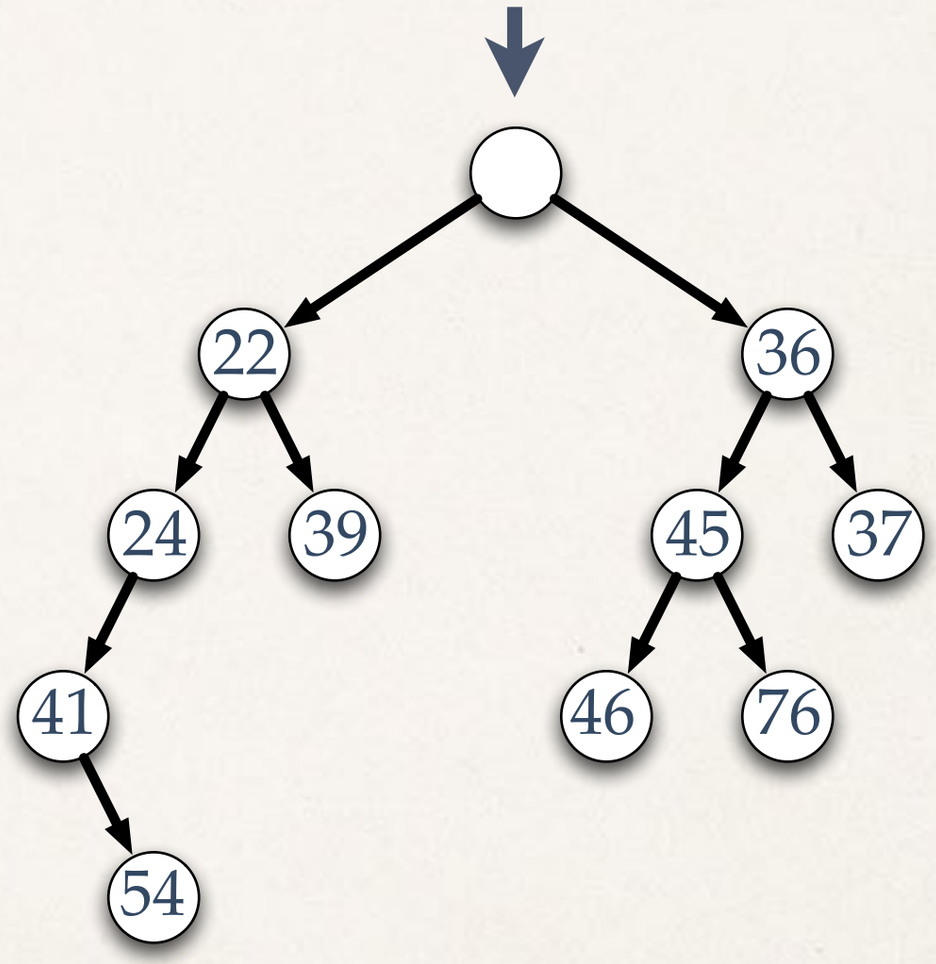
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



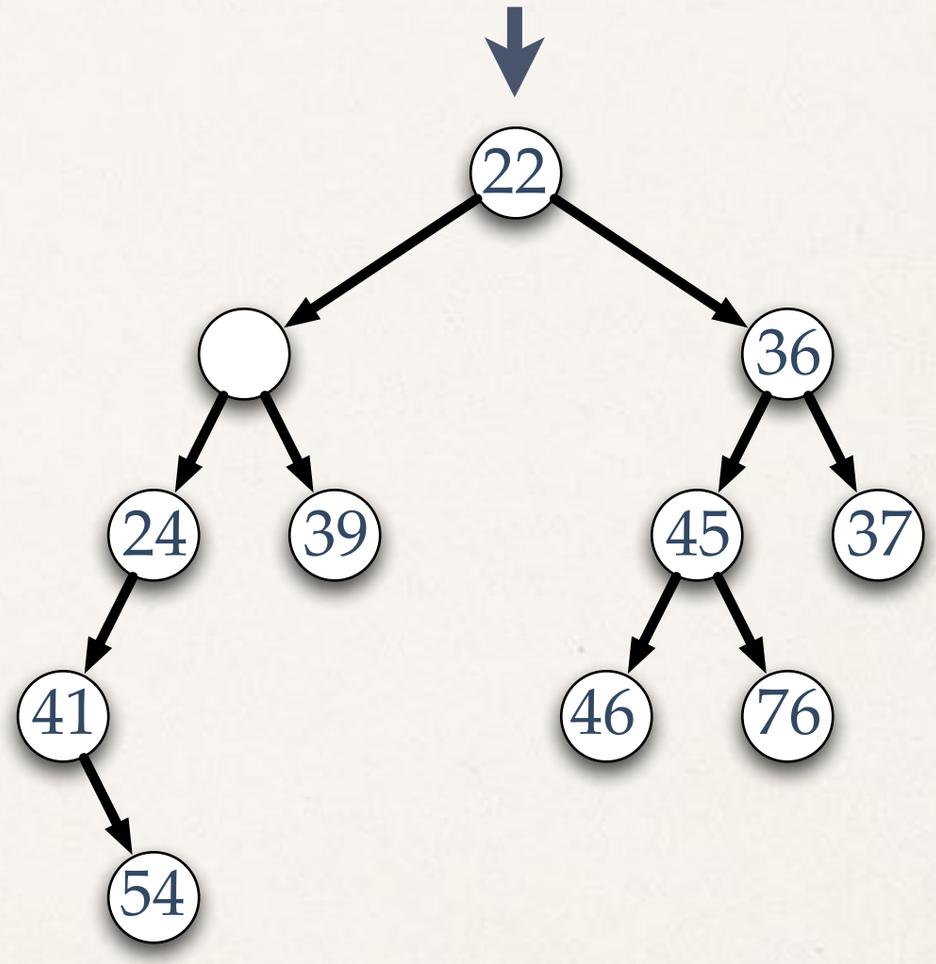
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



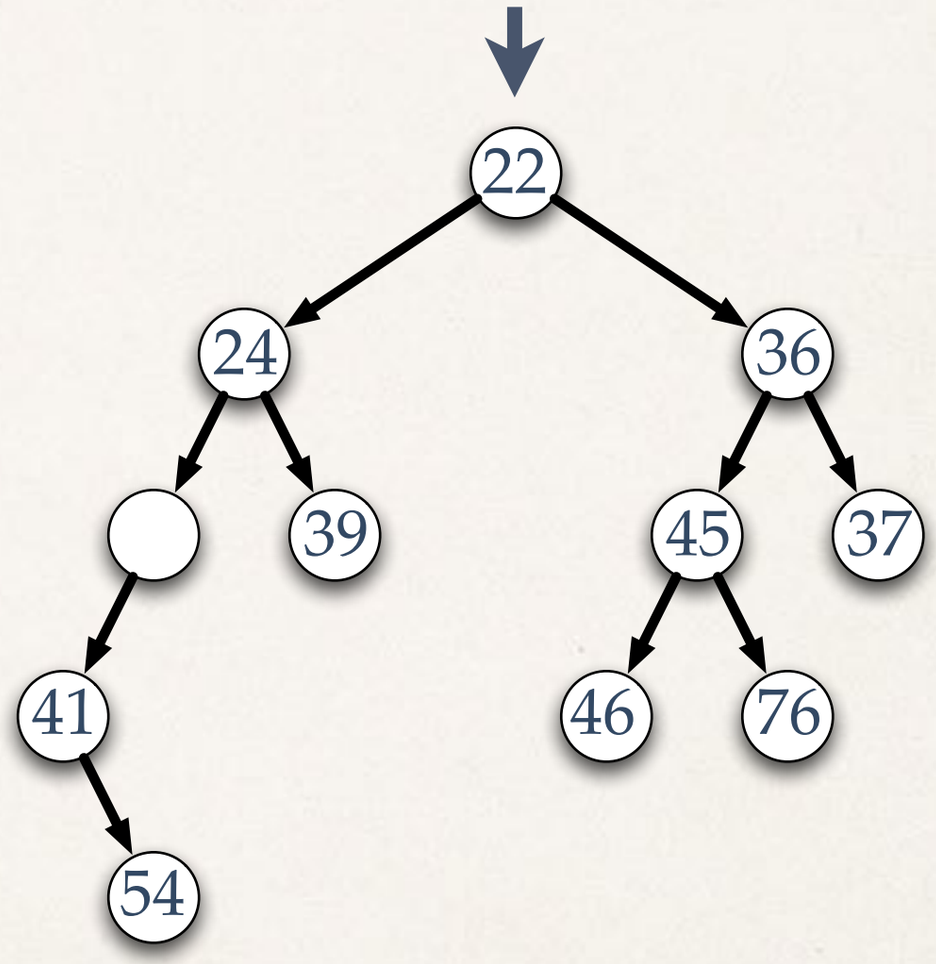
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



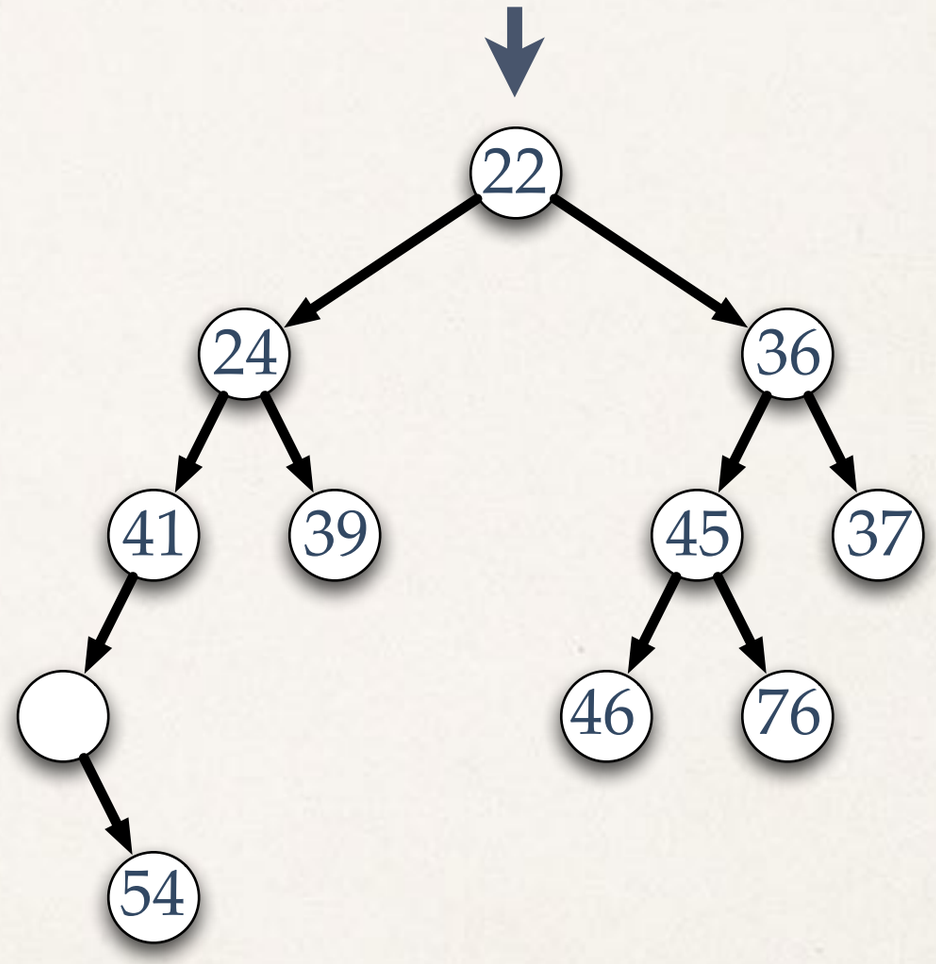
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



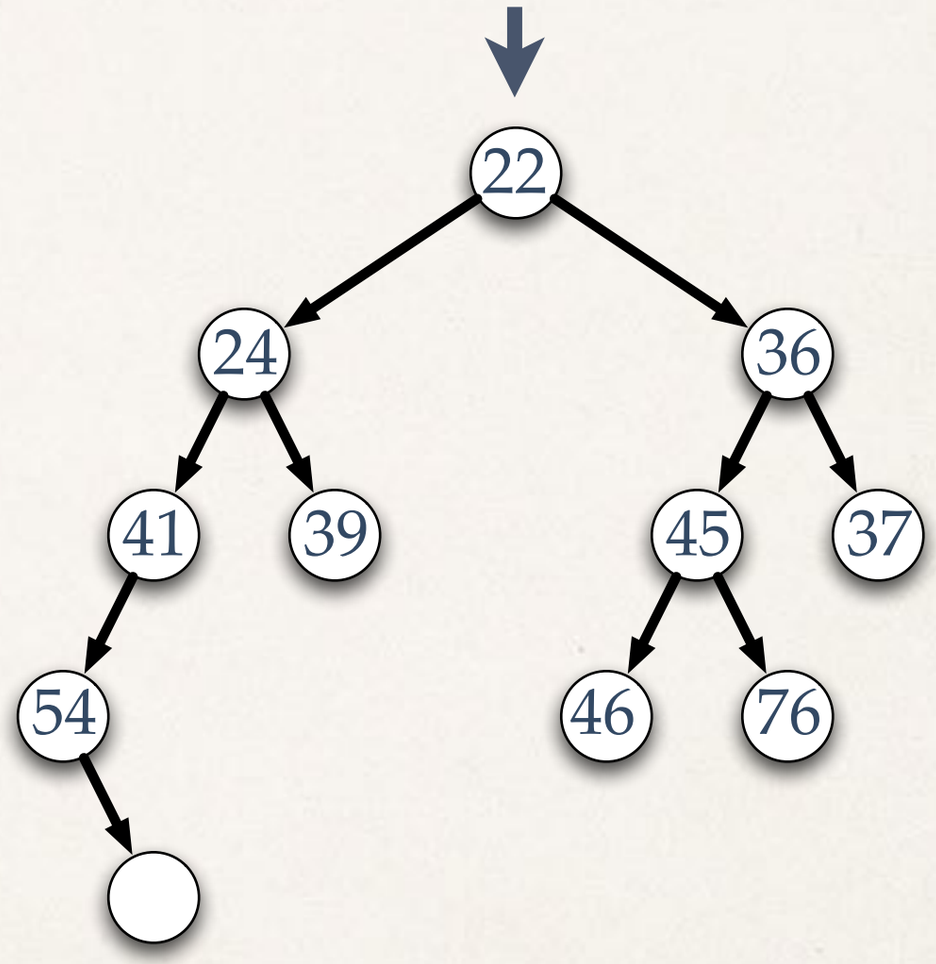
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



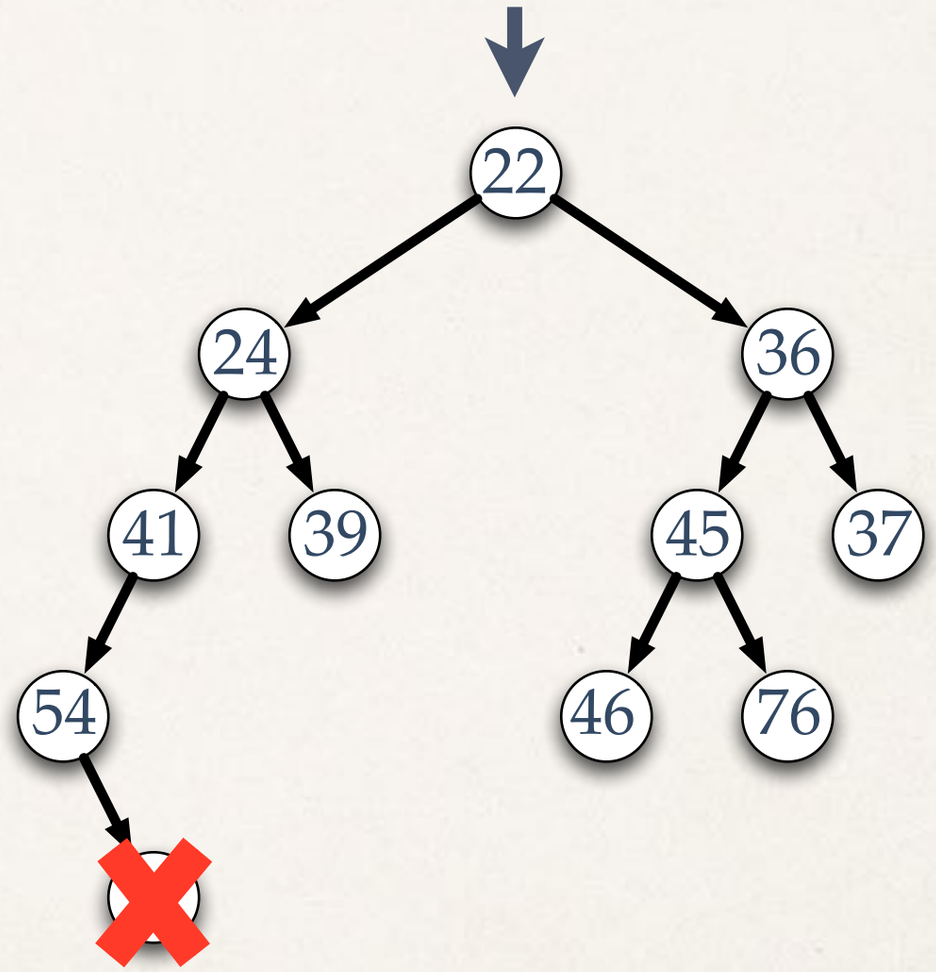
Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



Removing the minimum element: Another view

- Remove the smallest element.
- *Promote* its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



Implementing Heaps in SCHEME

- As with trees, we represent a heap as a list containing a value, and two heaps (the "left subheap" and the "right subheap").

```
(define (create-heap v H1 H2)
  (list v H1 H2))
```

- Helper functions for accessing the value and the subheaps:

```
(define (h-min H) (car H))
(define (left H)  (cadr H))
(define (right H) (caddr H))
```

Insertion

- **Note how the insert & exchange trick works**: We *alternate* insertion into subheaps. To implement this: always insert into left subheap, then exchange them.

```
(define (insert x H)
  (if (null? H)
      (create-heap x '() '())
      (let ((child-value (max x (h-min H)))
            (root-value  (min x (h-min H))))
        (create-heap root-value
                     (right H)
                     (insert child-value (left H))))))
```

Remove Min in SCHEME

- The real problem that emerges in **remove-min** is to combine the two subheaps into a single heap:

```
(define (combine-heaps H1 H2)
  (cond ((null? H1) H2)
        ((null? H2) H1)
        ((< (h-min H1) (h-min H2))
         (create-heap (h-min H1)
                      H2
                      (combine-heaps (left H1) (right H1))))
        (else
         (create-heap (h-min H2)
                      H1
                      (combine-heaps (left H2) (right H2))))))
```

- Then:

```
(define (remove-minimum H)
  (combine-heaps (left H) (right H)))
```

Information Coding on a modern computer

- All data in a computer is maintained in memory cells that can hold one "bit" of data: a 0 or a 1. Why?

- Thus letters are usually encoded with 8 bits (for 256 combinations). You could get by with less--say 5, but 8 is convenient for other reasons.

Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k

An application of trees: Huffman coding

- Consider the problem of digitally representing documents in English. One straightforward approach: each letter is represented by an 8-bit sequence. (This gives 256 “letters,” enough for basic text.)

- Now consider the natural data compression problem: We’d like to store documents using the least number of bits.

- If we have no information about the documents, there’s nothing you can do.

- However, in English text, for example, the letter “a” is far more common than the letter “z”. Perhaps we can exploit this by coding “a” using a shorter bit string, and “make up for this” by using a longer bit string for “z”?

Variable-length encoding

- This suggests a variable-length encoding. “Frequent” letters should get short encodings. Good idea!
- Problem: Some care is necessary…how do we “decode”? If
  - 0 represents a
  - 1 represents e
  - 01 represents c

we don’t know how to decode “01”!

Prefix-free codes

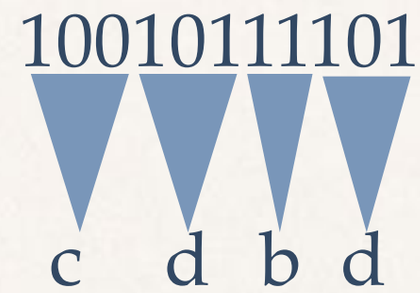
- Solution: *Prefix-free encoding*--no "codeword" is a prefix of any other codeword.

- Scanning an encoded string left-to-right, one can decode in a unique fashion. Why?

- Once a codeword is observed in the sequence, it cannot be the prefix of any other codeword; then you can decode with confidence!

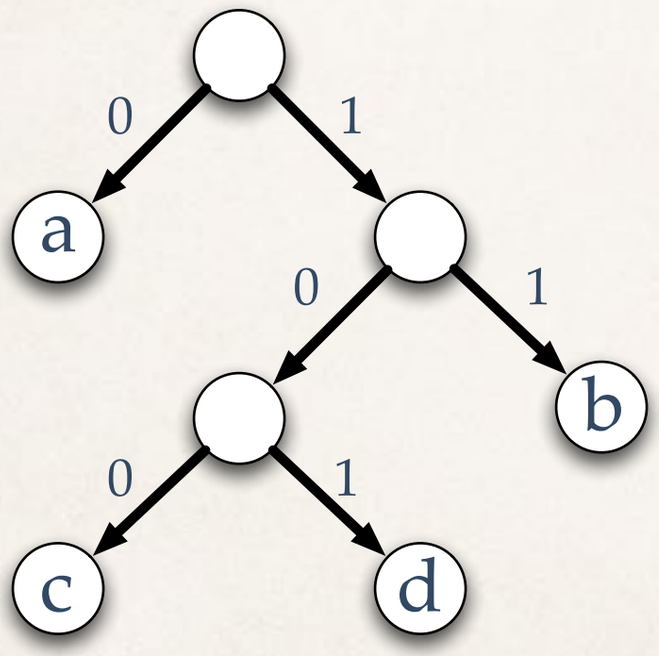
A prefix free code in action

- 0 represents a
- 11 represents b
- 100 represents c
- 101 represents d



Prefix-free codes can be represented as trees

- One natural way to represent a prefix-free code is as a binary tree where leaves are labelled with alphabet symbols.



Moving left in the tree corresponds to a 0; moving right a 1.

- 0 represents a
- 11 represents b
- 100 represents c
- 101 represents d

Why is such a "tree encoding" prefix-free?

Given frequencies...
we want the best tree

- Consider frequencies for English: “a” appears 8.1% of the time, “b” 1.4%, ...

- The best code will minimize the length of a document which exactly realizes these percentages: equivalently, it minimizes the AVERAGE number of bits used, when a letter is drawn from this distribution.

- Problem: Find the code that minimizes

$$\sum_{\text{letters } \ell} \text{codelength}(\ell) \cdot \text{frequency}(\ell)$$

Huffman's idea

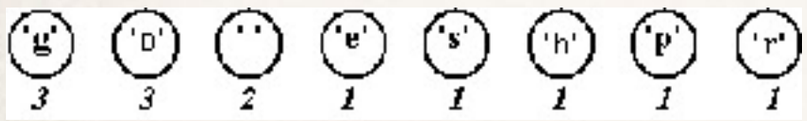
- Begin with each letter in its own tree. The *weight* of the tree is the frequency of the letter.

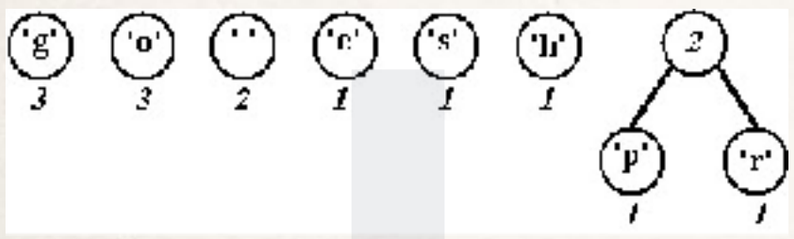
- Find the two lightest trees. **Join** them with a root above; new weight is their sum.

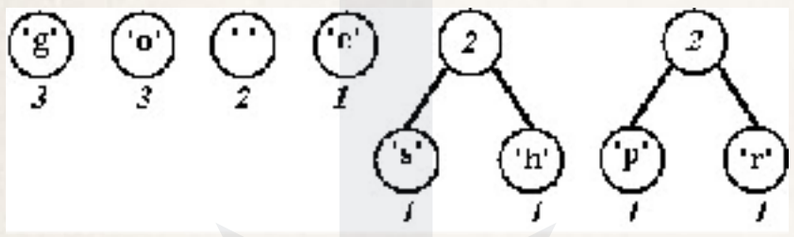
- Repeat until a single tree emerges.

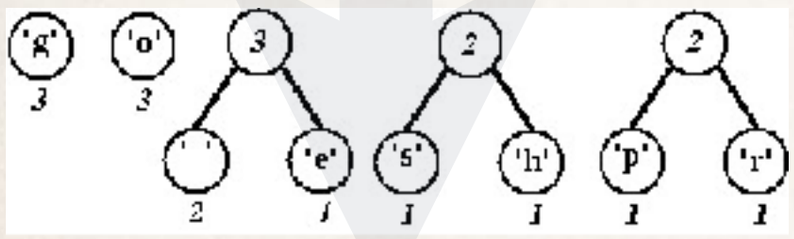
- Huffman proved that this tree is **optimal**: it generates the shortest average codeword over all trees. (Note that it depends on the distribution of the input symbol.)

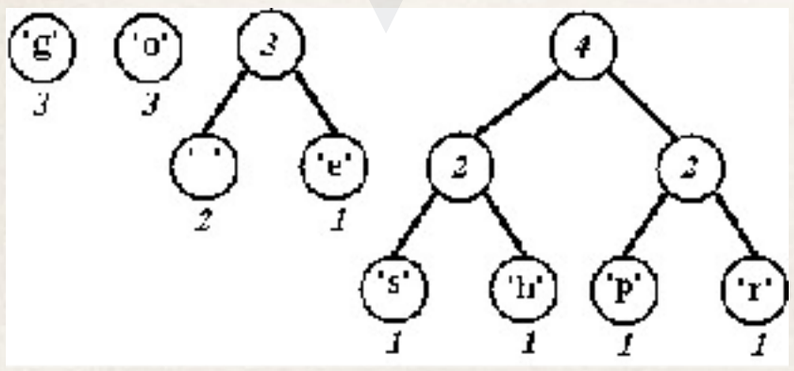
An example

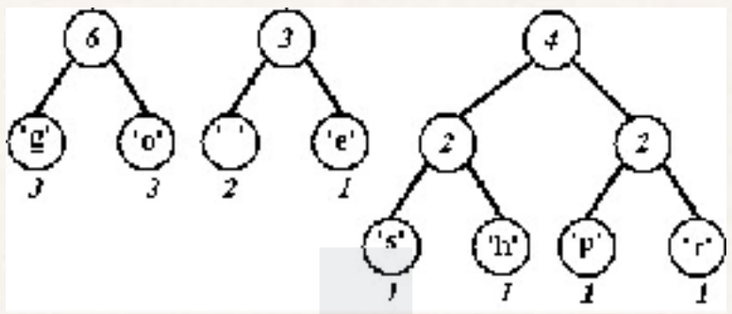

'g' 3
'o' 3
' ' 2
'e' 1
's' 1
'h' 1
'p' 1
'r' 1

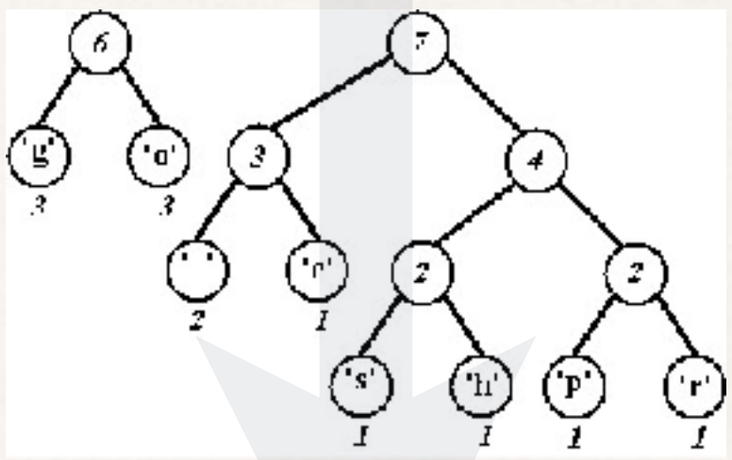

'g' 3
'o' 3
' ' 2
'e' 1
's' 1
'h' 1
2
'p' 1
'r' 1

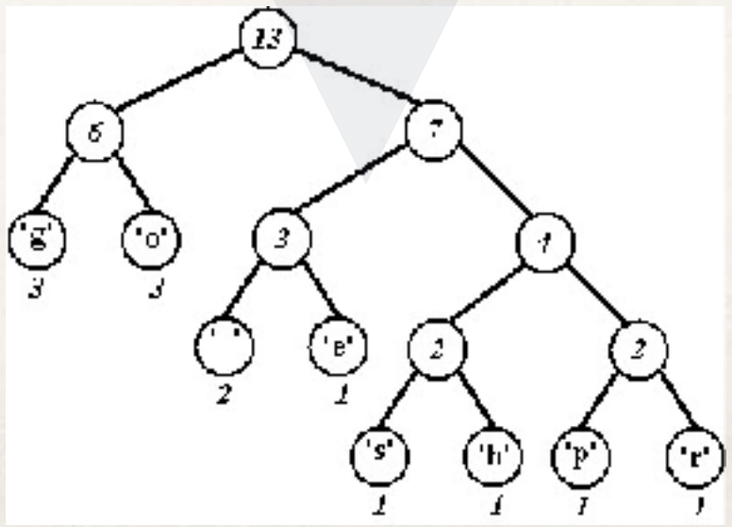

'g' 3
'o' 3
' ' 2
'e' 1
2
's' 1
'h' 1
2
'p' 1
'r' 1


'g' 3
'o' 3
3
' ' 2
'e' 1
2
's' 1
'h' 1
2
'p' 1
'r' 1


'g' 3
'o' 3
3
' ' 2
'e' 1
4
2
's' 1
'h' 1
3
'p' 1
'r' 1


6
'g' 3
'o' 3
3
' ' 2
'e' 1
4
2
's' 1
'h' 1
2
'p' 1
'r' 1


6
'g' 3
'o' 3
7
3
' ' 2
'e' 1
4
2
's' 1
'h' 1
2
'p' 1
'r' 1


13
6
'g' 3
'o' 3
7
3
' ' 2
'e' 1
4
2
's' 1
'h' 1
2
'p' 1
'r' 1

The effects?

- A Huffman code for English can achieve about 4.22 bits per character.
- ASCII (a standard coding) uses 8 bits per character.
- A straightforward "flat" code uses 5 bits per character.
- You'll have a chance to develop the Huffman coding algorithm in a future problem set.

Tries (or Prefix Trees)

- We have focused on binary trees, where each node has 0, 1, or 2 children.

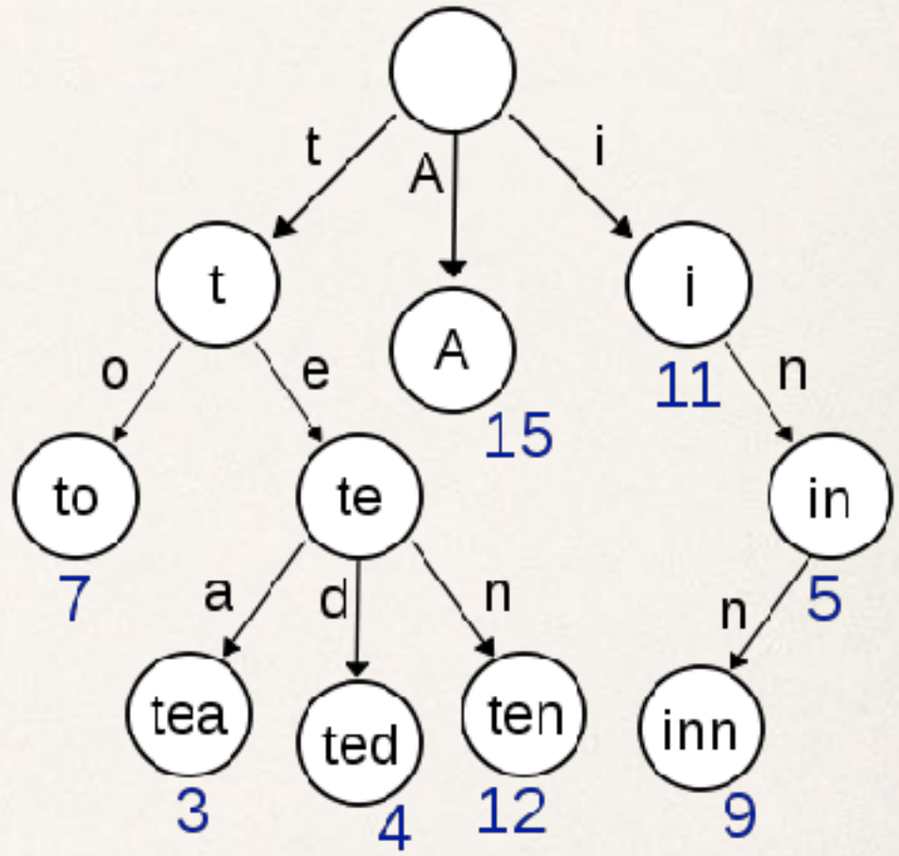
- In principle, you can operate with trees of various branching factors.

- As an example, we consider *tries*, also called *prefix trees*, a data structure often used to maintain dictionary entries of fixed length.

- Trees are another data structure that can implement the SET datatype. (It can handle **insert** and **ismember?**.)

Tries: examples, an informal definition

- Idea: Suppose set items are *words* over an alphabet.
  - English words are over the alphabet {a, b, c, ..., z, A, B, ..., Z}
  - Positive integers are words over the alphabet {0, 1, 2, ...,9}
- In a trie over the alphabet A, each node may have up to |A| children.
- Words are "stored" at the end of the path they index. Thus, each leaf corresponds to a stored element.



A trie over the English alphabet containing **A**, **to**, **tea**, **ted**, **ten**, and **inn**.

The cost of trie operations

- Roughly, the time taken to search a trie (answer a **ismember?** query) or insert grows linearly with the length of the key.

- This can be much worse than a binary search tree!

- However, for large sets of data of roughly the same length (e.g., an English dictionary), this can be a good solution. Note that no "balancing" is required (cf. our discussion of search trees).

Implementing the Trie

- Imagine a trie for storing large numbers.
- One natural way to implement a node:

`(trie0 trie1 ... trie9)`

- Each node must store a (sub)trie for each numeral {0, 1, ..., 9}. Note that tries are often very *sparse*: a given node may often have only a few nonempty subtries. For this reason, we may choose the following implementation:

`((0 . trie0) (1 . trie1) ... (9 . trie9))`

where the subtries are left out if they are empty.

An example with this implementation

- Then, the top node:

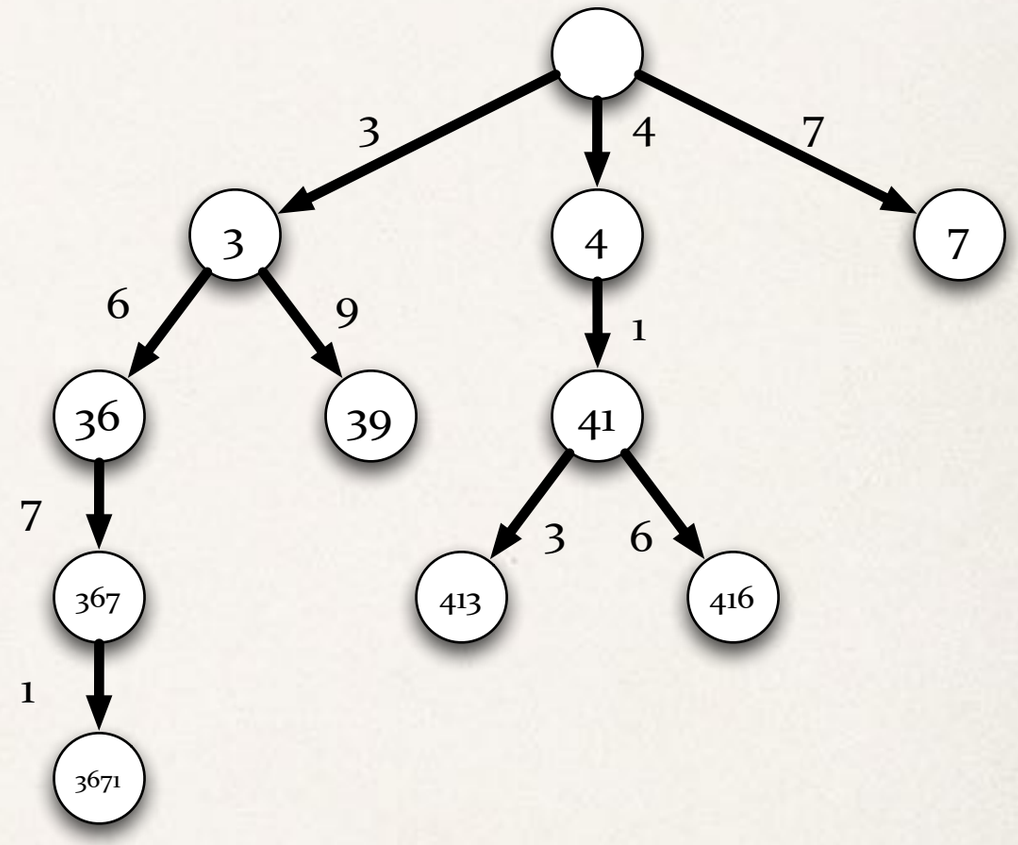
  ((3 . t_3) (4 . t_4) (7 . t_7))

- Its leftmost child:

  ((6 . t_6) (9 . t_9))

- A leaf:

  ()



Abstract data types

- Computer Scientists frequently organize their thinking about data types with the idea of an *Abstract Data Type*.

- An *abstract data type* is a high-level description of the functionality provided by a data type, without any reference to exactly how it is implemented.

- This "abstraction layer" hides implementation details: *any implementation of an ADT can be "plugged in" to a computing infrastructure that requires it.*

- You've seen an example of this in our discussion of the SET datatype.

The SET ADT

- Recall our discussion of the SET ADT.

- A SET datatype must offer two functions: **insert**(x, S) and **ismember**(x, S). (It also provides a distinguished object called **emptyset**).

  - To fully describe the ADT, we describe the functions it provides and, more importantly, the *semantics* that these functions offer.

  - For SET, this is easy: **ismember**(x,S) returns TRUE if **insert**(x, S) has ever been called, and FALSE otherwise.

- Note that we have defined the behavior of these operations without specifying the implementation.

Implementations of Set

- We've seen three implementations of Set:
  - Lists
  - Binary search trees (though this required that the set have a linear order)
  - Tries (though this required that the set be sets of words over a fixed alphabet)
- How do these implementations differ? In terms of the data structure they maintain and, additionally, their *computational requirements*.

A richer SET ADT

- We've been discussing a very basic notion of the SET ADT.
- You could ask for much more:
  - **Union**(S_1, S_2)
  - **Intersection**(S_1, S_2)
  - **Delete**(x, S_1)
- Producing an efficient implementation of such a rich set ADT is a deep mathematical issue: it was a longstanding research problem with fairly sweeping successes in the late 1990s.

The STACK ADT

- The STACK is an ADT that intuitively represents a "stack" of objects, from which you can *push* on new objects, and *pop* off recent objects.

- The STACK ADT requires the following family of operations:

  - **push**(x,S): "pushes" an element x onto the top of a stack S, returning the new stack

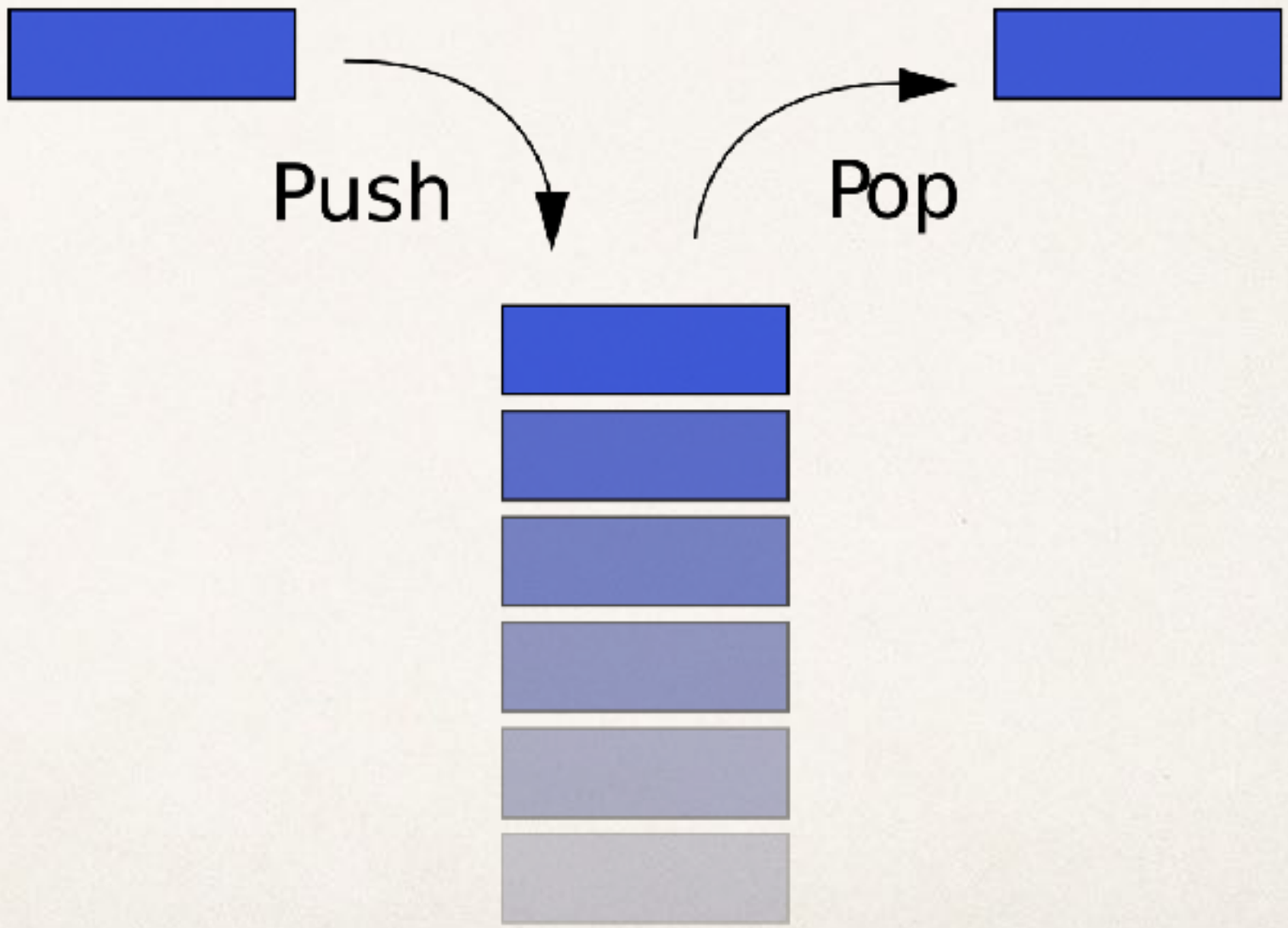
  - **top**(S): returns the top element of a stack.

  - **pop**(S): "pops" off the top element of a stack, returning the resulting stack.

  - **empty?**(S): returns TRUE if a stack is empty, FALSE elsewhere.

The STACK intuition

- These functions implement the natural behavior of a "stack" of objects.



Implementing a STACK in SCHEME

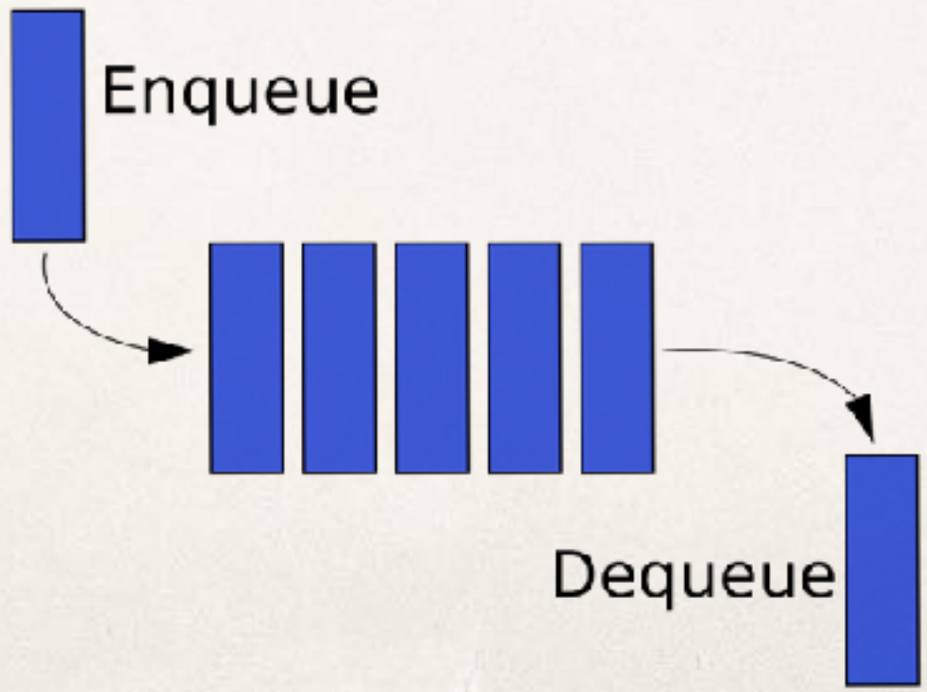
- Stacks can (of course) be naturally implemented in terms of lists:

```
(define (push x S) (cons x S))

(define (top S) (car S))

(define (pop S) (cdr S))

(define (empty? S) (if (null? S) #t #f))
```

This implementation seems hard to beat: any of these operations requires a *fixed number* of procedure calls.

The Queue ADT

- The Queue ADT captured the natural behavior of a queue:
  - Elements can be added to the end of the queue (**enqueue**).
  - Elements can be removed from the front of the queue (**dequeue**).



The QUEUE ADT functionality

- The QUEUE ADT must provide the following functionality:
  - **enqueue**(x, Q): place x at the end of the queue Q.
  - **dequeue**(Q): remove the front element from the queue Q.
  - **front**(Q): return the front element of the queue Q (without removing it).
  - **empty?**(Q): returns TRUE if the queue Q is empty, FALSE otherwise.

Implementing the Queue ADT in Scheme

- We can naturally implement the queue ADT as a list in scheme. There's a problem: placing an element at the back is expensive

- Can you think of a better implementation?

```
(define (enqueue x Q)
  (if (null? Q)
      (list x)
      (cons (car Q)
            (enqueue x (cdr Q)))))

(define (front Q) (car Q))

(define (empty? Q) (null? Q))

(define (dequeue Q) (cdr Q))
```

Efficiency of this implementation of the QUEUE ADT

- With this implementation: `(first second ... last)`

- Now **dequeue**(Q) is cheap, so is **front**(Q); each uses only a fixed number of calls.

- **enqueue** is terribly expensive: we need to step through the *entire list* to place the element at the end.

- Can you do better? Could both **enqueue** and **dequeue** be done with just a fixed number of calls, independent of the length of the queue?

The Priority Queue ADT

- A priority queue is an ADT that maintains a set of numbers, but only supports extraction of the minimum element as a means of deletion:

- **Empty?** Determines whether the priority queue is empty.

- **Insert(x)** Inserts the number x into the priority queue.

- **Extract_Min** Removes the smallest element from the priority queue and returns it.

Implementation of PRIORITY QUEUEs

- As an *unsorted list*.
  - Extract_min is slow! Have to traverse entire list. Insertion is fast: Constant time.
- As a *sorted list*.
  - Extract_min is fast! Constant time. Insertion is slow: at least N procedure calls, if there are N elements in the priority queue.
- As a *heap*.
  - Extract_min takes ~log(N) time to traverse the heap. Insertion takes ~log(N) time to traverse the heap. **Heaps are really good at this.**

Designing more efficient implementations of such ADTs...

- ...motivates our next topic:

Destructive assignment