*traversal:
  a basic operation behind binary tree (or tree) functions
  
  postorder on expression tree => evaluation
  preorder on two binary trees => equality testing
  inorder on \( \triangleleft \text{root} \triangleleft \) => ordered data
  
  level-order on a tree => closest leaf to root

* in-order revisited

\[
\text{inorder}(T) = \begin{cases} 
\text{output}(T \to \text{data}) & \text{if } \text{inorder}(T \to \text{left}) \\
\text{inorder}(T \to \text{right}) & \text{else}
\end{cases}
\]

while (...) {
  \( (T, \text{state}) \leftarrow \text{pop from stack} \)
  switch (state) {
    0: push(T, i); push(T \to \text{left}, 0); \text{break};
    1: output(T \to \text{data}); push(T \to \text{right}, 0); \text{break};
    2: push(T \to \text{right}, 0); \text{break};
  }
}

(2) can be mixed w/ 1 (push immediately popped)
  0: push(T, i); push(T \to \text{left}, 0); \text{break};
  1: output(T \to \text{data}); push(T \to \text{right}, 0); \text{break};

(b) \[ \square | \square | \square | \square \] can be replaced by a while
  0: while(T) { push(T, i); T = T \to \text{left}; } \text{break};

(c) \[ \square \] can be replaced by next while
  1: output(T \to \text{data}); T = T \to \text{right}; while(T) { ... } \text{break};

(d) only state 1 left, can be removed

while(T) { ... }
* why need stack in in-order traversal?

C needs to visit B next, so B on stack to wait
D needs to visit A next, so A on stack to wait

what if C "links" to B and B "links" to A?

start \( \rightarrow \) C \( \rightarrow \) B \( \rightarrow \) D \( \rightarrow \) A \( \rightarrow \) left-right F \( \rightarrow \) E

an implicit linked list of in-order traversal results.
no need for stack!

if next one is left \( \rightarrow \) lead \( \rightarrow \) nongrid \( \rightarrow \)
otherwise, next one is right \( \rightarrow \) right-left \( \rightarrow \) right

* how to store \( \rightarrow \) (successor of the node)

\[
\begin{array}{c|c|c}
\text{L} & \text{R} & \\
\hline
A & &
\end{array}
\]

but either \( R = \text{NULL} \) or \( \rightarrow = \text{NULL} \)

\[
\begin{array}{c|c|c}
\text{L} & \text{R} & \\
\hline
A & &
\end{array}
\]

shared, need \( \oplus \) to know which (one bit)

right-threaded binary tree

* what if we want inverse inorder traversal (right before left)

\( \leftarrow \) for predecessor

left-threaded binary tree

\[
\begin{array}{c|c|c}
\text{L} & \text{R} & \\
\hline
A & &
\end{array}
\]

* threaded binary tree: left- and right-threaded

all the NULL links in original binary tree replaced with \( \leftarrow \rightarrow \) and the purpose of NULL done by \( \oplus \)

"Subsec. 5.5.3: insert to threaded READING ASSIGNMENT"
* pre-order revisited

\[
\text{preorder}(T) = \begin{cases} 
\text{output}(T \to \text{data}) ; \\
\text{preorder}(T \to \text{left}) ; \\
\text{preorder}(T \to \text{right}) ; 
\end{cases}
\]

\[
\text{while}(\cdots) \{ 
\begin{array}{l}
(T, \text{state}) = \text{pop from stack} \\
0: \text{output}(T \to \text{data}) ; \text{push}(T, 1) ; \text{break} \\
1: \text{push}(T, 2) ; \text{push}(T \to \text{left}, 0) ; \text{break} \\
2: \text{push}(T \to \text{right}, 0) ; \text{break} \\
\end{array}
\}
\]

same as

\[
\text{while}(\cdots) \{ 
\begin{array}{l}
T = \text{pop from stack} \\
\text{output}(T \to \text{data}) ; \\
\text{push}(T \to \text{right}) ; \text{push}(T \to \text{left}) ; 
\end{array}
\}
\]

* if use queue instead of stack

\[
\begin{array}{c}
A \text{ in} \\
B \text{ in} \\
C \text{ in} \\
E \text{ in} \\
D \text{ in} \\
F \text{ in} \\
G \text{ in} \\
\end{array}
\]

level-order traversal

e.g. for finding the leaf closest to root

* recall: maze search

level-order: shortest path out
recursive (in/pre/post-order): left-most path out
* * A * so far

will deal with

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>r</td>
<td>l</td>
</tr>
</tbody>
</table>

goal: find the node with key property **??** efficiently within some (special) binary tree

* case 4: **??** = largest

e.g. key means priority
data is an entry to an item in your todo list

idea: put the node w/ largest key close to the root

( how about the root directly ?)

but after getting **14(...)**, hard to get next (second largest) node

* binary max-tree:

  1. root key larger than key of other node (or equal to)
  2. every sub-tree is a max-tree

Get-Largest(T) { return T; }

Remove-Largest(T) {
  check the larger key of T→left or T→right; call it node
  replace T→key, T→data w/ node→key, node→data;
  Remove-Largest(node);
}