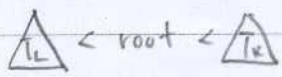
* traversal:

a basic operation behind binary tree (or tree) functions

postorder on expression tree ⇒ evaluation
pre order on two binary trees ⇒ equality testing
inorder on 

⇒ ordered data

level-order on a tree ⇒ closest leaf to root

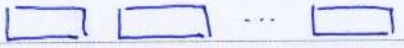
* inorder revisited

```
inorder(T)
= { inorder(T->left)
  { output(T->data)        =
  { inorder(T->right)
```

```
while(...) {
  (T, state) ← pop from stack
  switch(state){
  0: push(T,1); push(T->left, 0); break;
  1: output(T->data); push(T,2); break;
  2: push(T->right, 0); break;
  }
}
```

ⓐ 2 can be mixed w/ 1 (push immediately popped)

```
0: push(T,1); push(T->left,0); break;
1: output(T->data); push(T->right,0); break;
```

ⓑ  can be replaced by a while

```
0: while(T) { push(T,1); T = T->left;} break;
```

ⓒ  can be replaced by next while

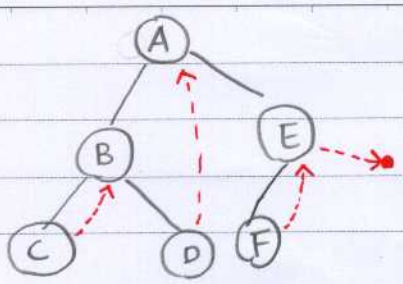
```
1: output(T->data); T = T->right; while(T){...} break;
```

ⓓ only state 1 left, can be removed

```
while(T) {...}
pop T from stack
output(T->data)
T = T->right
```

another while

* 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

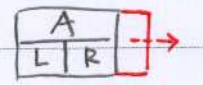
what if (C) "links" to (B) and (D) "links" to (A) ?

start —left→ left → (C) ⇢ (B) —right→ (D) ⇢ (A) —right→ left → (F) ⇢ (E)

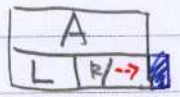
an implicit linked list of in-order traversal results,
no need for stack!

if ⇢ , next one is ⇢ (leaf ⇢ noright ⇢)
otherwise, next one is —right→ left → left → .. → left → (hasright —right→)

* how to store ⇢ (successor of the node)



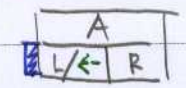
but either R = NULL
or ⇢ = NULL



shared,
need ▨ to know which (one bit)

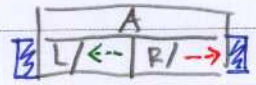
right-threaded binary tree

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



left-threaded binary tree

* threaded binary tree : left- and right-threaded



all the NULL links in original binary tree
replaced with ⇠ ⇢ and the purpose of NULL done by ▨

* Subsec. 5.5.3 : insert to threaded

READING ASSIGNMENT

* pre-order revisited

preorder(T)

```
= { output(T → data);
    preorder(T → left);
    preorder(T → right);
```

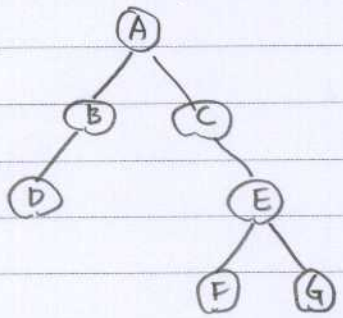
=

```
while(···) {
  (T, state) = pop from stack
  0: output(T→data); push(T, 1); break;
  1: push(T, 2); push(T→left, 0); break;
  2: push(T→right, 0); break;
}
```

same as

```
while(···) {
  T = pop from stack
  output(T → data);
  push(T → right); push(T → left);
}
```

* if use queue instead of stack

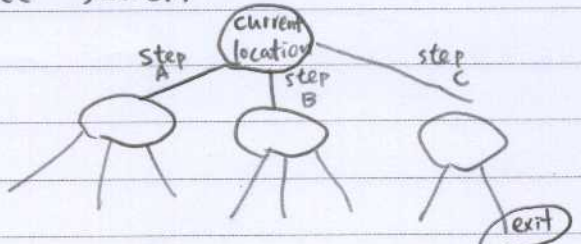


A	in			A push
A	out	C in	B in	A pop, C push, B push
C	out	E in		B pop, D push
B	out	D in		D pop
E	out			C pop, E push
D	out	G in	F in	E pop, G push, F push
G	out			F pop
F	out			G pop

level-order traversal

e.g. for finding the leaf closest to root

* recall: maze search



"search tree"
if same location not visited again

level-order: shortest path out

recursive (in/pre/post-order): left-most path out

* | A | | | so far
| l | r |

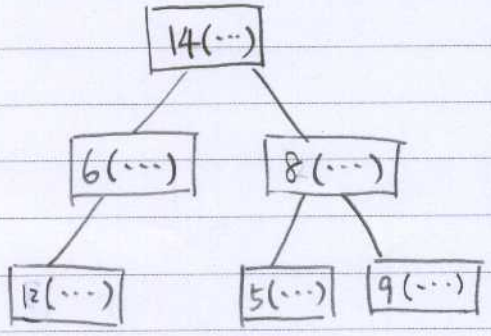
will deal with | key | data | | next
| l | r |

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

* case 1 : [??] = 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 bin. 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);
}
```