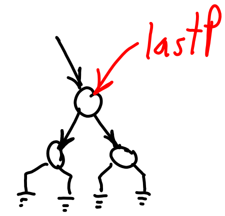
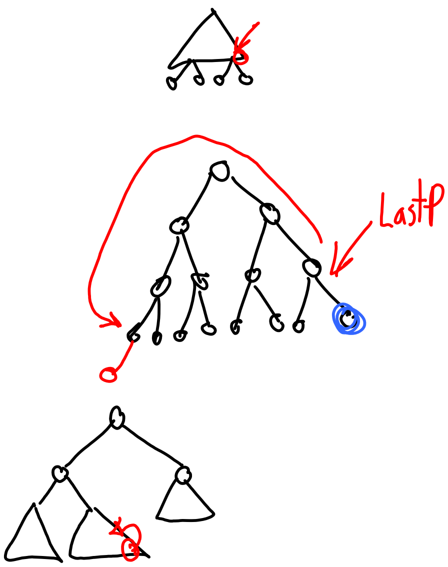
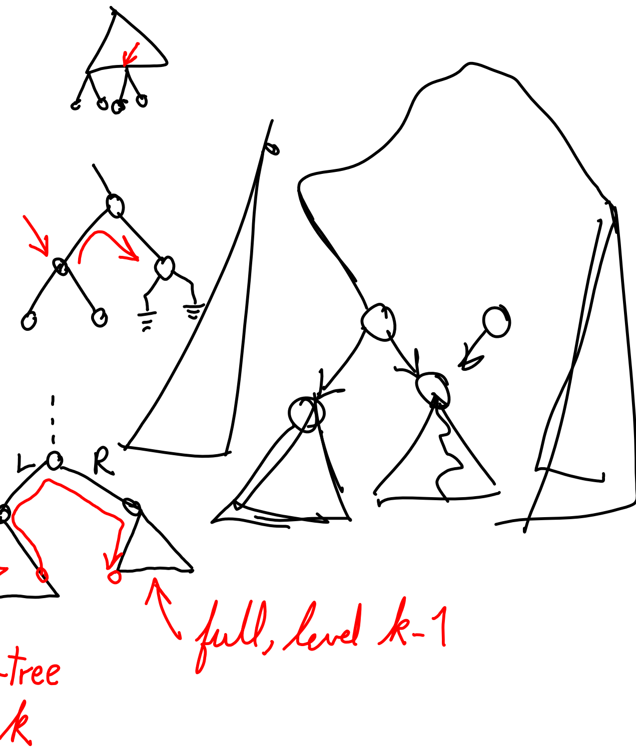
private + friend

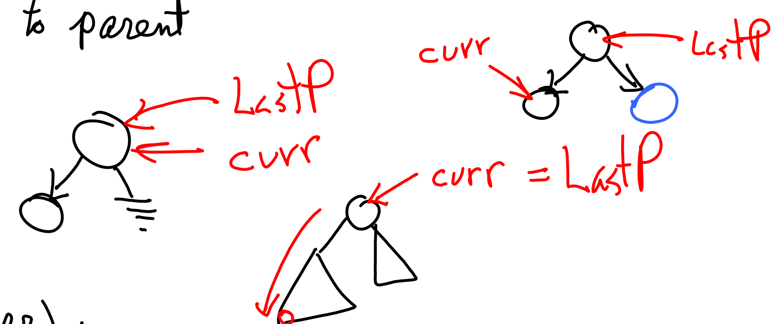
find next Parent



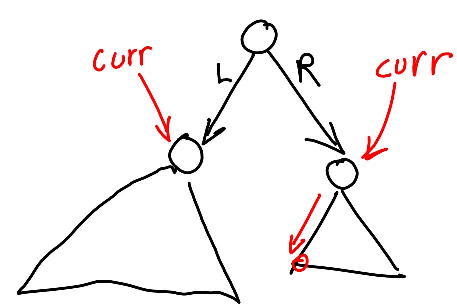


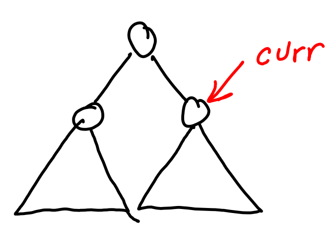


⇒ either, we are at root, or there is a Left branch above, or we move up to parent




```
getNextParent(Node *curr)

  if (curr == root){
      curr = goLeftmostLeaf(curr);
  }else
  if (curr->parent->left == curr){

      curr = curr->parent->right
      curr = goLeftMostLeaf(curr)
  }else {
      curr = curr->parent
      curr = getNextParent(curr)
  }
  return (curr)
```





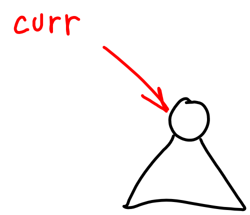
Exercise Prove correct

Fact: we start from rightmost leaf (ok, parent of) of complete/full subtree of height h, or we start at root.

ReHeapUp (curr)

either at root, or parent needs fixed, or we are done.

if (curr == root) return;

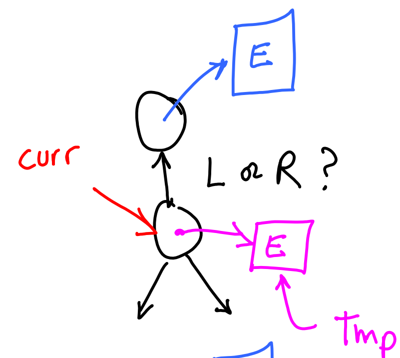


if (curr->event.time < curr->parent->event.time)

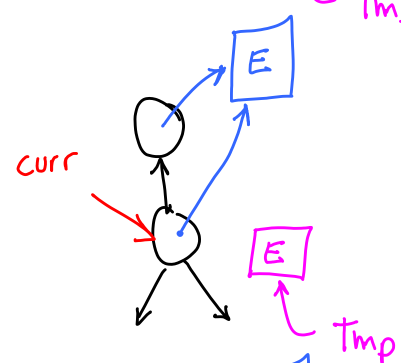
min heap

swapWithParent (curr)

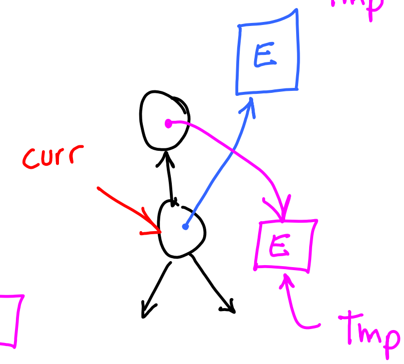
tmp = curr->event



curr->event = curr->parent->event



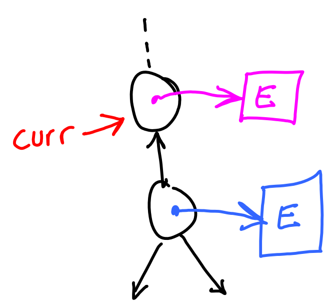
curr->parent->event = tmp



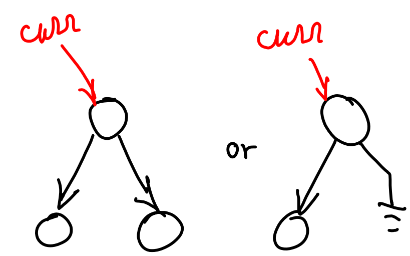
curr ← curr->parent

reHeapUp (curr)

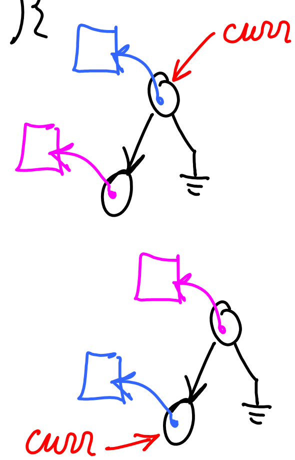
return



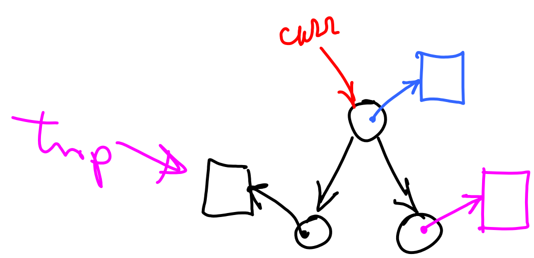
```
reHeapDown ( curr )

    if ( isLeaf ( curr ) ) return;
```




```
    if ( curr->right == NULL ) {

        if ( curr->left->event.time < curr->time ) {

            swapWithParent ( curr->left )
            curr = curr->left
            reHeapDown ( curr )
        }
        return;
```

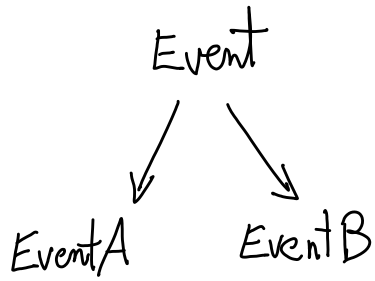


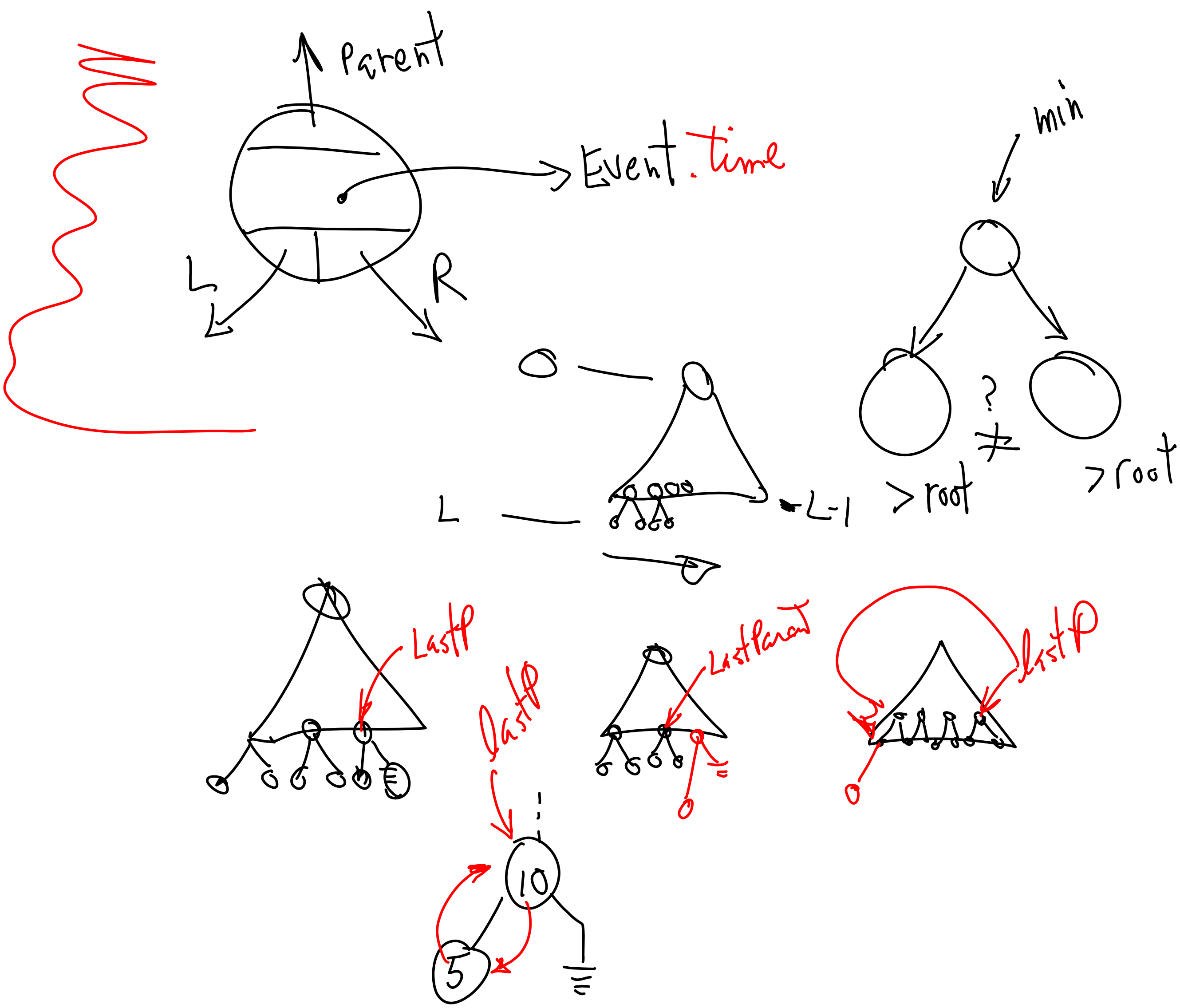

```
    if ( curr->right->event.time < curr->left->event.time )
        tmp = curr->right
    else
        tmp = curr->left
```

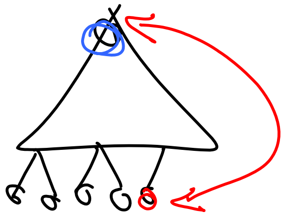


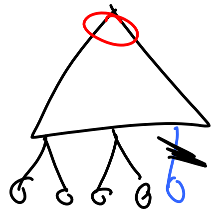

```
    if ( tmp->event.time < curr->event.time ) {
        swapWithParent ( tmp )
        curr = tmp
        reHeapDown ( curr )
    }
    return;
```

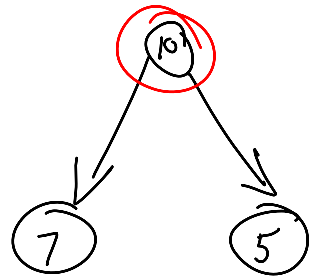
Polymorphism

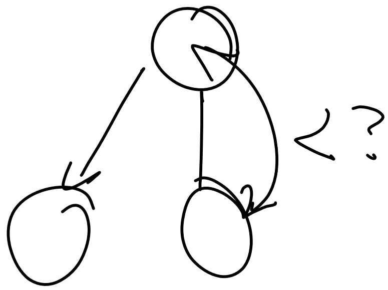


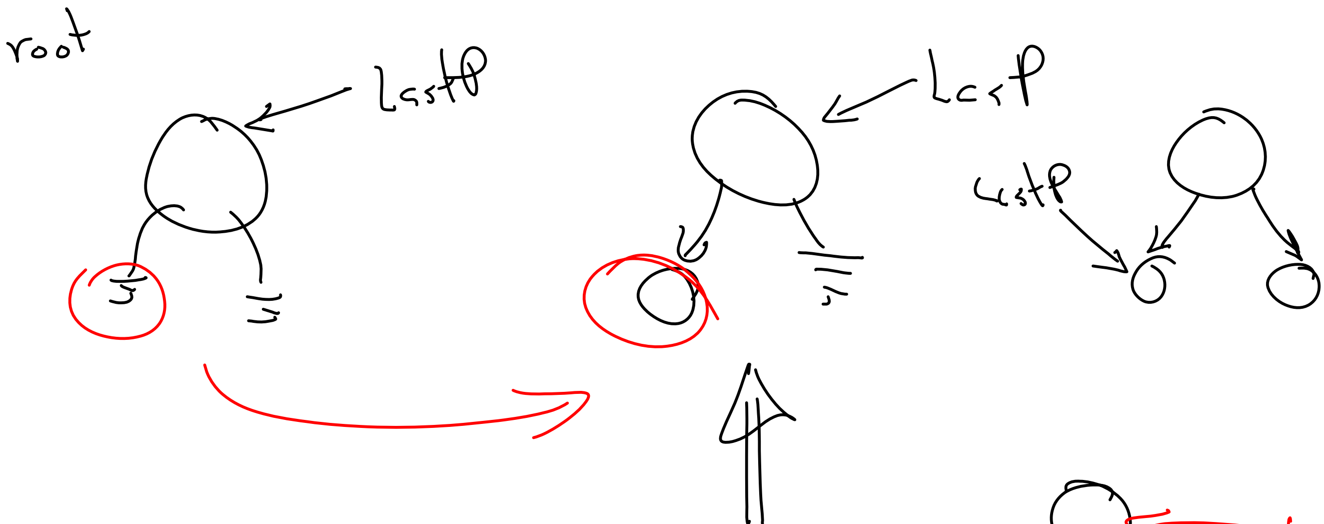


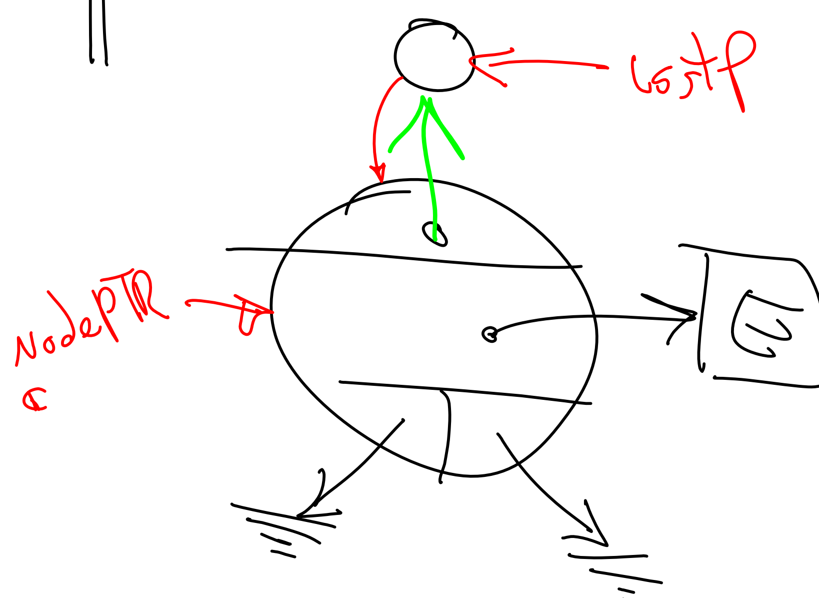












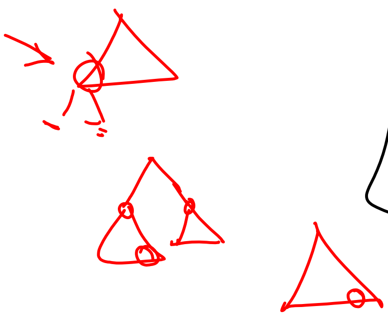

```
if ( lastParent->left == NULL)
   lastParent->left = nodePtr;
   nodePtr->parent = lastParent();
   return;
else
   lastParent->right = nodePtr;
   nodePtr->parent = lastParent();
   updateLastParent();
endif
```

base case:

works if the tree has height 0.

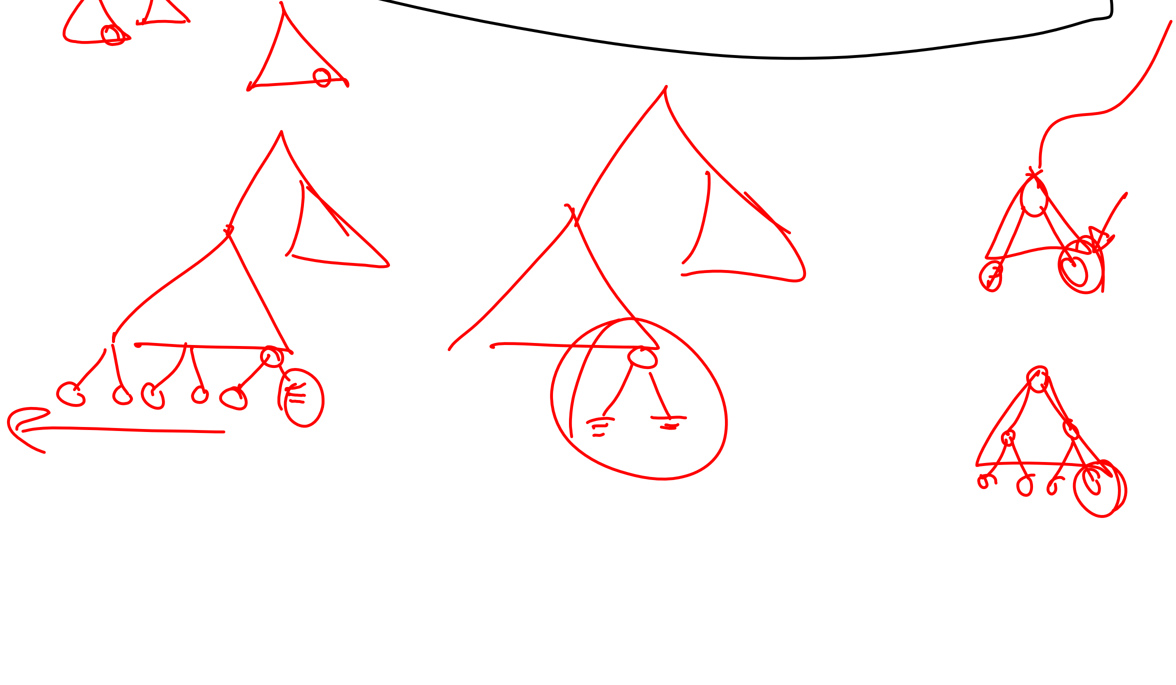
null, or 1 node

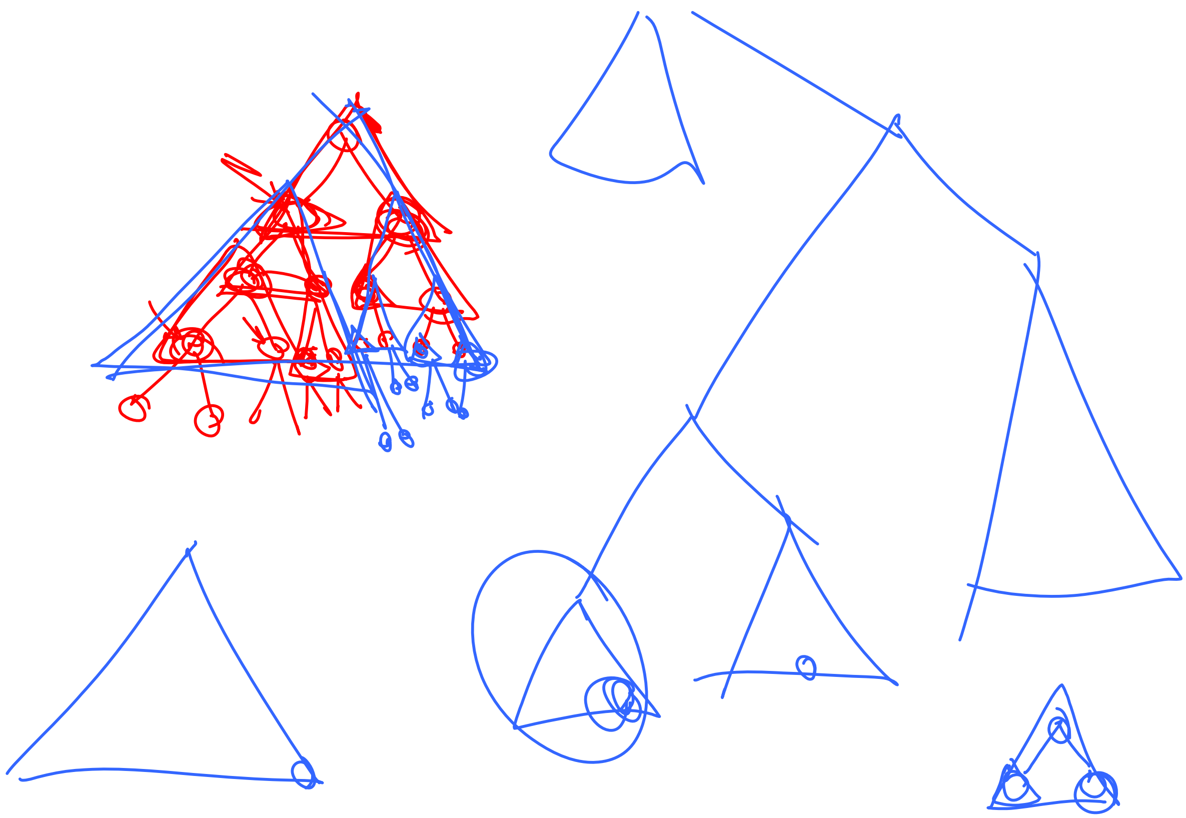
inductive case:



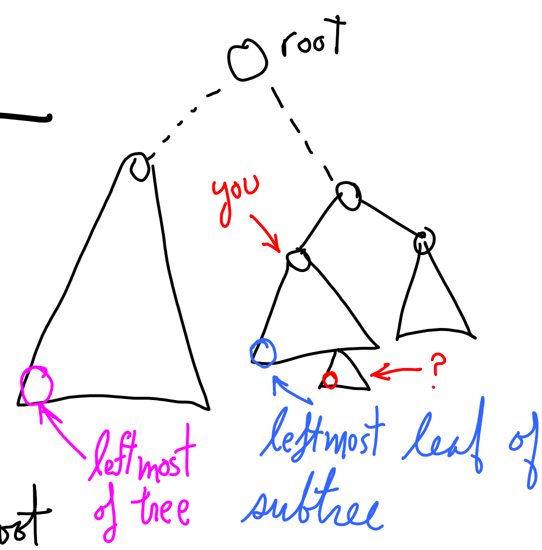
works for height h, therefore works for h+1

$\boxed{\text{Prove}}$





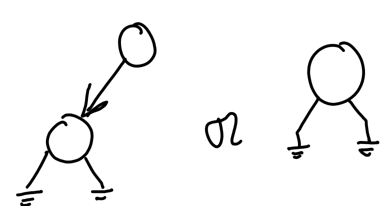
recursion, inductive proof



gotoLeftmostLeaf(curr)

defn A leftmost leaf of tree T is a node,

1. a ~~leaf~~ (has no L child)
2. is a left child, or is the root of the tree.
3. there is a path to the node, starting at the root, that uses only left links.



goL(curr)

if(isEmpty()) return(NULL); //-- doesn't exist

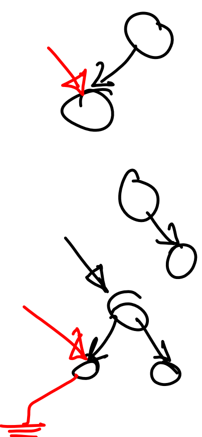
//-- basis case.

$n = 0$ ~~if (curr = root)~~

~~if (curr → left == NULL) && (curr → right == NULL)~~

$n = 1$ ~~return(curr);~~ //--- ~~satisfies~~ 1. & 2. & 3.

//--- basis case.



if(curr → left == NULL) return(curr);

curr = goL(curr → left); //-- Recursive call to child.

//-- inductive step.

return(curr);

Correctness Proof

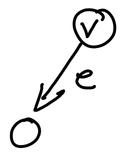
case 1. If the tree is empty, goL() returns NULL, which is correct.

case 2. If goL() is called initially w/ curr at the root, and the root is the only node in the entire tree, then it returns a pointer to the root, which is correct.

Given 1. and 2., we need only consider the case where goL() is called initially at the root of a sub-tree (the entire tree is also a sub-tree). It terminates correctly if it returns the leftmost leaf of this subtree.

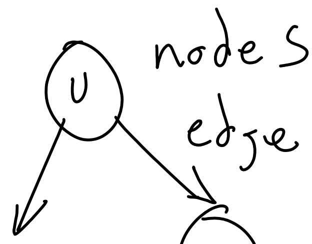
<u>Lemma</u> Every node on the path from the root to its leftmost leaf, x, is the root of a sub-tree which has the same leftmost leaf.

Proof: Suppose node n has some other left leaf, y. There is a path from n to y using only left links. Since n is on the path from the root to x, and this path uses only left links, there is a path of left links from n to x. Nodes x and y cannot be internal nodes on these paths, since they are leaves. If x and y are not identical, there must be some edge in n-y that is not in root–n–x. Consider the first such edge closest to root. It is a left edge



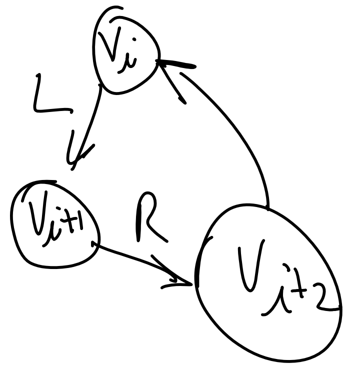
from some node v. Since it is the first such edge, either v is the root, or v has an incoming edge common to root-n-x and n-y. Thus we have root–n–v–x and n-v-y. Since these paths do not share edge e, there must be an edge e′ from v on the path v–x. Since e′ must be a left edge, v has two left edges. $\times\!\!\!\times$

Suppose the length of the path root – x is n. Suppose goL( ) works correctly for all trees where length of root-x is (n-1).



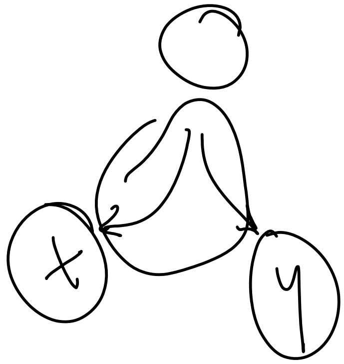
$E = \{ (u,v) \ldots \}$

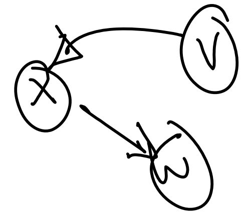
$V = \{ u, v \ldots \}$



$(V_i, \{V_i, V_{i+1}\}, V_{i+1}, (V_{i+1}, V_{i+2}), \ldots)$

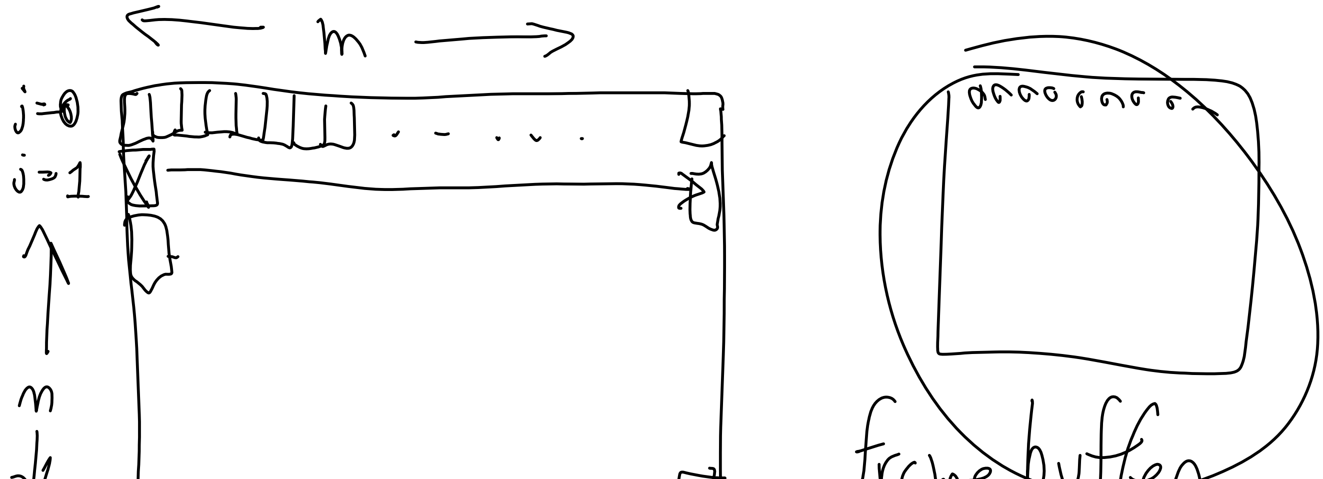
$(v, x), (x, w) \qquad (x, w)(v, x)$





$n=0$

if goL() works for $T_n \rightarrow$ works T_{n+1}



frame buffer

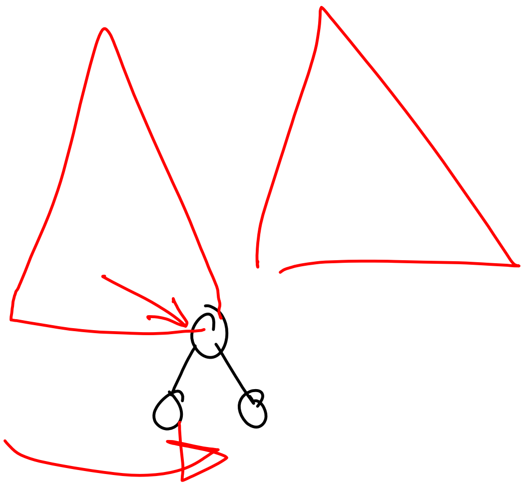
display

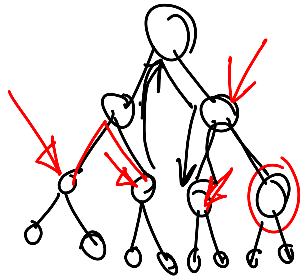
$j <= (n-1)$

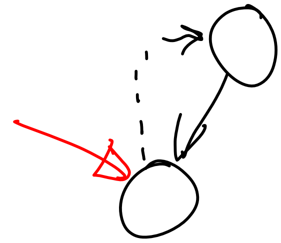
A[n][m]

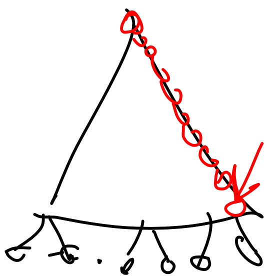
cout << A[0][i] $i <= (m-1)$

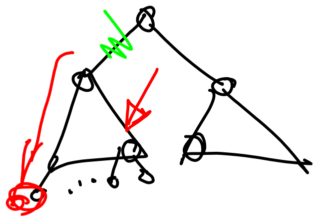
cout << "\n"

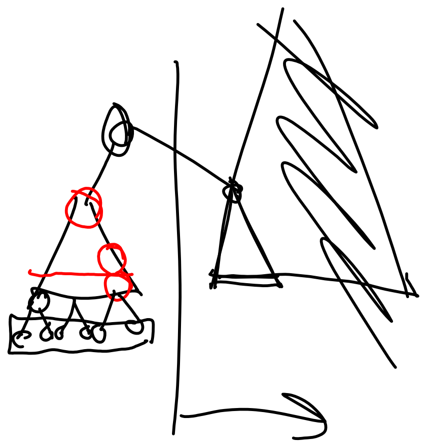


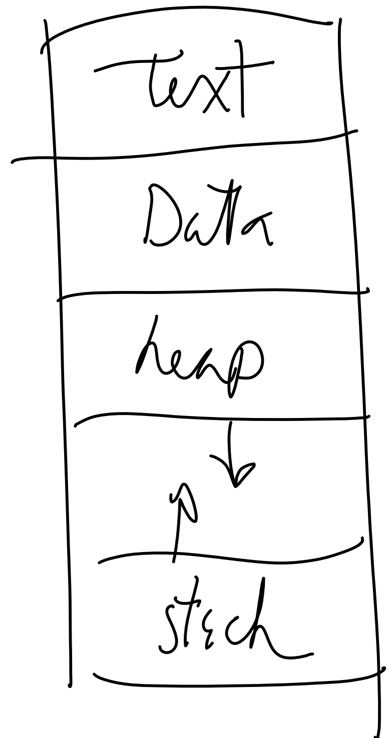











Text
Data
heap
stack