Heap / Binary Search Tree

Hsuan-Tien Lin

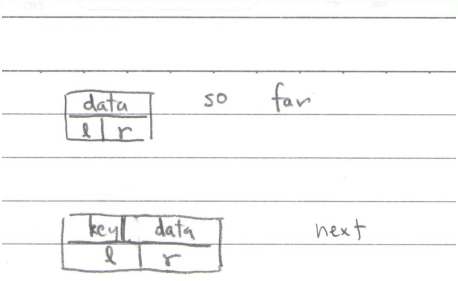
Dept. of CSIE, NTU

April 13, 2021

What We Have Done

- tree: hierarchical access
- binary tree: "simplest" tree, where (complete binary tree $\sim$ array)
- binary max-tree: binary tree + max-at-root, for priority queue

Need: Priority Queue (with Binary Tree)



- key: priority
- data: item in todo list

goal: get the node with largest key

Design Thoughts of Priority Queue with Tree (2/4)

`removeLargest` needs 2nd largest to replace

- to allow fastest removal, put 2nd largest close to next entry points
- next entry points of tree?

max tree:

- root key $\geq$ other node's key
- every sub-tree also max tree

`removeLargest` (version 0): recursive duel of sub-tree roots

Design Thoughts of Priority Queue with Tree (3/4)

- time complexity of `removeLargest` w.r.t. *h*?
- to make *h* small w.r.t. *n*, need short trees
  —what is shortest binary tree with *n* nodes?
- (binary) max-heap: max-tree + complete binary tree

does `removeLargest` (version 0) work with binary heap?

Design Thoughts of Priority Queue with Tree (4/4)

`removeLargest` (version 0) NOT work with binary heap
—hard to preseve complete binary tree

- preserve complete binary tree first? how?
- `removeLargest` (version 1)
  - move last node to root
  - three-way-duel (root & two sub-roots) in each sub-tree recursively: $O(h)$

lesson learned: preserve harder constraints first

Priority Queue Insertion

binary max-heap: max-tree + complete binary tree

- insertion: preserve which first?
- how? insert where first?
- how to preserve other property?

`insert`

- insert at last position
- duel with parents until satisfied: $O(h)$

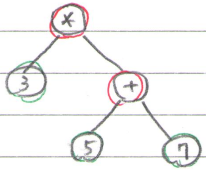
Binary Max-Heap in Array

complete binary tree	consecutive array
max-heap	partially-ordered consecutive array

unordered array	partially-ordered array (max-heap)
selection sort $O(n) \cdot O(n)$	heap sort $O(n) \cdot O(\log n)$

$O(n \log n)$ sorting with only the original array:
unordered $\rightarrow$ heap $\rightarrow$ heap sort

More on Binary Trees: (Binary) Expression Tree



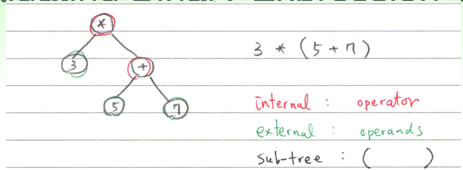
3 * (5 + 7)

internal : operator

external : operands

sub-tree : (   )

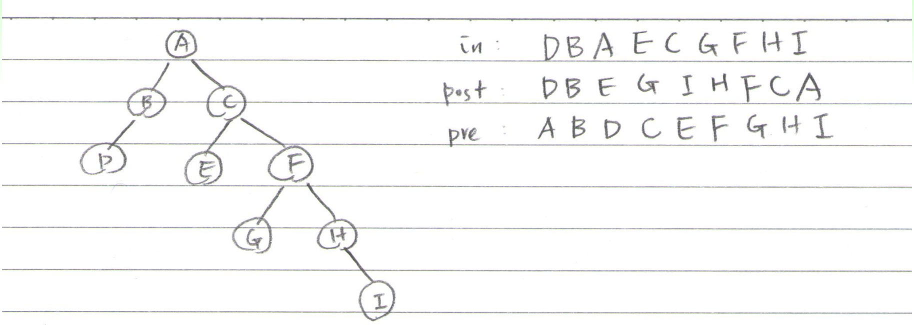
Evaluating Binary Expression Tree




```
EVALUATE(T)
if T.data is a number
    return T.data
else
    a = EVALUATE(T.left) // recursion on left
    b = EVALUATE(T.right) // recursion on right
    return compute(a, T.data, b) // action on data
```

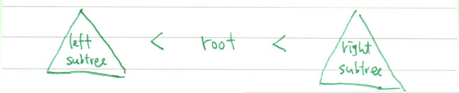
traversal: how *T.data* is visited
post-order traversal: *T.data* visited after sub-tree recursions

Other Traversals



- post-order: `Post(T.left); Post(T.right); Visit(T.data);`
  —expression tree evaluation: Visit by result computation
- pre-order: `Visit(T.data); Pre(T.left); Pre(T.right);`
  e.g. comparing if two trees are equal using equality as Visit
- in-order: `In(T.left); Visit(T.data); In(T.right);`
  e.g. output ordered data from tree if $T_L < root < T_R$ (see next)

Binary Search Tree



BST-SEARCH(T, *key*)

while $T \neq$ NIL
    mid $= T$ // root
    if *mid.key* $=$ *key*
        return *mid*
    elseif *mid.key* $<$ *key*
        return ??
    elseif *mid.key* $>$ *key*
        return ??
return FAIL

complexity??

BINARY-SEARCH(A, *key*)

left $= 1$, *right* $= A.length$
while *left* $\leq$ *right*
    mid $= \text{floor}(\frac{left+right}{2})$
    if $A[mid] = key$
        return *mid*
    elseif $A[mid] < key$
        $left = mid + 1$
    elseif $A[mid] > key$
        $right = mid - 1$
return FAIL

complexity: $O(\log n)$