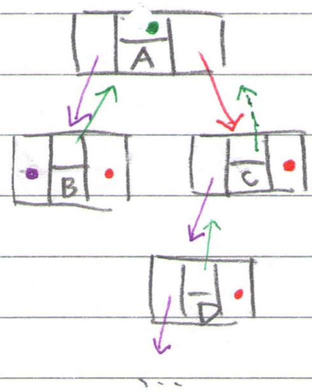
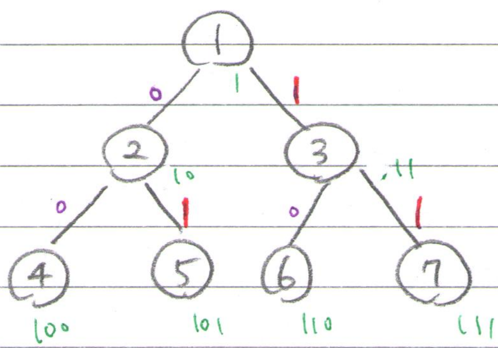
* linked representation of binary tree

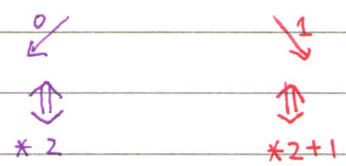


left
right
parent (optional)

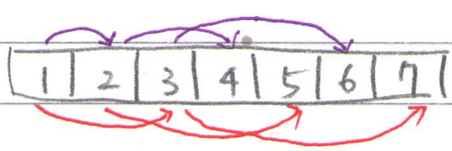
* full binary tree



node # = $(1 \cdot \text{path code})_2$



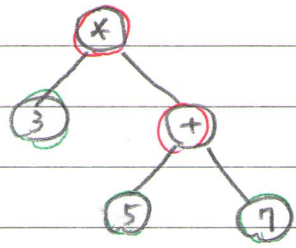
can "pack" the tree in a vector



- links "implicit" (no need to store)
- waste space if not full binary tree
  (useful if nearly full)

complete binary tree : nearly full (w/ nodes $1 \sim n$ exactly)

* expression tree revisited



3 * (5 + 7)

internal : operator

external : operands

sub-tree : (   )

print out infix notation

```
InfixPrint ( p ) {
  • if (isLeaf(p))  print p -> data;  // operand
    else {
        print "(";
  •     InfixPrint (p -> left);
  •     print p -> data;    // operator
  •     InfixPrint (p -> right);
        print ")";
    }
}
```

• : inorder traversal of the tree (print ⇒ visit)

visit sequence : 3, *, 5, +, 7

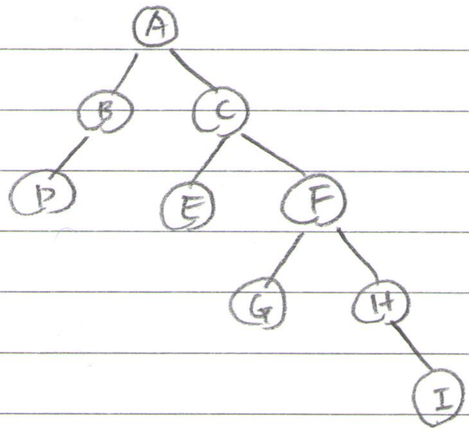
* postfix notation ⇒ postorder traversal

Postfix (p -> left); Postfix (p -> right); visit p -> data;

prefix notation ⇒ prefix traversal

visit p -> data; Prefix (p -> left); Prefix (p -> right);

*



in : D B A E C G F H I

post : D B E G I H F C A

pre : A B D C E F G H I

* why traversal : many bin. tree operations are similar to one of the traversal

postorder on exp. tree ⇒ evaluation

preorder on two bin. trees ⇒ equality test

inorder on T_L < root < T_R ⇒ ordered output

data	
l	r

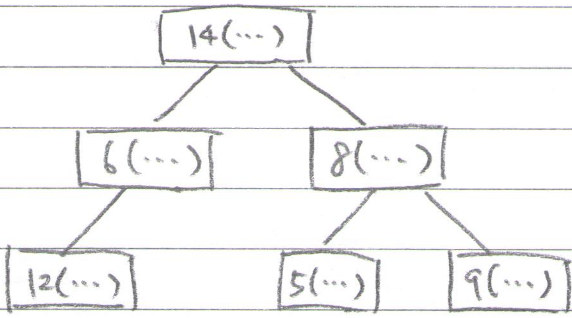
so far

key	data
l	r

next

* task: find the node w/ largest
  e.g. { key means priority
  data is an entry to an item in your todo list

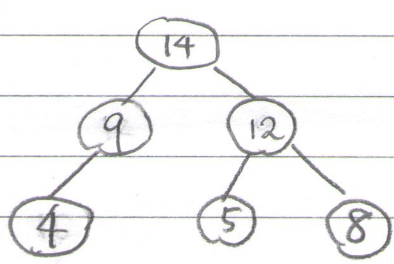
idea: put the node w/ largest key close to the root
— how about the root itself?



but after getting $\boxed{14(\cdots)}$, hard to get next (2nd-largest) node

* binary max-tree
  ① root key larger than (or =) key of other node
  ② every sub-tree is a bin. max-tree

for each v, parent(v) → key ≥ v → key




```
GetLargest(T) { return T.root → data; }
RemoveLargest(T) {  node ⇐ larger of two children of T.root;
                    replace T.root w/ node [in terms of content];
                    RemoveLargest(subtree at node);
}
```

* worst-case time of Remove Largest ?

$O(h)$ and hence possibly $O(n)$

③ how about requiring a complete binary tree ?

$O(h) = O(\log n)$

called max-heap

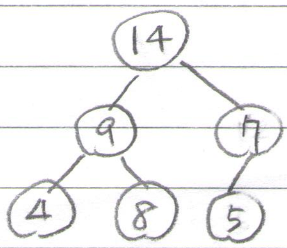
" but can we maintain it efficiently " ?

* remove Max

swap "last" to the "root", and roll down

* insert

put to "last", and roll up



remove Max ?

insert (10)

* complete binary tree → array (special)

max-heap → partially ordered array

$\boxed{\text{if there is a max-heap}}$ on an array

usual sel. sort

$O(n) \cdot \underbrace{O(n)}_{\text{selection}}$

heap sort

$O(n) \cdot O(\log n) = O(n \log n)$

* $\boxed{}$ from unsorted : $O(n \log n)$ by calling n insertion

or faster ! $O(n)$

reading assignment

* min-heap instead of max-heap in text book
  key can be anything that is Comparable
  ADT w/ insert & removeMax called priority-queue

  - PQ w/ heap : $O(\log n)$ insertion $O(\log n)$ removal
  - PQ w/ max-tree : $O(h)$ insertion $O(h)$ removal
  - PQ w/ ordered linked list : $O(n)$ insertion $O(1)$ removal
  - PQ w/ unordered linked list : $O(1)$ insertion $O(n)$ removal

  STL : PQ w/ heap (on vector)

* heap sort

  selection sort + max-heap
  n iter $O(\log n)$ $\Rightarrow$ $O(n \log n)$ w/ only original array