12/03/2014: (i) Write pseudocode to generate a binary tree given the each of the following: (a) node order printed using both in-order traversal and pre-order traversal and (b) node order printed using both in-order traversal and post-order traversal (b) show that given only the node order (of length n) printed using pre-orde traversal, there are atleast 2^{n-1} possible trees (ii) We saw a pseudocode for post-order traversal using recursio Write iterative pseudocode for the same, without invoking recursion. Does it help the iterative pseudocode if every child had pointer to its immediate parent? (iii) Carrying forward from the last class: Assuming that the hash values are like random numbers, derive an expression for the expected number of probes for an insertio with open addressing. **Deadline:12/03/2014**

Theorem.

Given an open address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$, assuming simple uniform hashing.

Proof. Define p_i = *Pr*(exactly i probes access occupied slots) for $i = 0, 1, 2, \ldots$ (Note that for $i > n$, $p_i = 0$). The expected number of probes is then $1 + \sum_{i=0}^{\infty} i \cdot p_i$.

Now define q_i = *Pr*(at least i probes access occupied slots), then $\sum_{i=0}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} q_i$

(why? (exercise)).

$q_i = \sum_{j \geq i} p_i$

The probability that the first probes accesses an occupied slot is $\frac{n}{m}$, so $q_1 = \frac{n}{m}$. A second probe, if needed, will access one of the remaining $m - 1$ locations which contain $n - 1$ possible keys, so $q_2 = \frac{n}{m} \cdot \frac{n-1}{m-1}$. Hence for $i = 1, 2, \ldots, n$

$$q_i = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \left(\frac{n}{m}\right)^i = \alpha^i.$$

Hence the following holds:

$$1 + \sum_{i=0}^{\infty} i \cdot p_i = 1 + \sum_{i=1}^{\infty} q_i \leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots = \frac{1}{1-\alpha}. \quad \blacksquare$$

n = # element

m = # slots

$\alpha = \frac{n}{m}$ = # of elements in a slot in uniform hashing

Example:

Evaluate Arithmetic Expressions

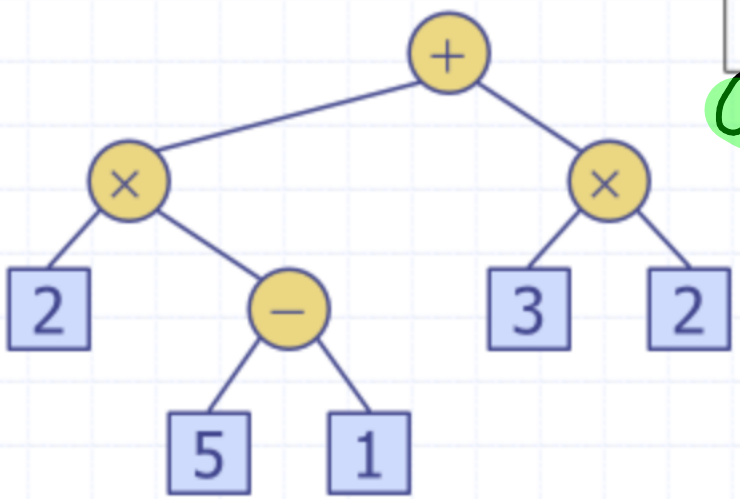
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

```
Algorithm evalExpr(v)
    if isExternal (v)
        return v.element ()
    else
        x ← evalExpr(leftChild (v))
        y ← evalExpr(rightChild (v))
        ◊ ← operator stored at v
        return x ◊ y
```

using child pointer

Postorder

(Does not need parent pointers)



Note: Lot of recursive calls here. A stack is used to implement recursion.

H/w: Replace recursion with iterative procedure using stack. Will parent pointers help??

Perhaps what should work is two stacks. As you pop a node off the first stack, you push that node on the second stack, and push the children back onto the first stack.

Another option is to mark each node as you start exploring the subtree rooted at that node.

```
/*
   Assuming you have a stack setup with push() and pop() operations.
   Also assuming that all nodes are initially marked to 0.
   (This function will reset them back to zero when finished)
*/
void postorder(Node *n) {
  push(n);

  while (stack.size > 0) {
    n = (Node*)pop();

    if (n->marked || (n->left == NULL && n->right == NULL)) {
      n->marked = 0;
      printf("%d\n", n->value);
    }
    else {
      n->marked = 1;
      push(n);

      if (n->right) push(n->right);
      if (n->left) push(n->left);
    }
  }
}
```

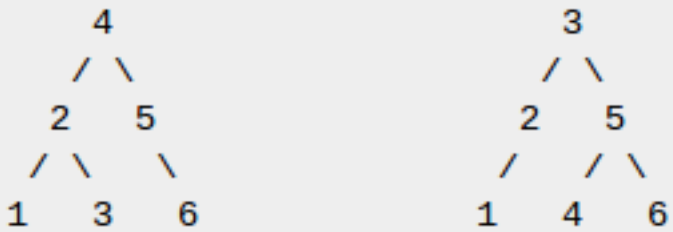
Iterative algorithm for Post order traversal using Pointer to parent node (without using stacks)

define 2 flags :
curr - current node
prev - previous node visited

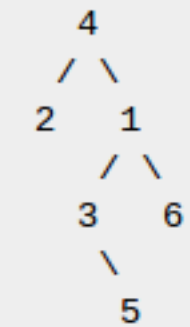
```
==========================================//
curr = root
prev = root
while ( curr has a child)
{
     if( left child is present)
       {
          curr = left child
          prev = *(pointer to parent of curr)
          if ( curr has no child)
             {
                print curr
                curr = prev
              }
        }
      if( right child is present)
        {
          curr = right child
          prev = *(pointer to parent of curr)
          if ( curr has no child)
             {
                print curr
                curr = prev
              }
        }
}
print curr
==========================================//
```

http://stackoverflow.com/questions/1136999/reconstructing-a-tree-from-its-preorder-and-postorder-lists

In general, a single tree traversal does not uniquely define the structure of the tree. For example, as we have seen, for both the following trees, an inorder traversal yields [1,2,3,4,5,6].

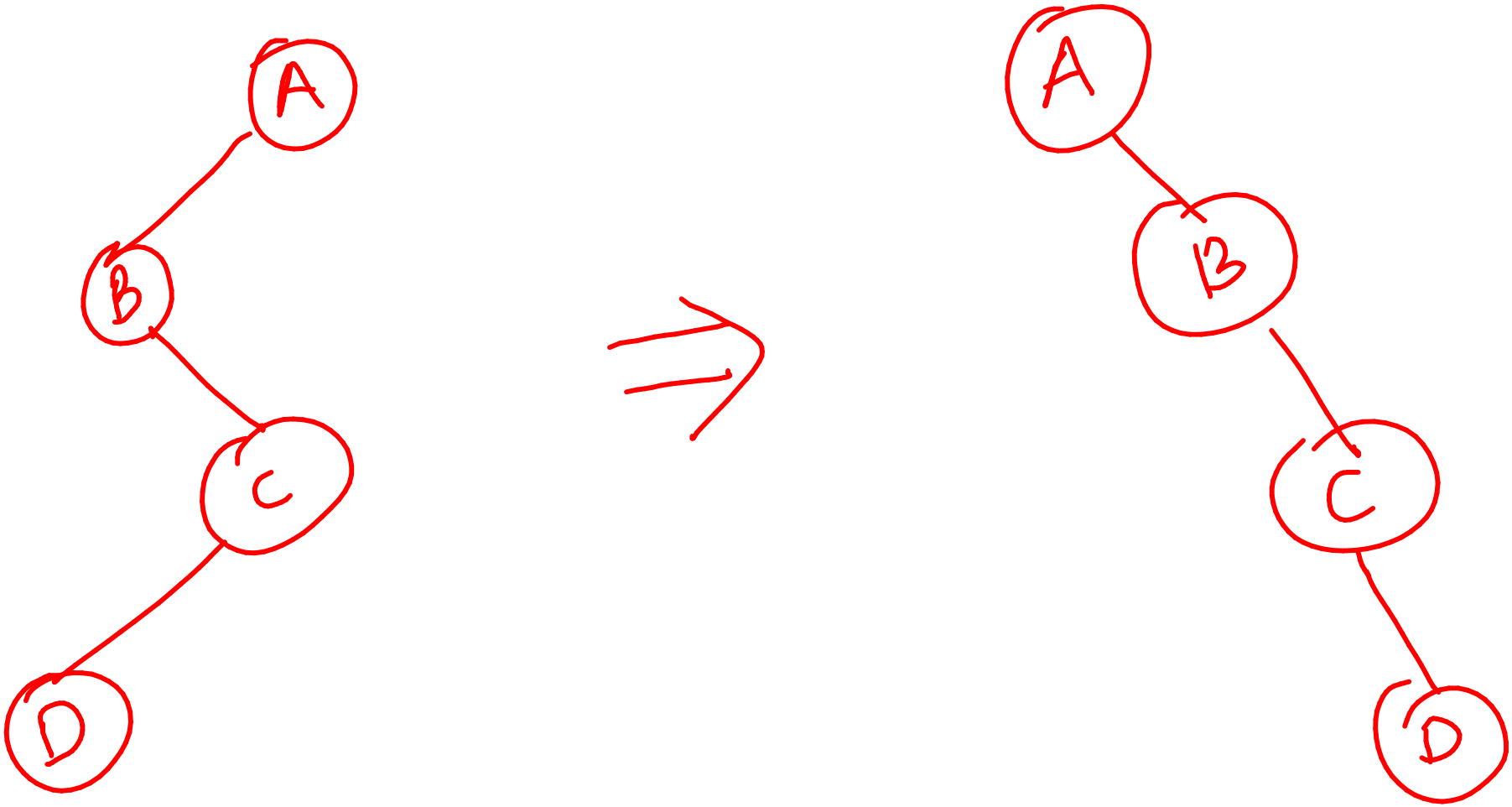


The same ambiguity is present for preorder and postorder traversals. The preorder traversal for the first tree above is [4,2,1,3,5,6]. Here is a different tree with the same preorder traversal.



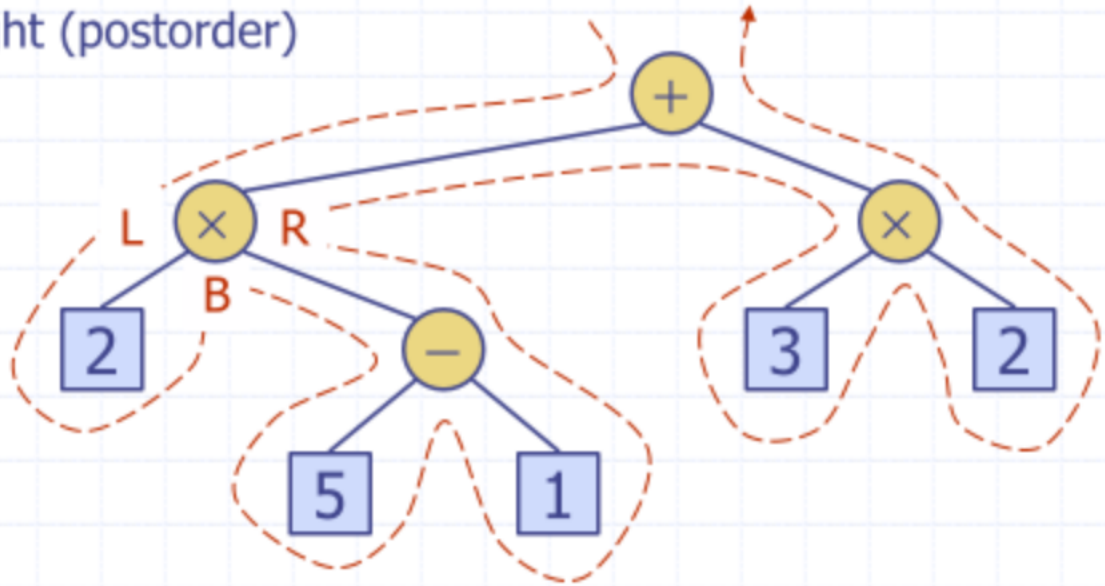
Similarly, we can easily construct another tree whose postorder traversal [1,3,2,6,5,4] matches that of the first tree above.

Pre & post do not uniquely determine Tree:



Euler Tour Traversal

- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



Priority Queues (PQ)

FIFO: Priority was "order" of insertion

PQ is generalisation of queue

Priority Queue ADT (§ 7.1.3)

- A priority queue stores a collection of entries
- Each **entry** is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, x)
    inserts an entry with key k and value x
  - removeMin()
    removes and returns the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

flight id

Eg: timestamp diff from current time

[Recall Extra Quiz problem 1]

Total Order Relations (§ 7.1.1)

Order on objects that are trees ⓐ Height ⓑ subset

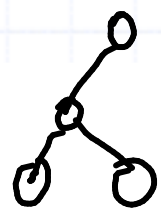
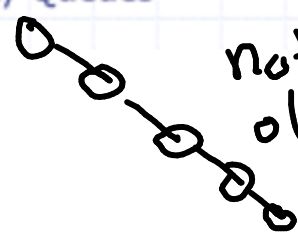
- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key

Partial order

- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Additional requirement for total order

- 2 elements must be comparable

Eg of partial order  &  not comparable objects w.r.t ordering "tree" "subset"

For "min" in Priority Queue (PQ) we are only interested in total order, since for non-comparable elements, min is not defined

Entry ADT (§ 7.1.2)

- An **entry** in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - key(): returns the key for this entry
  - value(): returns the value associated with this entry

- As a Java interface:

```
/**
 * Interface for a key-value
 * pair entry
**/
public interface Entry {
   public Object key();
   public Object value();
}
```

} Interested in min of keys

{Gives access to the total order}

Comparator ADT (§ 7.1.2)

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- The primary method of the Comparator ADT:
  - compare(a, b): Returns an integer i such that $i < 0$ if $a < b$, $i = 0$ if $a = b$, and $i > 0$ if $a > b$; an error occurs if a and b cannot be compared.

Eg keys can be trees & you can define/redefine compare (tree1, tree2)

b
a
Partialorder if each coordinate equally important

Example Comparator

- Lexicographic comparison of 2-D points:

```
/** Comparator for 2D points under the
    standard lexicographic order. */
public class Lexicographic implements
    Comparator {
  int xa, ya, xb, yb;
  public int compare(Object a, Object b)
   throws ClassCastException {
    xa = ((Point2D) a).getX();
    ya = ((Point2D) a).getY();
    xb = ((Point2D) b).getX();
    yb = ((Point2D) b).getY();
    if (xa != xb)
        return (xb - xa);
    else
        return (yb - ya);
  }
}
```

x coordinate more important than y coordinate

- Point objects:

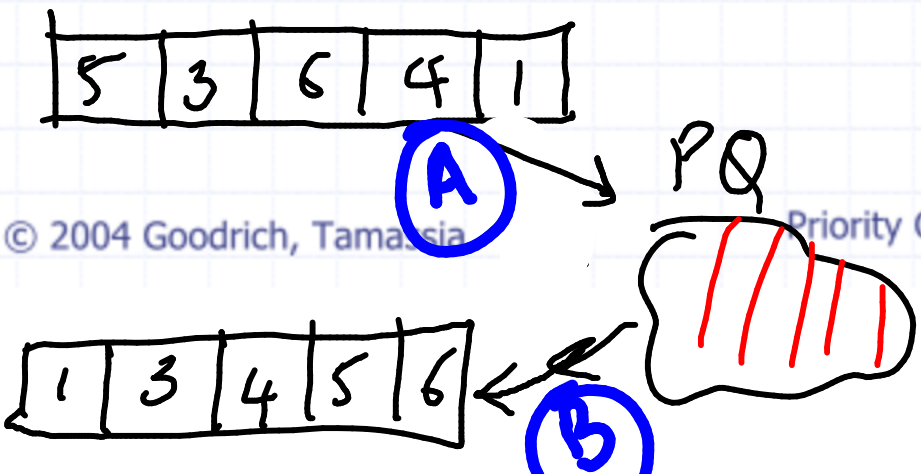
```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D        {
  protected int xc, yc; // coordinates
  public Point2D(int x, int y) {
    xc = x;
    yc = y;
  }
  public int getX() {
        return xc;
  }
  public int getY() {
        return yc;
  }
}
```


a: (x,y) b: (x,y)

Priority Queue Sorting (§ 7.1.4)

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
Input sequence *S*, comparator *C* for the elements of *S*
Output sequence *S* sorted in increasing order according to *C*
P ← priority queue with comparator *C*
while ¬*S.isEmpty* ()
e ← *S.removeFirst* ()
P.insert (*e*, *0*)
while ¬*P.isEmpty*()
e ← *P.removeMin*().*key*()
S.insertLast(*e*)



Sequence-based Priority Queue

- Implementation with an unsorted list (linked list)

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence ✓
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key ✗

- Implementation with a sorted list (linked list)

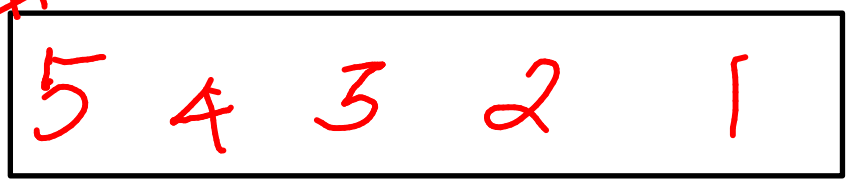
- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item to maintain sorted order ✗
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning ✓

This is the Insertion Sort Algorithm

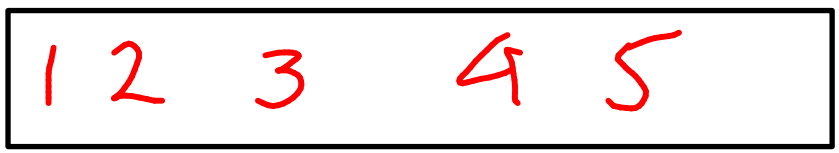
Complexity of insertion sort: $\sum_{i=1}^{n} i = n(n+1)/2 = O(n^2)$

Question: What if we used a sorted array instead of a sorted linked list & used binary search in "insert"?

I/P:



& assume insertion starting from first position of "sorted array" implementation of PQ



search can take $O(\log i)$ time for i^{th} insertion but shifting will anyways take $O(i)$ time!!

So time = $\sum_{i=1}^{n} (\log_2 i + i) = O(n^2)$

Q: Isn't there a way to minimize the "shifting" using an array implementation?

Ans: Yes. By inserting at "strategic" positions (as against "naively" at array beginning)

Hint: Try interpreting array as a HEAP! → free!

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to
     $$1 + 2 + \ldots + n$$
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	***Sequence** S*	***Priority queue** P*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
.	.	.
(g)	(2,3,4,5,7,8,9)	()

Insertion sort: Sorting using "sorted list"

Selection-Sort

: Sorting using "unsorted list"

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time

Thus both sorted & unsorted list implementations of PQ give $O(n^2)$ sorting algos!

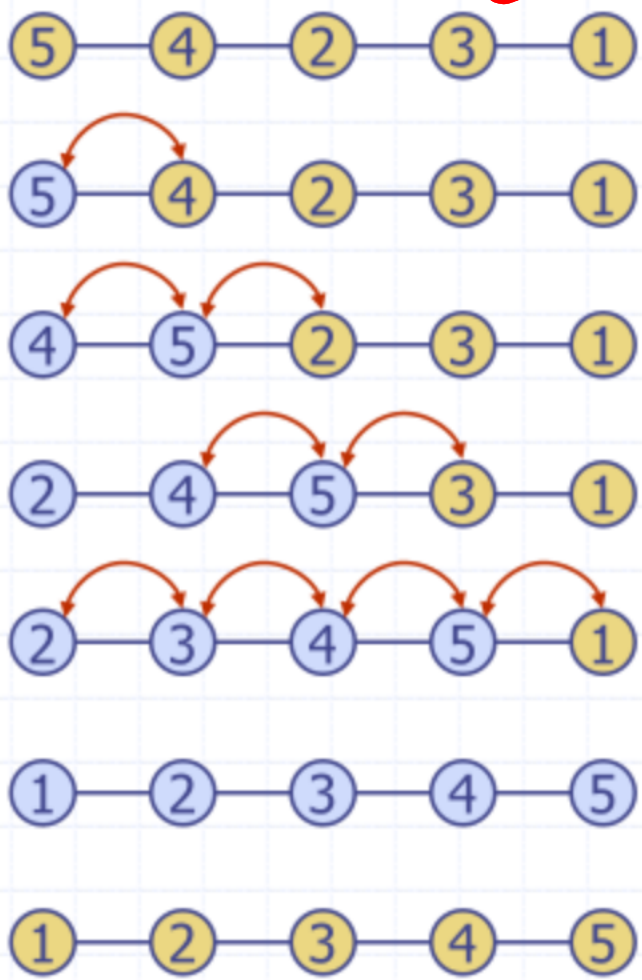
Selection-Sort Example

	***Sequence** S*	***Priority Queue** P*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
.	. .	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

More or less same when done in-place

In-place Insertion-sort / Selection-sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



↓ some steps skipped

Replacing "inserts" with "swaps" to avoid external PQ OS

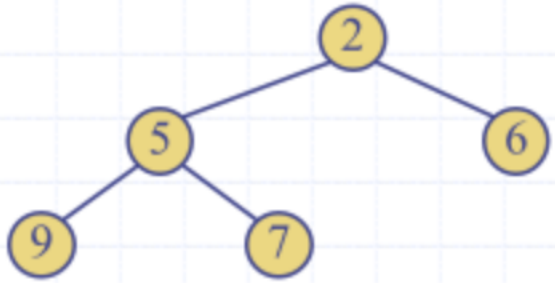
H/w:

(iv) We saw in class how insertion sort using a PQ can be modified into an in-place insertion sort. Along similar lines, modify the selection sort using a priority queue into an in-place selection sort

Inplace: Insertion = In-place sort using sorted list for PQ = swapping max element

Inplace : Selection = In-place sort using unsorted list for PQ = swapping min element

Heaps



Recall Priority Queue ADT (§ 7.1.3)

- A priority queue stores a collection of entries
- Each **entry** is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, x)
    inserts an entry with key k and value x
  - removeMin()
    removes and returns the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

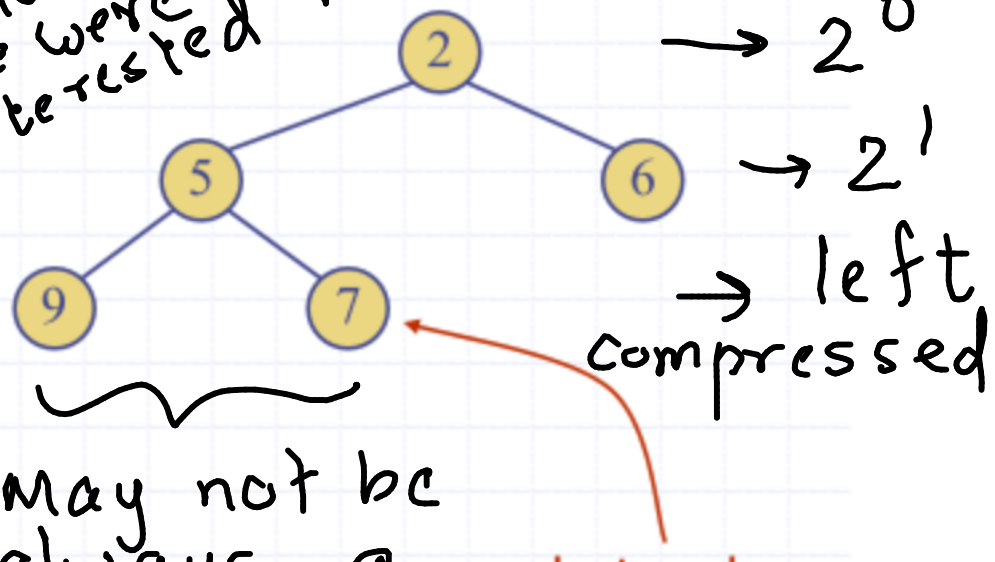
Recall Priority Queue Sorting (§ 7.1.4)

- We can use a priority queue to sort a set of comparable elements
  - Insert the elements with a series of insert operations
  - Remove the elements in sorted order with a series of removeMin operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: $O(n^2)$ time
  - Sorted sequence gives insertion-sort: $O(n^2)$ time
- Can we do better?

Algorithm *PQ-Sort*(S, C)
  Input sequence S, comparator C for the elements of S
  Output sequence S sorted in increasing order according to C
  $P \leftarrow$ priority queue with comparator C
  while $\neg S.isEmpty()$
    $e \leftarrow S.remove(S.first())$
    $P.insertItem(e, e)$
  while $\neg P.isEmpty()$
    $e \leftarrow P.removeMin()$
    $S.insertLast(e)$

Heaps (§7.3)

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
    - at depth $h-1$, the internal nodes are to the left of the external nodes
- The last node of a heap is the rightmost node of depth h



Since we were interested in "sorted array"

$\rightarrow 2^0$

$\rightarrow 2^1$

$\rightarrow$ left compressed

May not be always a "proper" tree

implies (by transitivity of "$\geq$") that key(v) $\geq$ every ancestor(v)

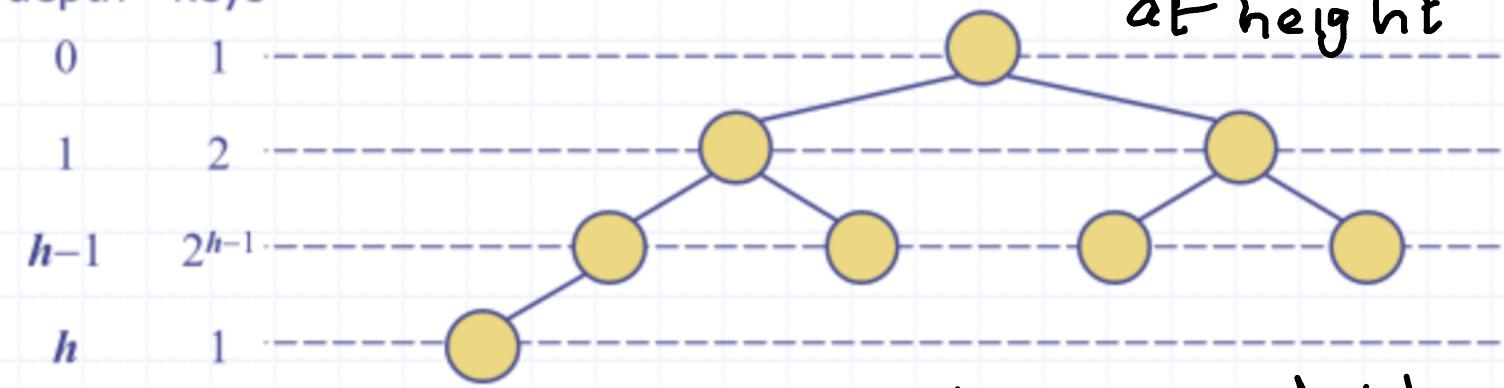
Height of a Heap (§ 7.3.1)

- Theorem: A heap storing n keys has height $O(\log n)$

  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$

$1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$

→ Since tree has height h, there must be atleast 1 node at height h

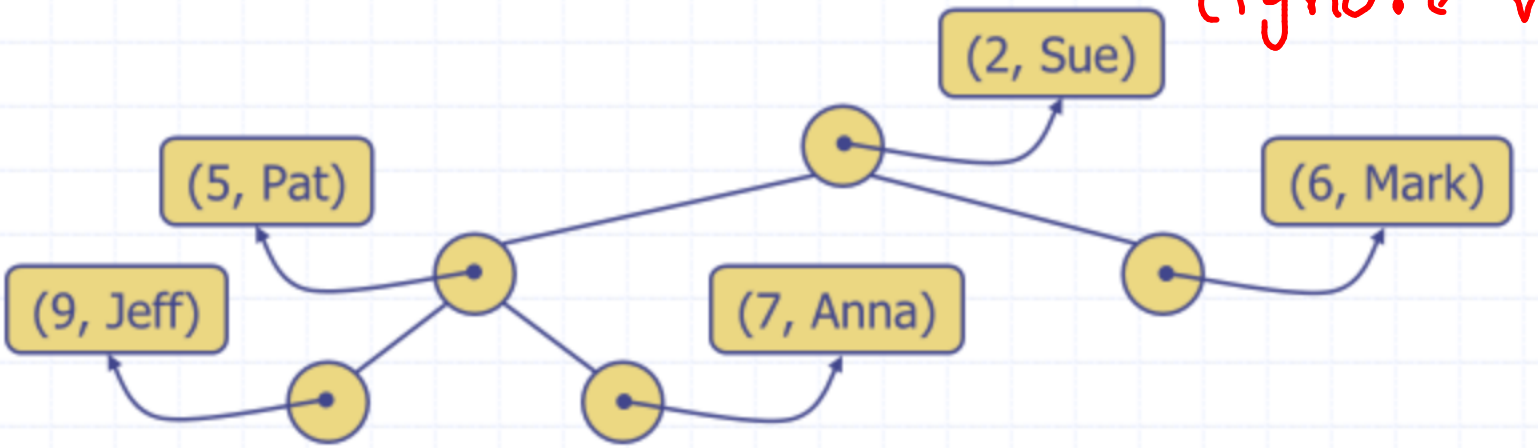


Also: $n < 1 + 2 + 4 \ldots + 2^h + 1 = 2^{h+1}$

$\log n + 1 \geq h + 1 > \log n \Rightarrow h = \lfloor \log n \rfloor$

Heaps and Priority Queues

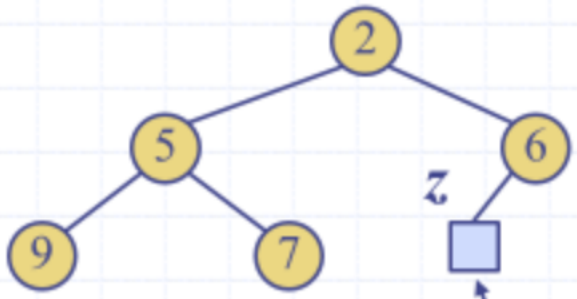
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node & root
- For simplicity, we show only the keys in the pictures (ignore values)



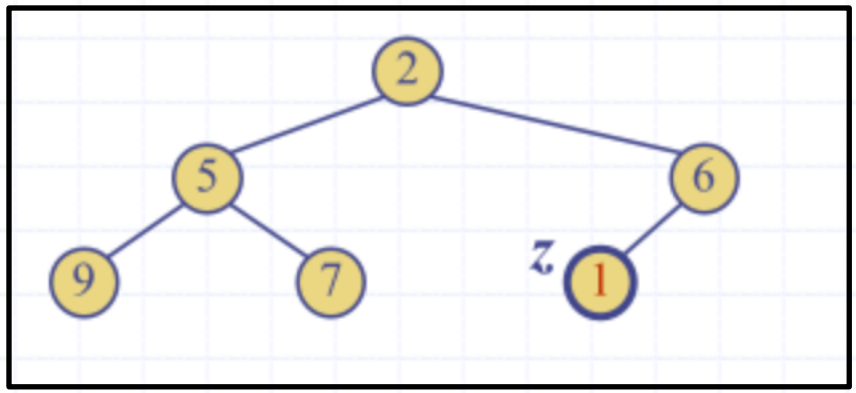
Insertion into a Heap (§ 7.3.3)

- Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)

positions of last node in array + 1



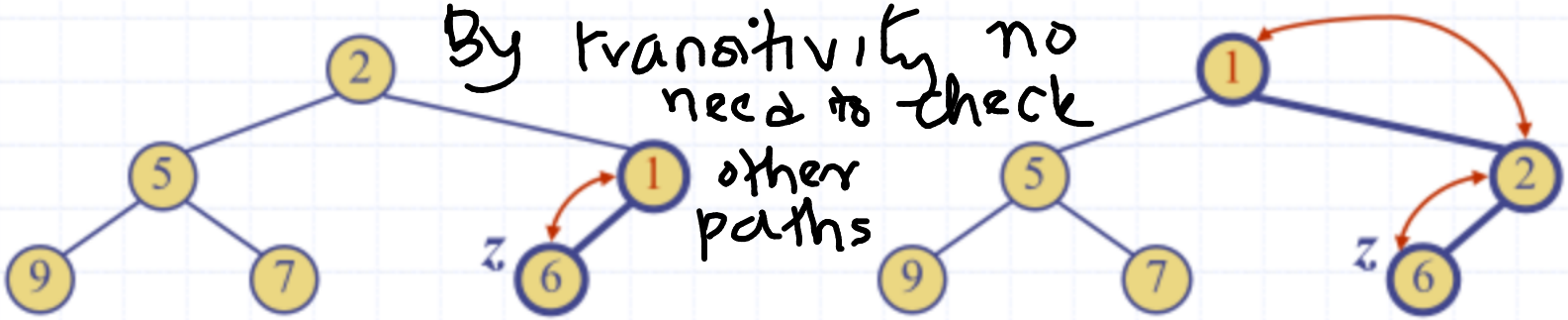
insertion node



Violates "Heap order"

Upheap [Restore heap property for the single violating path]

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time

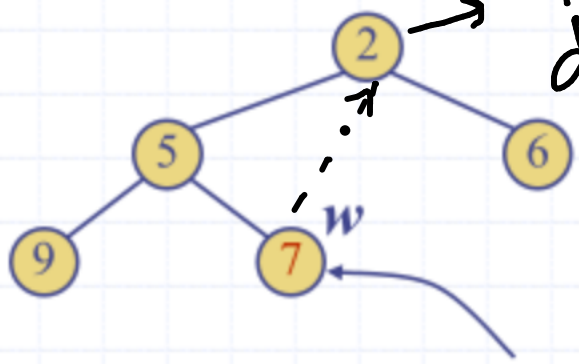


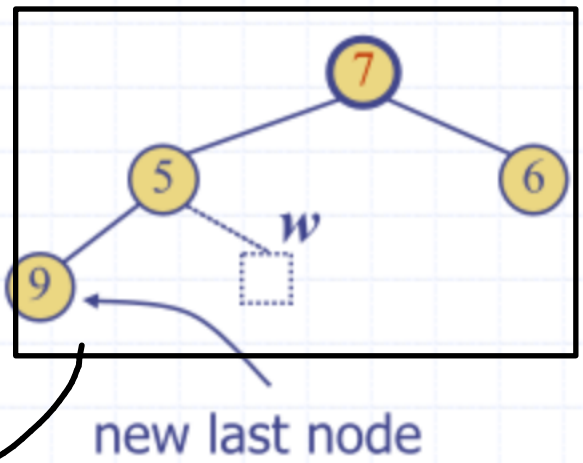
By transitivity no need to check other paths

By transitivity of "≤" no need to check further!

Removal from a Heap (§ 7.3.3)

- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)





Removing 2 destroys bin tree prop.

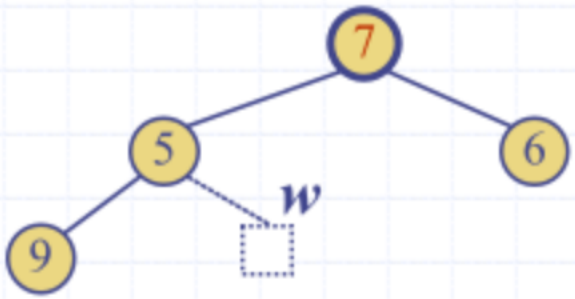
Restore bin tree prop & violate heap prop

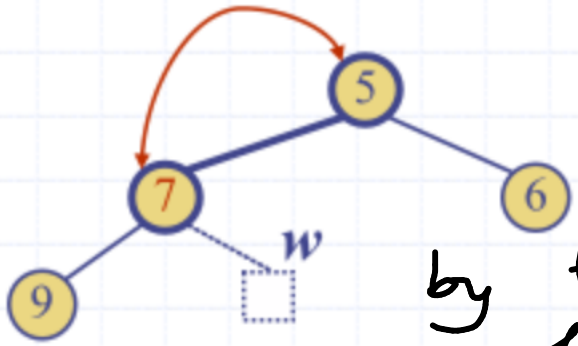
Down heap moves "7" down the path of "smaller" values

Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

path is determined by smaller keys





Ensures correctness by transitivity of "≥"