Comp202 Midterm Review

Topics We Have Covered So Far

1. Introduction (Why does this class exist?)
2. OO-Programming Basics
3. Analysis
4. Recursion
5. Arrays (Static, Dynamic)
6. Linked Lists (Single, Double)
7. List ADT
8. Stack ADT, Queue ADT
9. Tree ADT, Binary Tree ADT, Tree Traversal, Tree Implementations
10. Binary Search Trees, AVL, 24, RB Trees
11. Priority Queue ADT, Heaps
12. Maps, Hashing

Analysis

- Given that `foo(i,j)` runs in $O(1)$ time, calculate the asymptotic time complexities of the following

1.
```
for(i = 0; i < n; i +=10)
    for(j = 0; j < i; j++)
          foo(i,j);
```

2.
```
for(i = 0; i < n; i +=10)
    for(j = 1; j < n; j *= 2)
          foo(i,j);
```

Recursion

- Pseudo code for recursive squaring?

$$p(x,n)=\begin{cases} 1 & \text{if n} = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n>0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n>0 \text{ is even} \end{cases}$$

- Time complexity?

The java.util.List ADT

- The java.util.List interface includes the following methods:

size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns a boolean indicating whether the list is empty.
get(i):	Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.
set(i, e):	Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.
add(i, e):	Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}()]$.
remove(i):	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

main methods

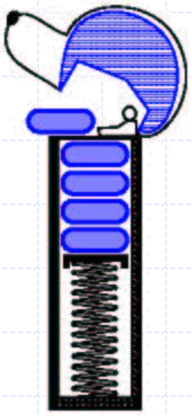
Implementing the List ADT

- With arrays?
- With linked lists?
- With trees?

- Complexities of the methods in each implementation

The Stack ADT

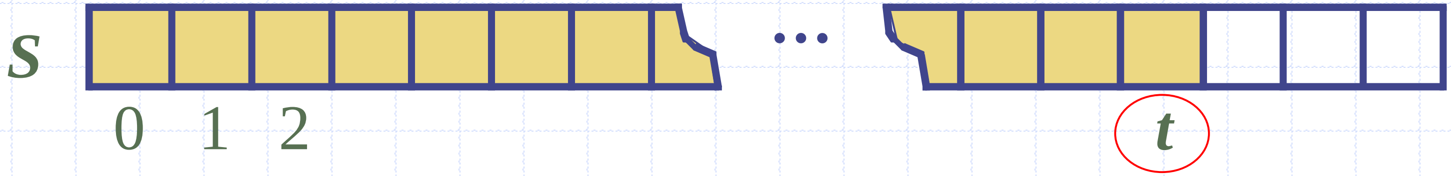
- The Stack ADT stores arbitrary objects
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Insertions (push) and deletions (pop) follow the last-in first-out (LIFO) scheme



- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Array-based Stack

- A simple way of implementing the Stack ADT is to use an array
- We add elements from left to right
- A variable keeps track of the index of the top element



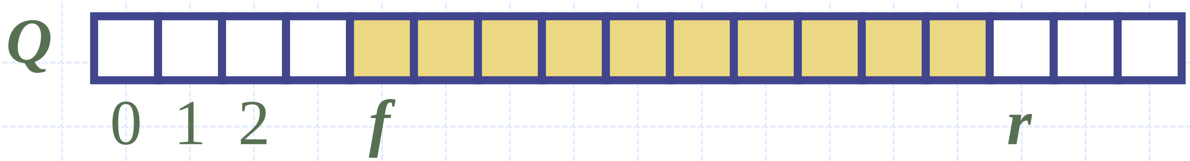
The Queue ADT

- The Queue ADT stores arbitrary objects
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Auxiliary queue operations:
  - object first(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Boundary cases:
  - Attempting the execution of dequeue or first on an empty queue returns null

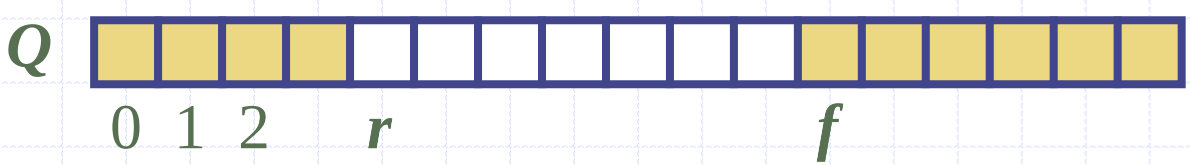
Array-based Queue

- Use an array of size N in a **circular** fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

normal configuration



wrapped-around configuration



Tree ADT

- Hierarchical collection of nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterable children(p)
  - Integer numChildren(p)
- Generic methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- The BinaryTree ADT extends the Tree ADT with additional methods:
  - node left(p)
  - node right(p)
  - node sibling(p)

These methods could be part of a given node or the tree class itself…

Euler Tour Traversal - BTs

Algorithm *eulerTour* (T, p)

pre-visit (p)

pl = *left* (p)

if $pl \neq$ **null**

eulerTour (T, pl)

visit (p)

pr = *right* (p)

if $pr \neq$ **null**

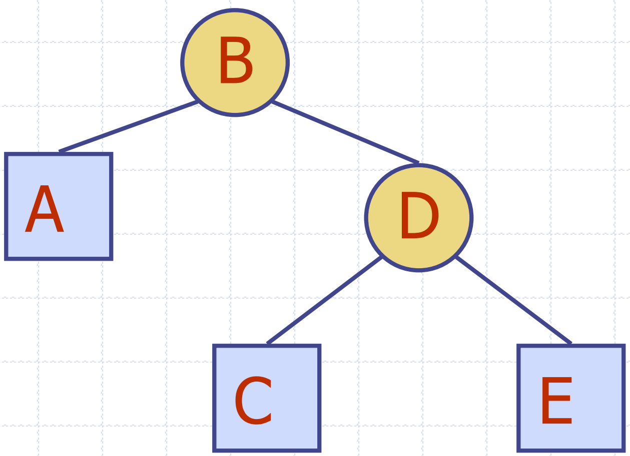
eulerTour (T, pr)

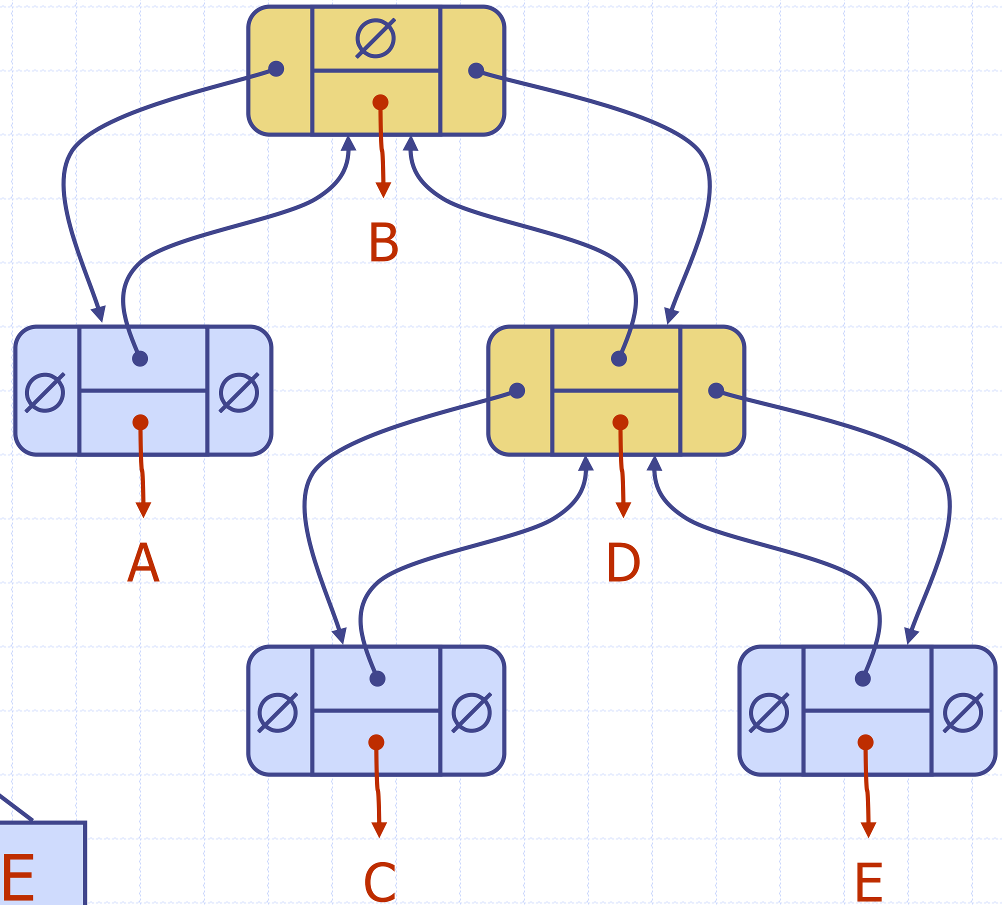
post-visit (p)

Pre-order
In-order
Post-order

Linked Structure for Binary Trees

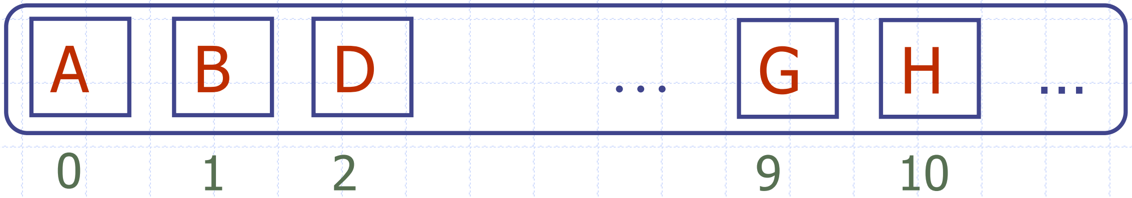
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

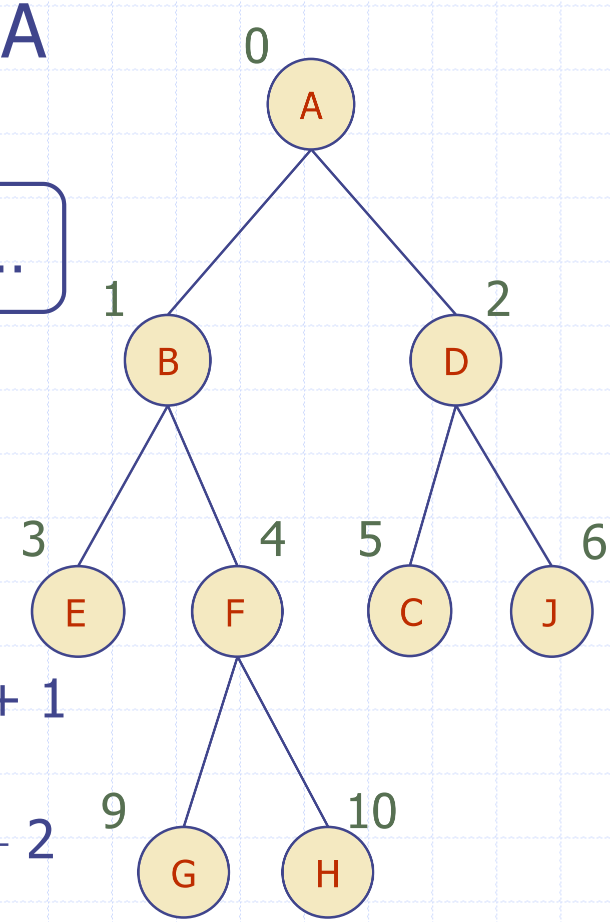




Array-Based Representation of Binary Trees

- Nodes are stored in an array A

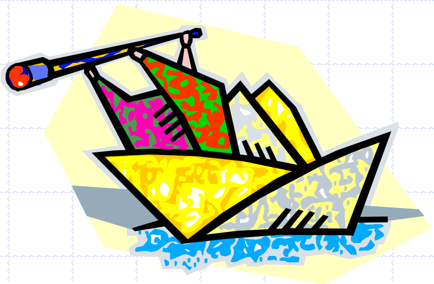




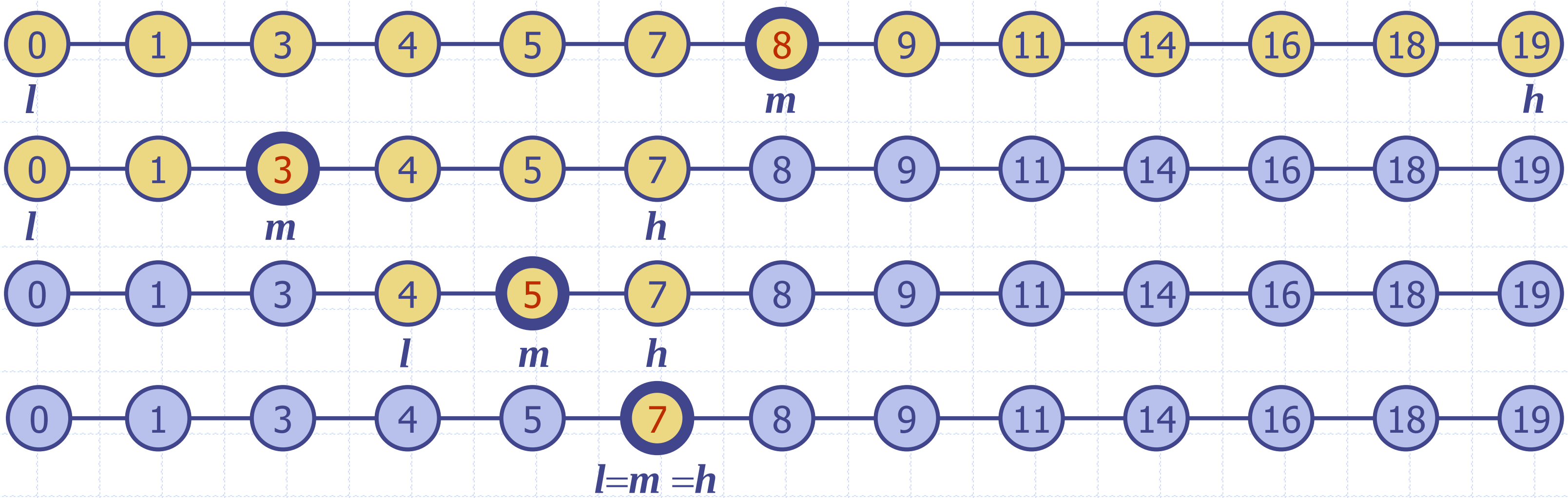
Node v is stored at A[rank(v)]

- rank(root) = 0
- if node is the left child of parent(node),
  rank(node) = 2 · rank(parent(node)) + 1
- if node is the right child of parent(node),
  rank(node) = 2· rank(parent(node)) + 2

Binary Search



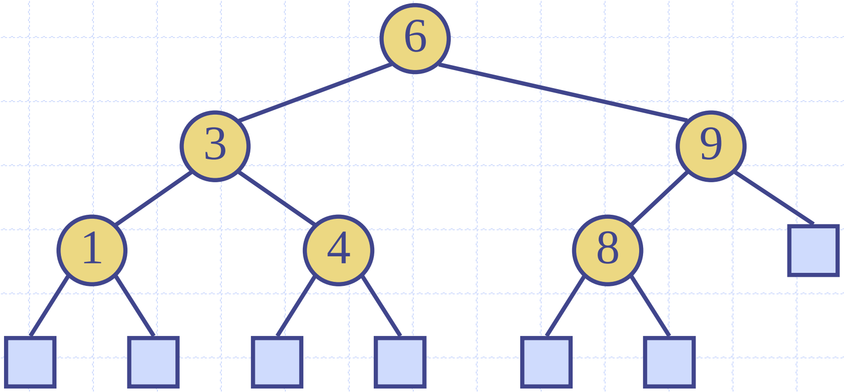
- Example: find(7)



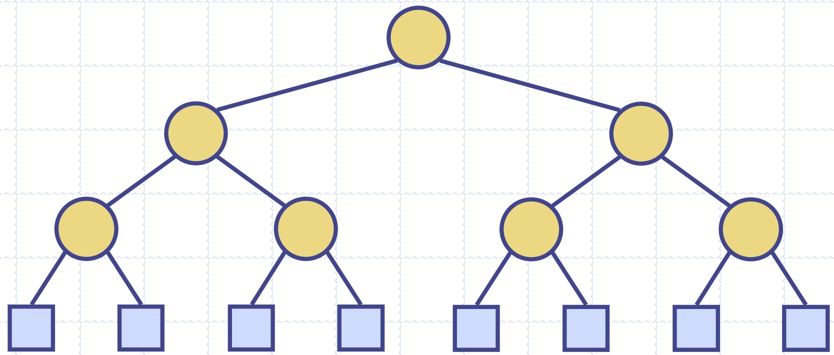
Binary Search Trees

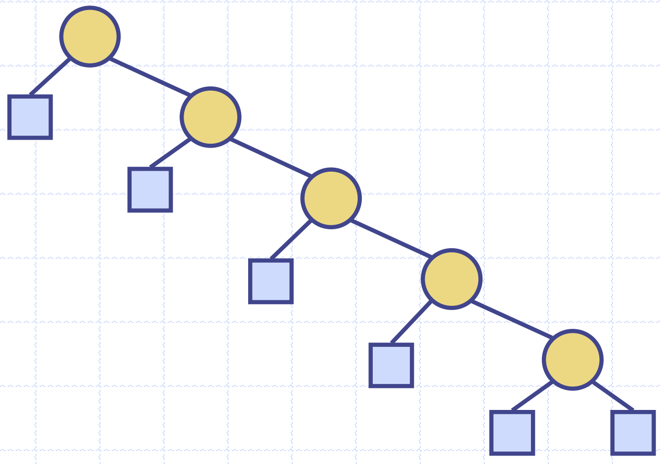
- Most important property?
- What type of traversal "makes sense"?
- Search?
- Insert
- Delete?

- E.g.: find(4), add(5), add(2), remove(3)



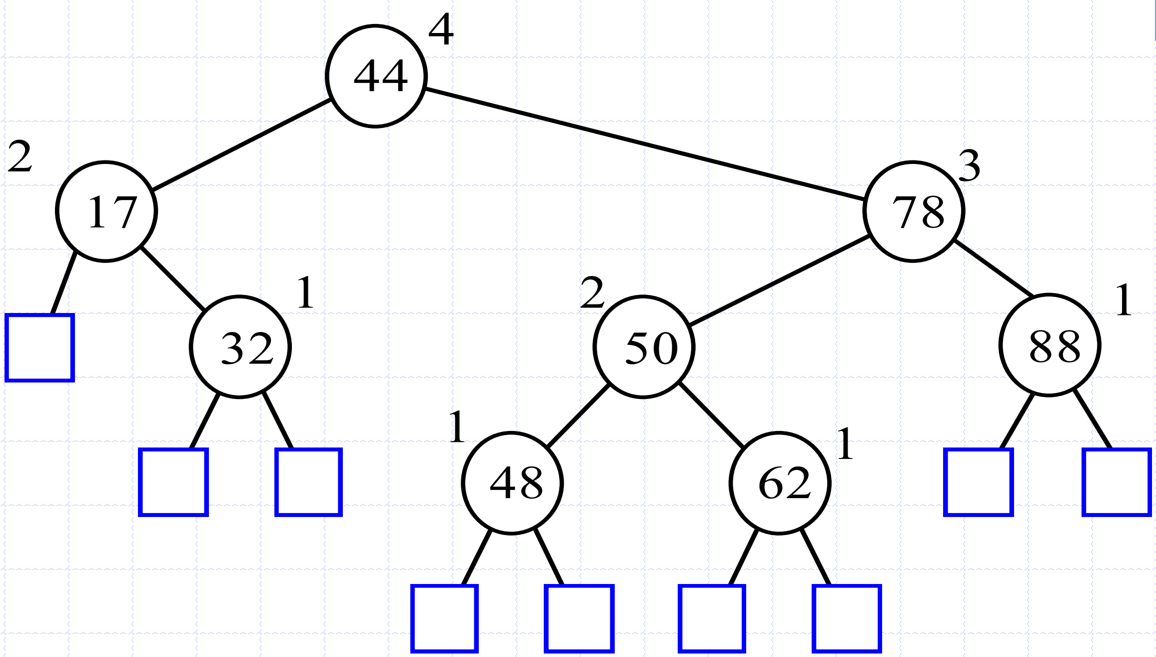
Performance





AVL Trees

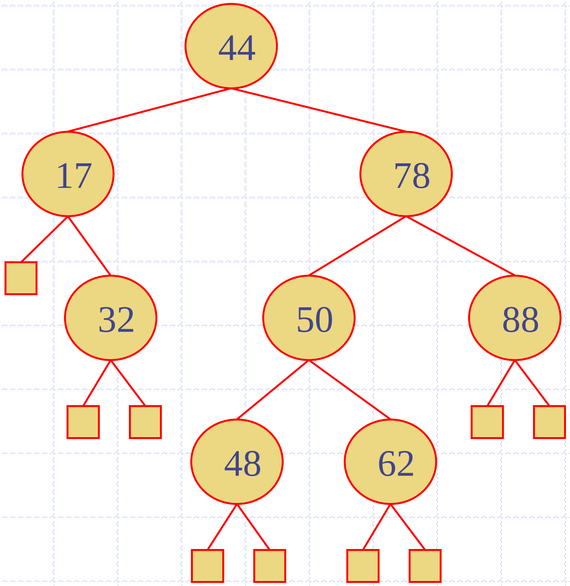
- AVL Trees are balanced binary search trees that satisfy the height-balance property:
- For every internal node v of T, heights of the children of v can differ by at most 1
- Any subtree of an AVL tree, is itself an AVL tree



An example of an AVL tree where the heights are shown next to the nodes

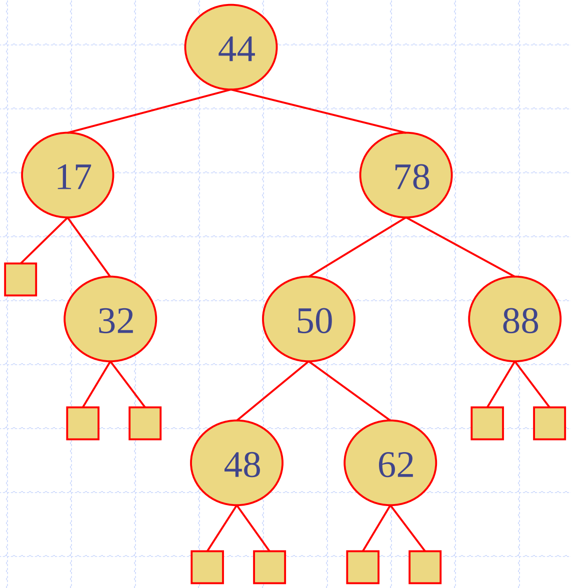
Insertion and Deletion

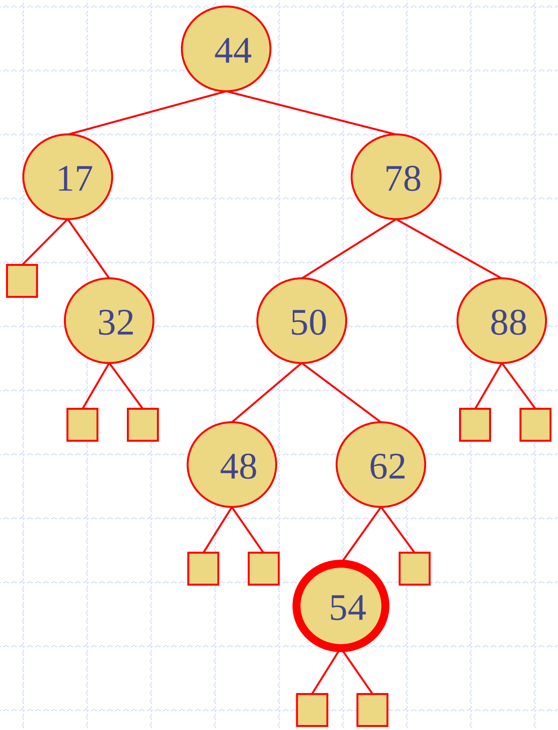
- Find where to insert/remove
- Insert/remove as if it was an ordinary BST
- Re-balance if needed (only need to check the ancestors)



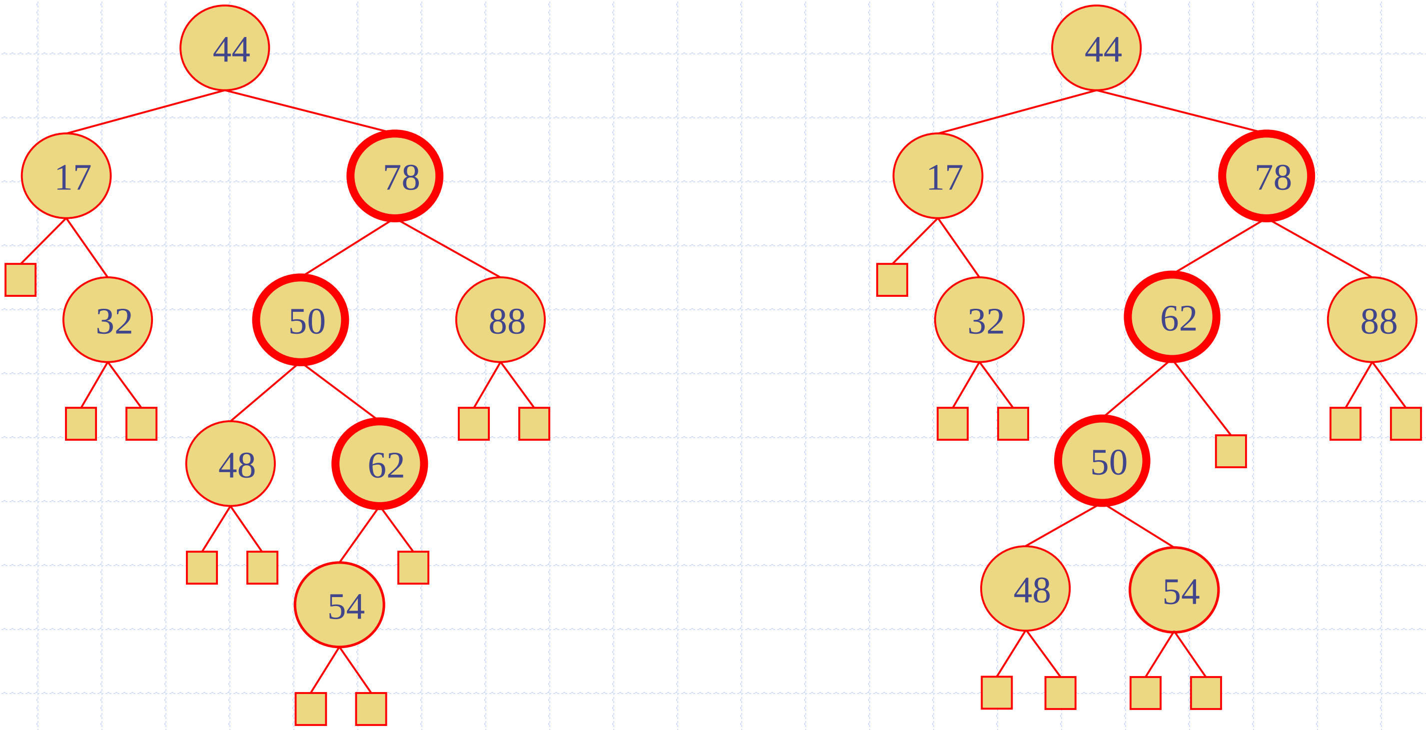
Insert 54

AVL Insertion

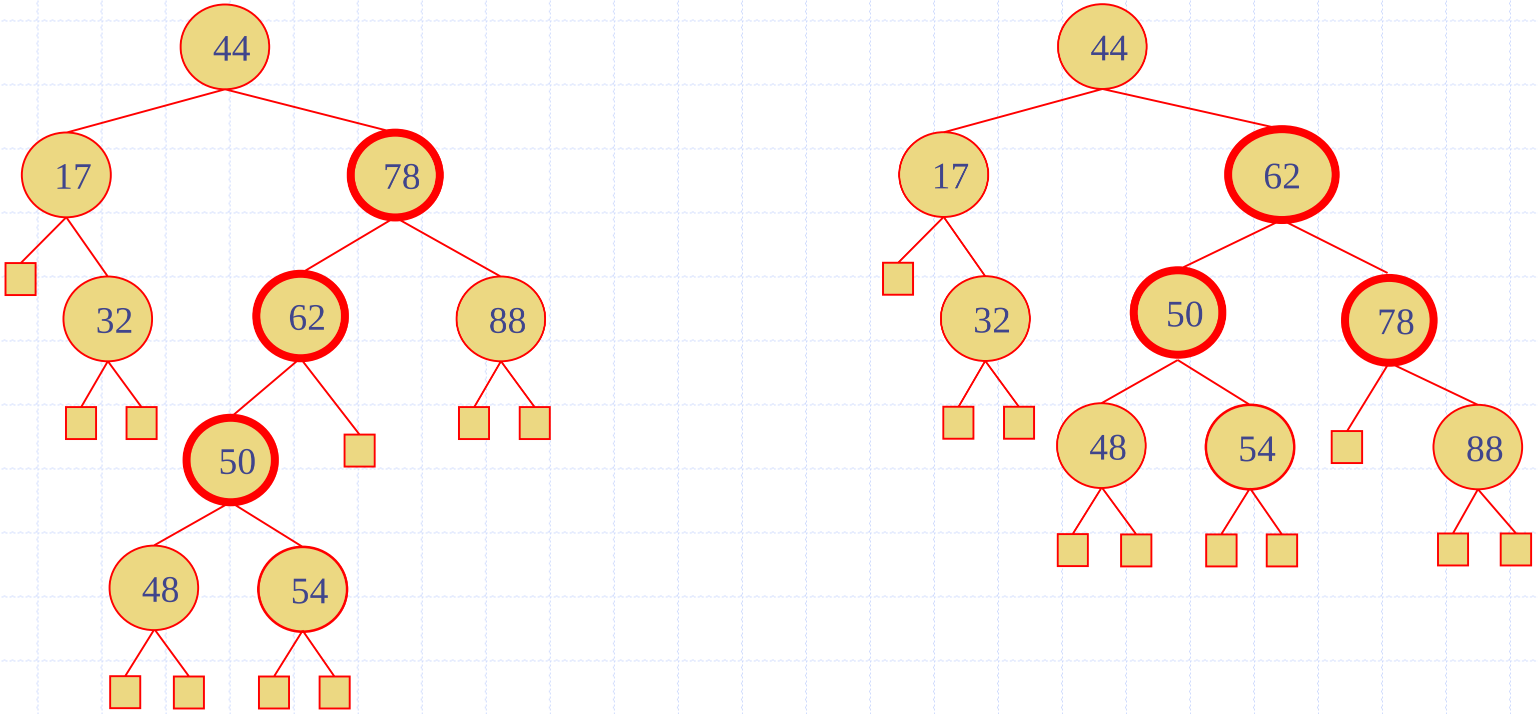




AVL Insertion

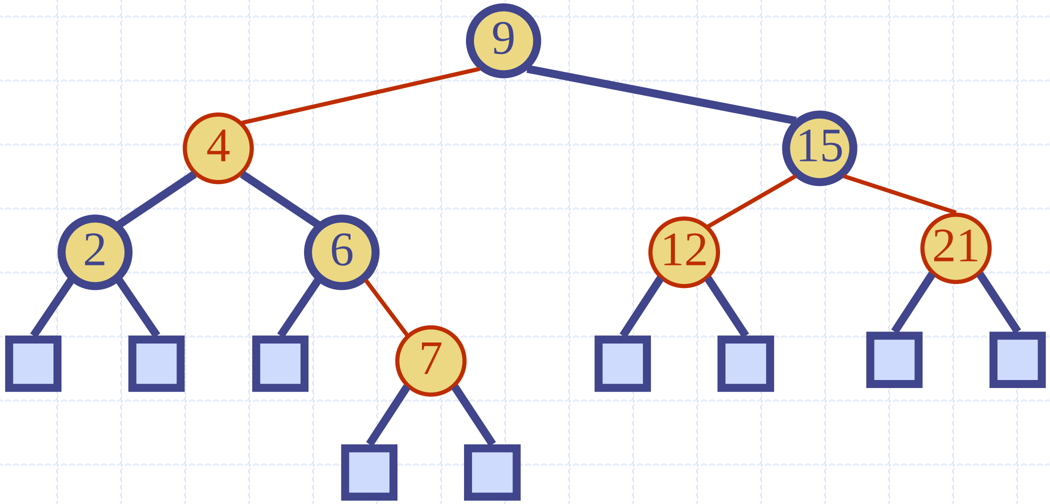


AVL Insertion



Red-Black Trees

- A red-black tree can also be defined as a binary search tree that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black
  - Depth Property: all the leaves have the same black depth



23

Insertion

Algorithm *insert*(*k*, *o*)

1. We search for key ***k*** to locate the insertion node ***z***

2. We add the new entry (***k***, ***o***) at node ***z*** and color ***z*** red

3. **while** ***doubleRed***(*z*)

   if ***isBlack***(***sibling***(***parent***(*z*)))

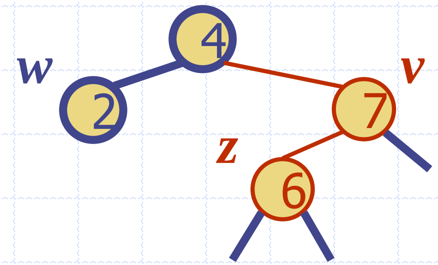
   z ← ***restructure***(*z*)

   return

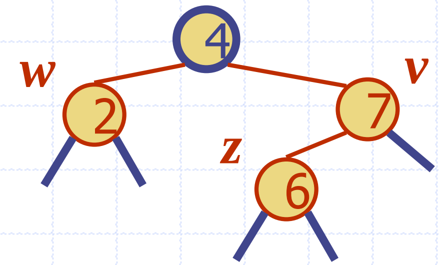
   else { ***sibling***(***parent***(*z*)) is red }

   z ← ***recolor***(*z*)

Case 1:



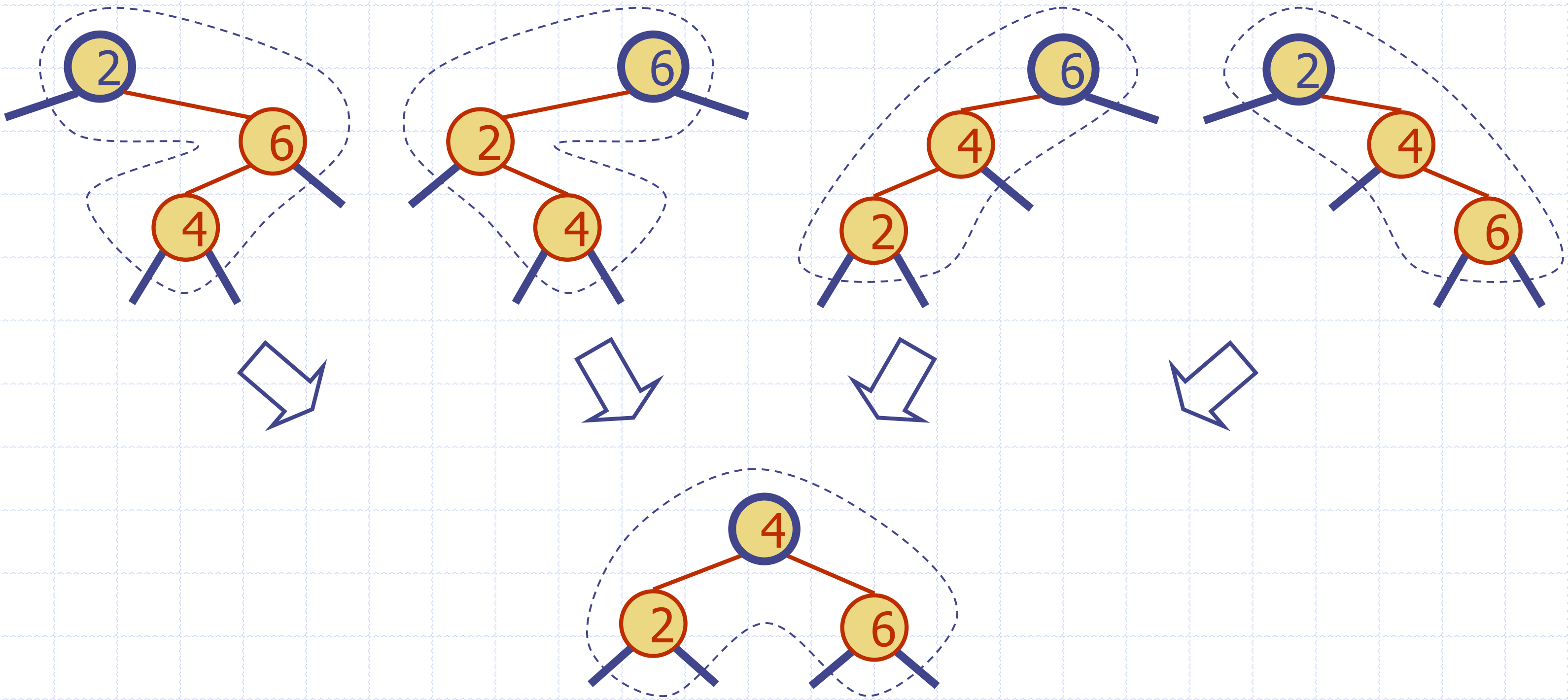
Case 2:



24

Restructuring

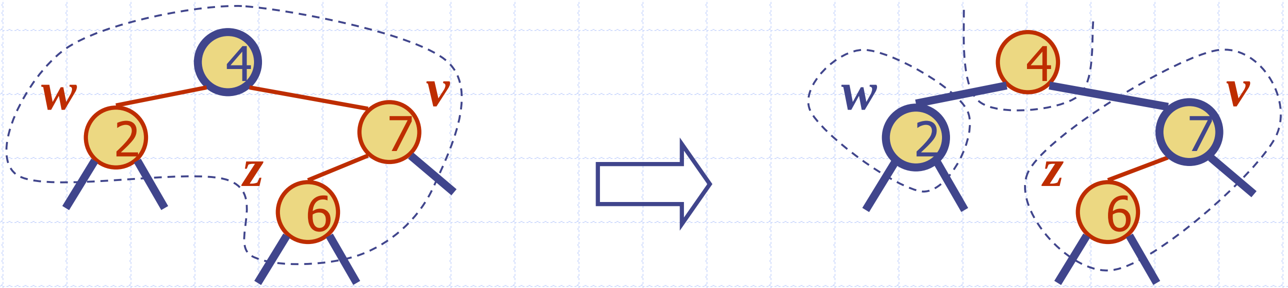
- There are four restructuring configurations depending on whether the double red nodes are left or right children



25

Recoloring

- The parent v and its sibling w become black and the grandparent u becomes red, unless it is the root
- The double red violation may propagate to the grandparent u



26

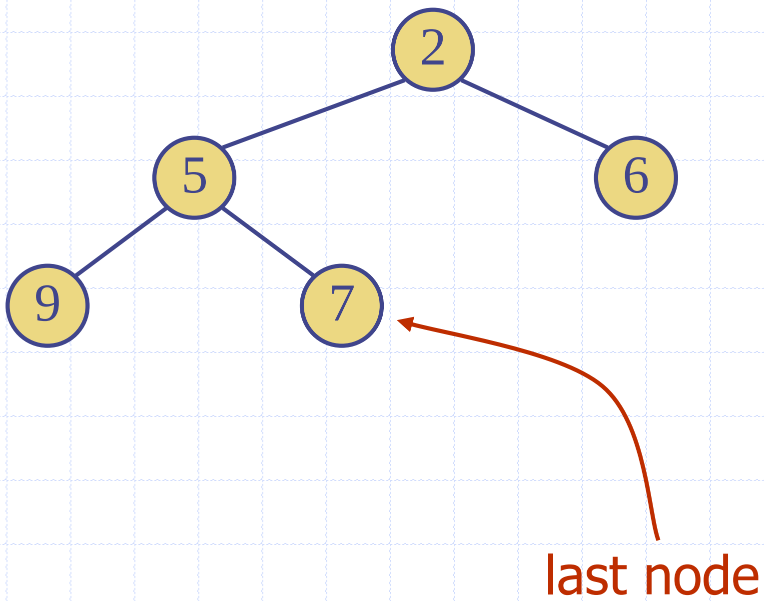
Priority Queue ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, v) inserts an entry with key k and value v
  - removeMin() removes and returns the entry with smallest key, or null if the the priority queue is empty
- Additional methods
  - min() returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size(), isEmpty()
- Note that the max version is also possible

Heaps

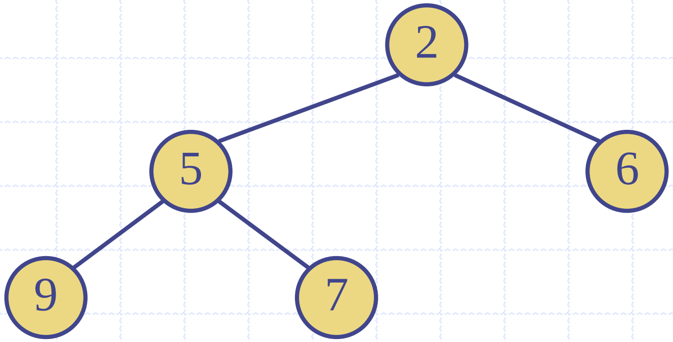
- A heap is a binary tree satisfying:
- Heap-Order:

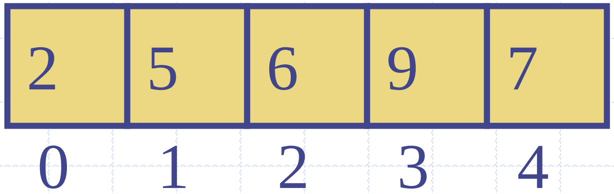
  $key(v) \geq key(parent(v))$
- Complete Binary Tree:
- The last node of a heap is the rightmost node of maximum depth



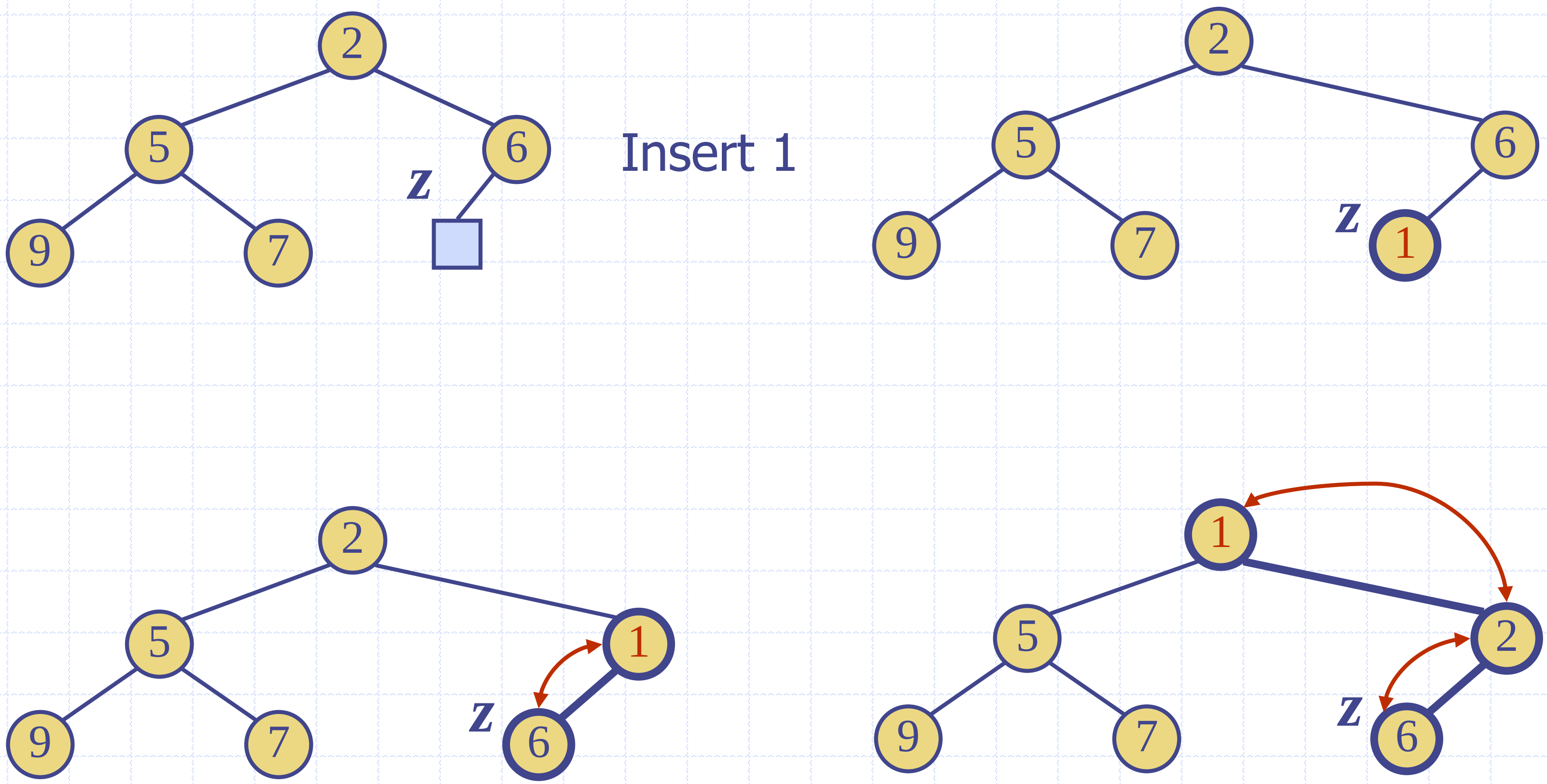
Array-based Heap Implementation

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- Operation add corresponds to inserting at rank $n + 1$
- Operation remove_min corresponds to removing at rank n, after the swap





Heap Insertion



Priority Queue Sorting

Algorithm *PQ-Sort*(S, C)

Input list S, comparator C for the elements of S

Output list 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.insert (*e*, $\varnothing$)

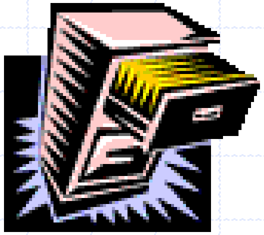
while $\neg$*P.isEmpty*()

e $\leftarrow$ *P.removeMin*().*getKey*()

S.addLast(*e*)

Sorted List as PQT?
Unsorted List as PQT?
Heap as PQT?
AVL or RB Tree as PQT?

The Map ADT



- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

  h_2: integers $\rightarrow [0, N-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

  $$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" the keys in an apparently random way

Hashing

Hash Codes:

- Memory address

- Polynomial accumulation

$k = a_0\, a_1 \ldots a_{n-1}$

$h(k) = a_0 + a_1 z + a_2 z^2 + \ldots$
$\quad + a_{n-1} z^{n-1}$

at a fixed value z, ignoring overflows

Compression:

- Divison

$h(k) \bmod N$

- MAD

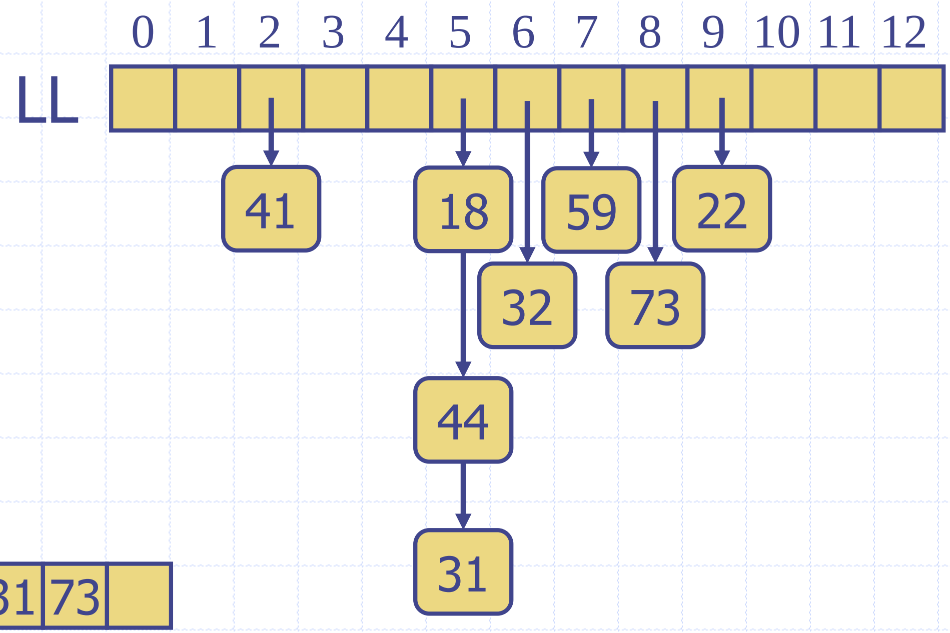
$(\, \mathrm{abs}(a \cdot h(k) + b) \bmod P\,) \bmod N$

Collision Handling

- Separate Chaining: Buckets are containers
  - Linked-lists
  - Balanced Binary Search Trees (Java 8)
- Open Addressing: Check other buckets
  - Linear probing (check the adjacent ones in circular fashion)
  - Double Hashing (calculate the offset using another hash function)

Example

k	$h(k)$	$d(k)$
18	5	3
41	2	1
22	9	6
44	5	5
59	7	4
32	6	3
31	5	4
73	8	4



	0	1	2	3	4	5	6	7	8	9	10	11	12
LP			41			18	44	59	32	22	31	73	

	0	1	2	3	4	5	6	7	8	9	10	11	12
DH	31		41			18	32	59	73	22	44		