



Ch.14 Tournament Trees

© copyright 2006 SNU IDB Lab.



BIRD'S-EYE VIEW (0)

- Chapter 12: Binary Tree
- Chapter 13: Priority Queue
 - Heap and Leftiest Tree
- Chapter 14: Tournament Trees
 - Winner Tree and Loser Tree



BIRD'S-EYE VIEW

- A **tournament tree** is a complete binary tree that is most efficiently stored by using the array-based binary tree
- Study two varieties of tournament trees
 - Winner tree
 - Loser tree
- Tournament Tree Application
 - Bin packing

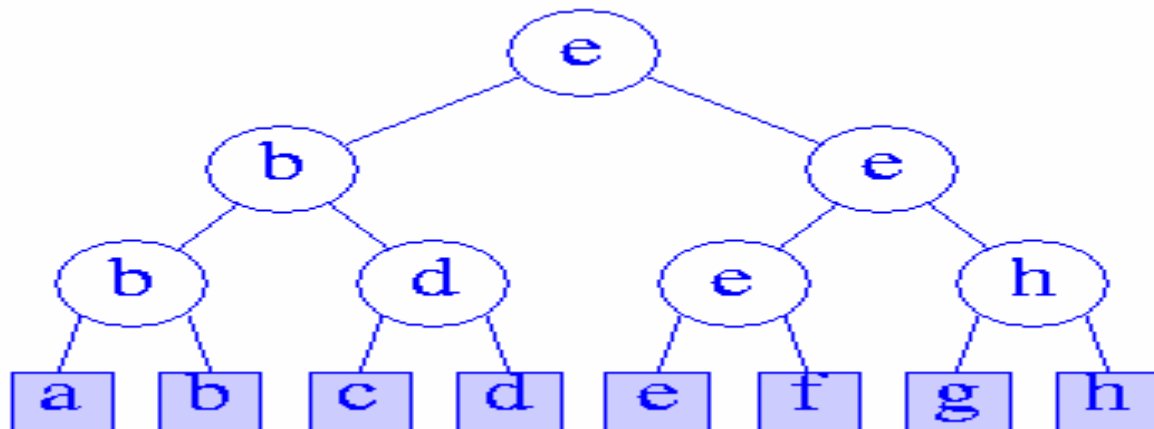


Table of Contents

- Winner Tree
- Loser Trees
- Tournament Tree Applications
 - Bin Packing Using First Fit (BPFF)
 - Bin Packing Using Next Fit (BPNF)

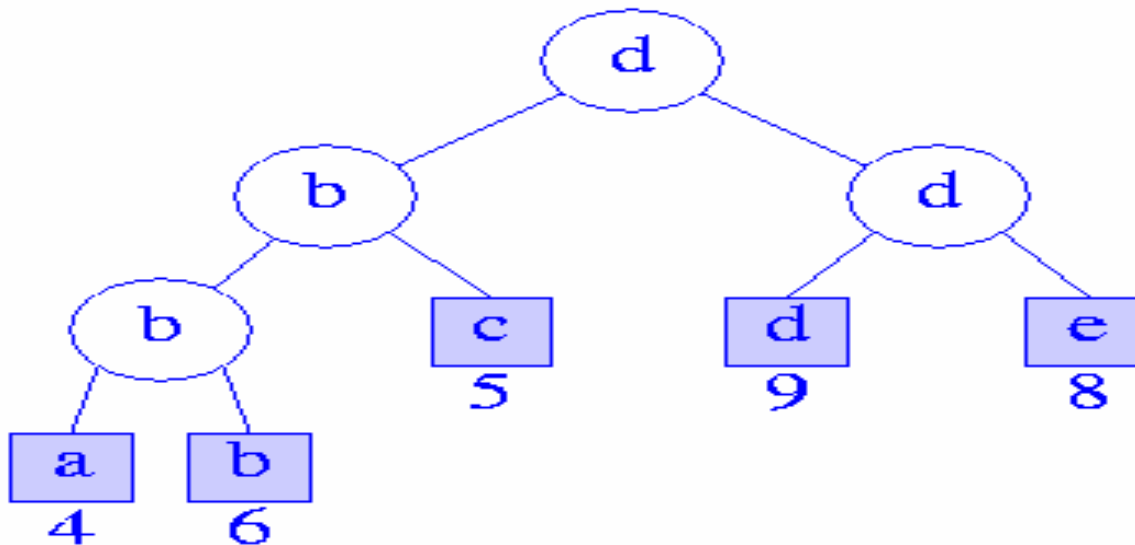
Winner Tree

- Definition: A winner tree for n players is a complete binary tree with n external and $n-1$ internal nodes
- Each internal node records the winner of the match played there.



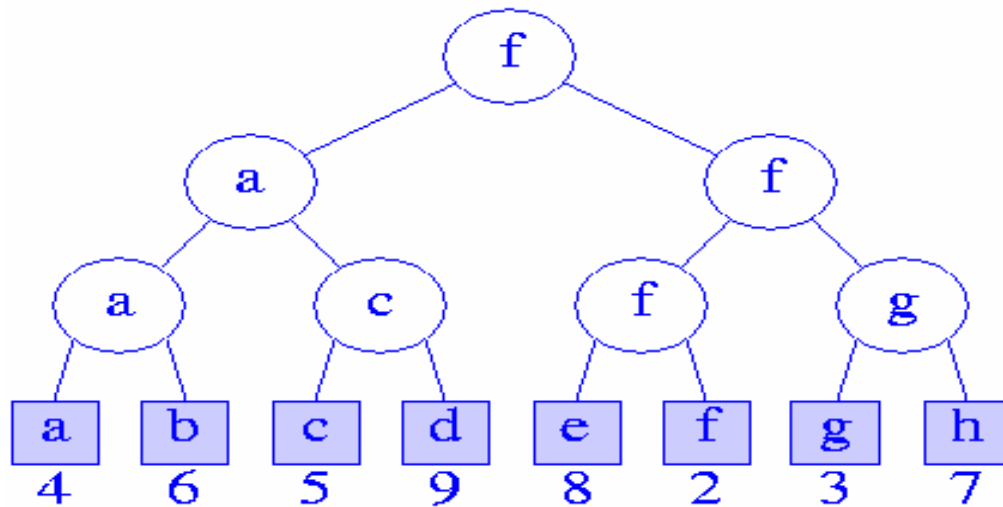
Max Winner tree

- The player with the larger value wins



Min Winner tree

- The player with the smaller value wins



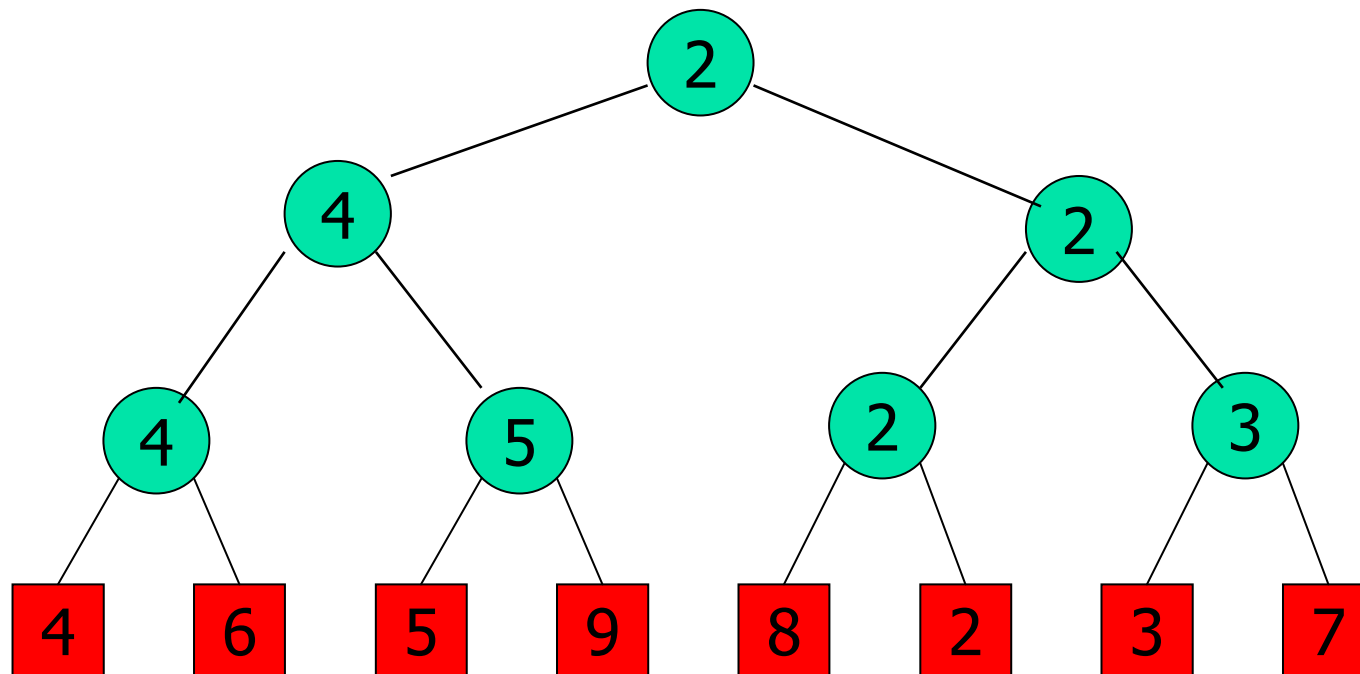


Complexity of Winner Tree

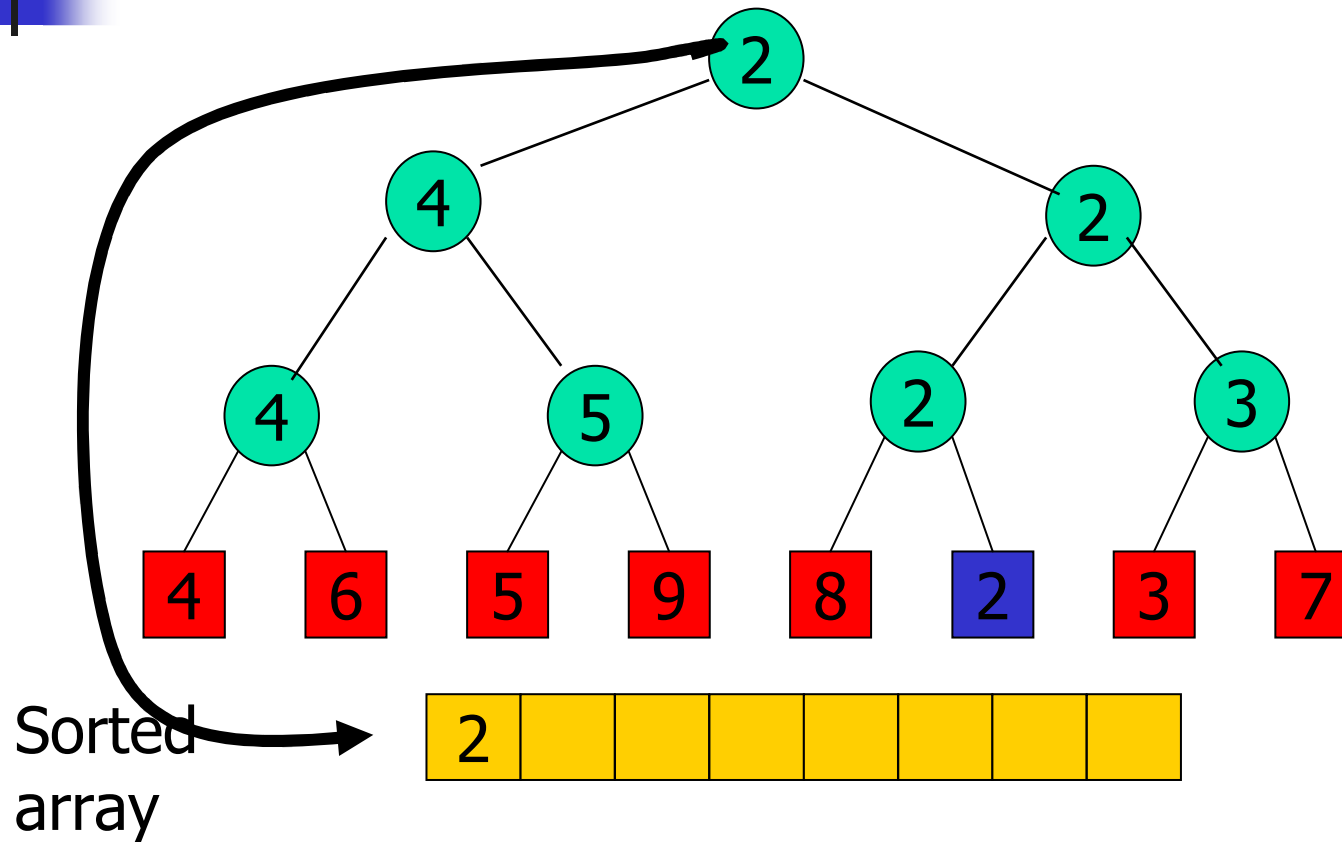
- $O(\log(n))$ time to **restructure** n player winner tree
- $O(1)$ time to play match at each match node
- $n - 1$ match nodes
- $O(n)$ time to **initialize** n player winner tree

WT Applications – Sorting (1)

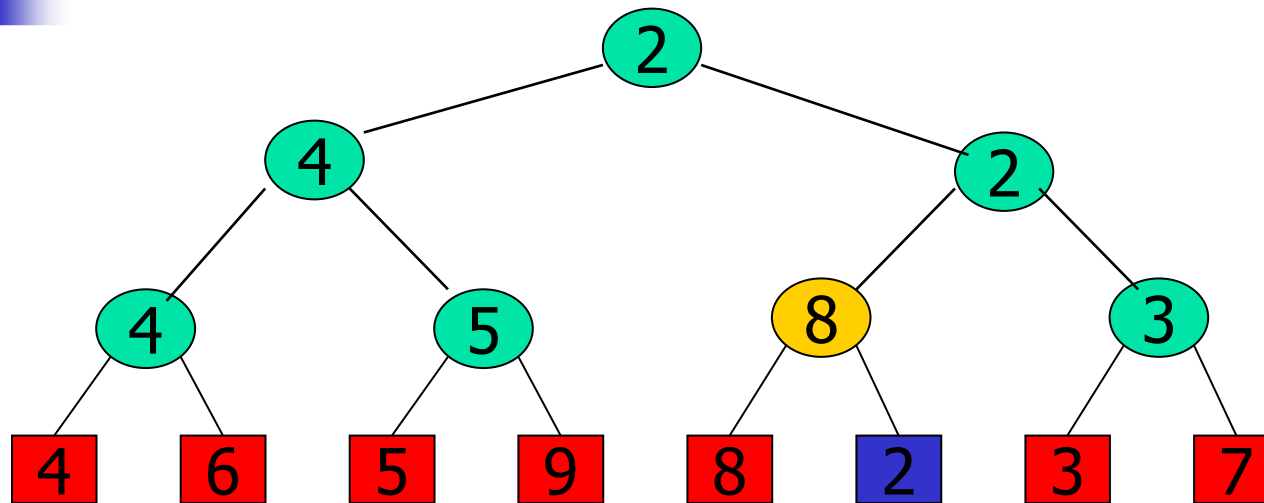
Min Winner Tree



WT Applications – Sorting (2)



WT Applications – Sorting (3)

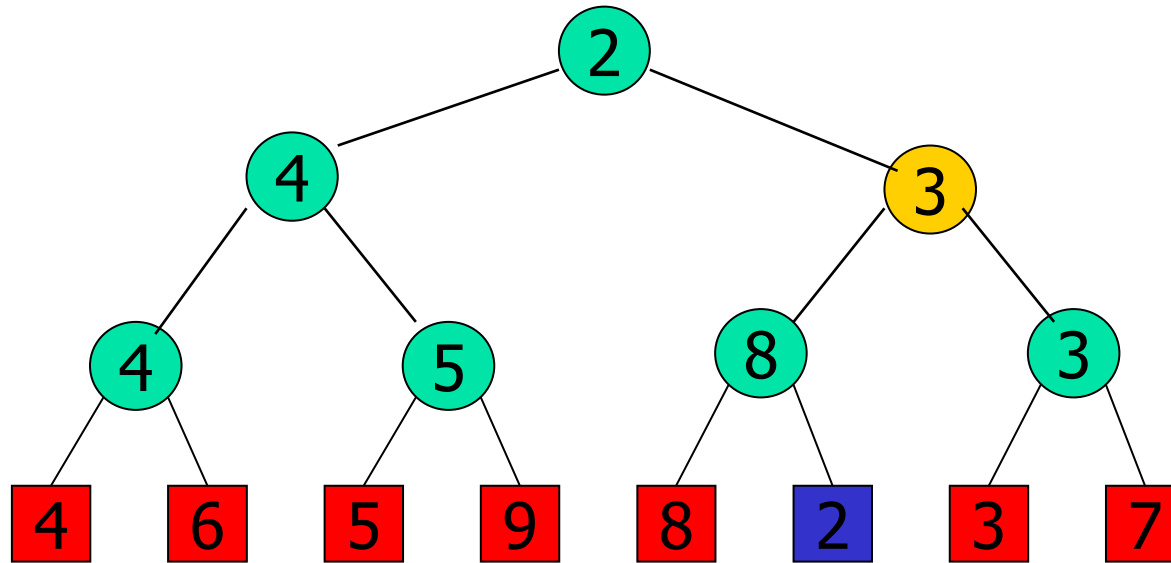


Sorted
array



- Restructuring starts at the place where "2" is removed

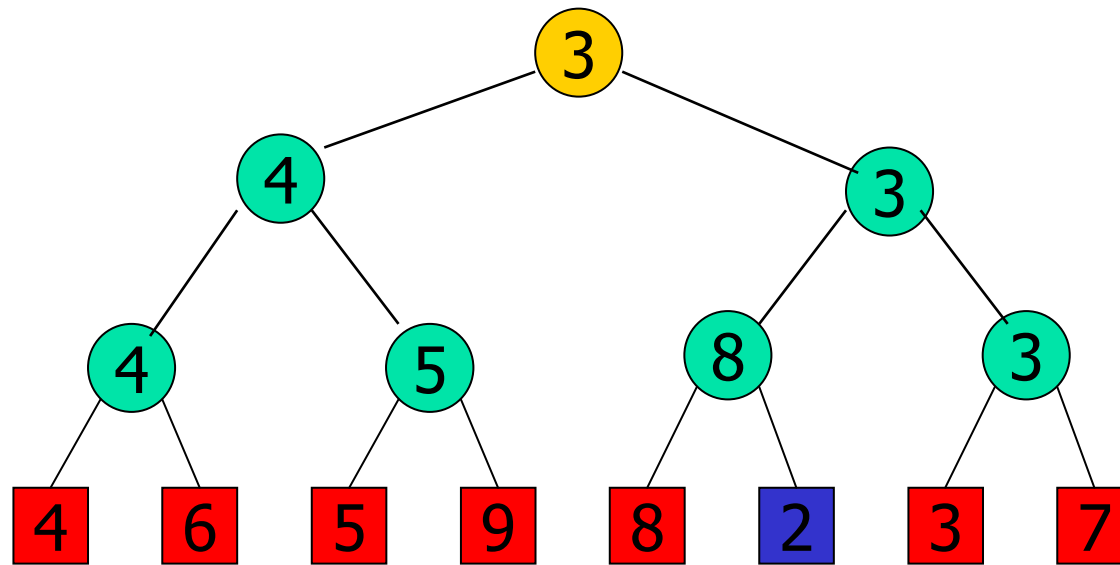
WT Applications – Sorting (4)



Sorted
array



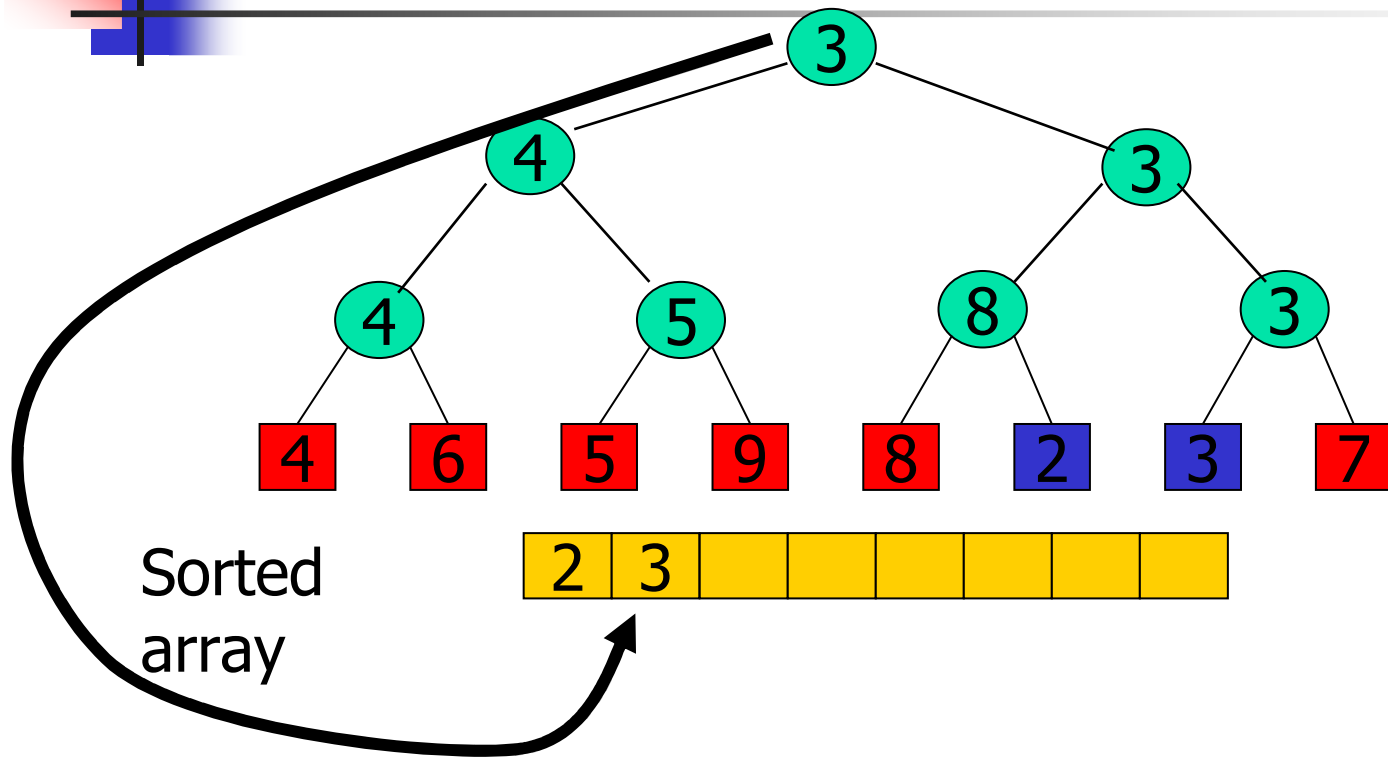
WT Applications – Sorting (5)



Sorted
array

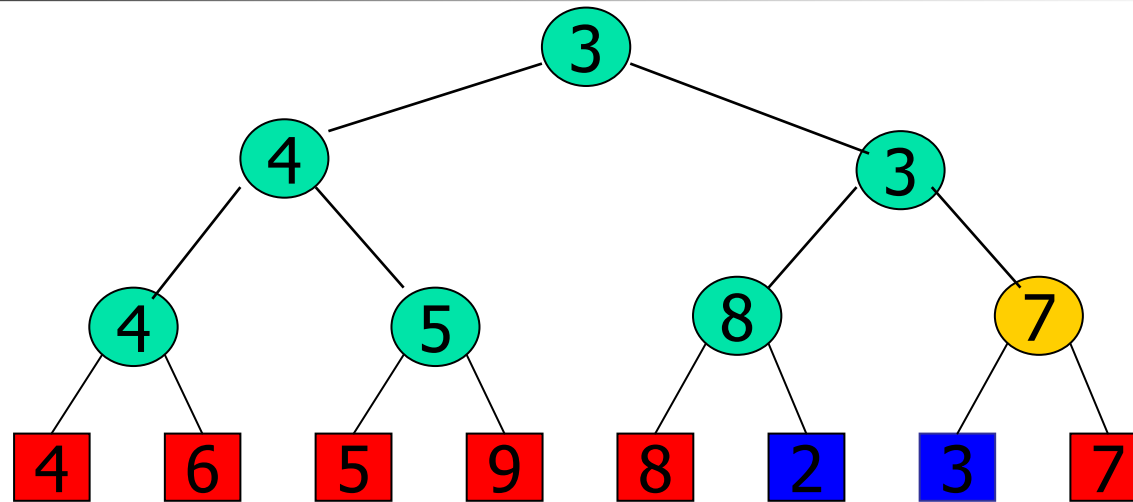


WT Applications – Sorting (6)



- Restructuring starts at the place where "3" is removed

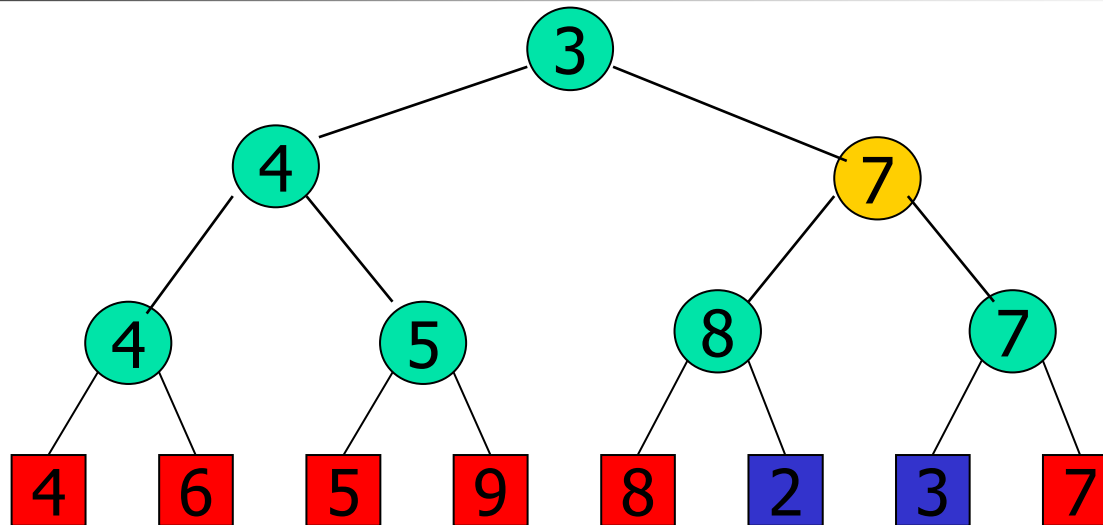
WT Applications – Sorting (7)



Sorted
array



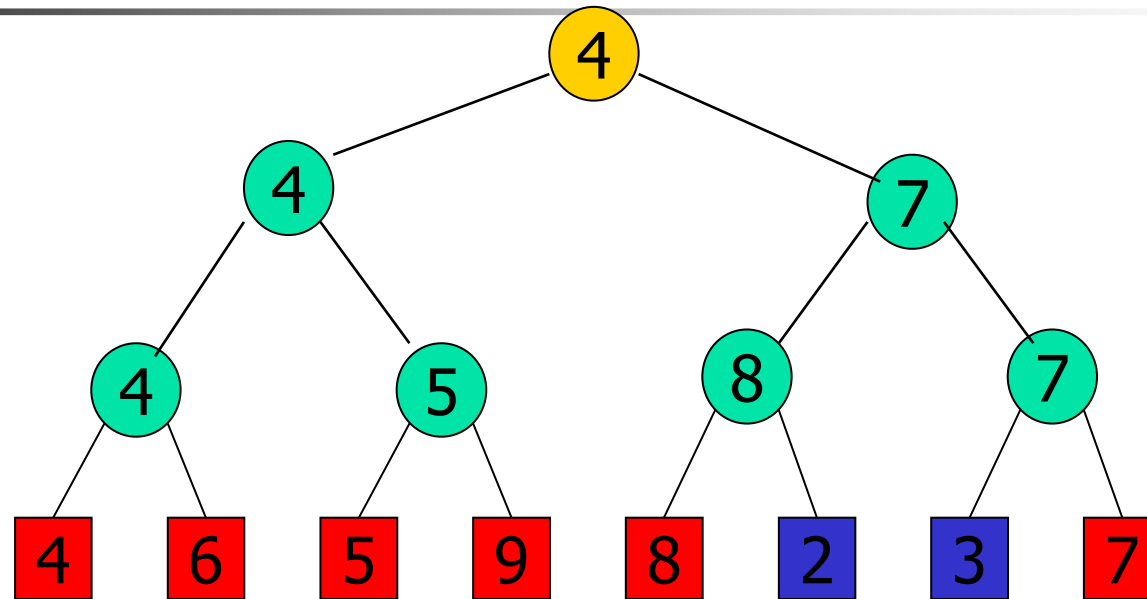
WT Applications – Sorting (8)



Sorted
array



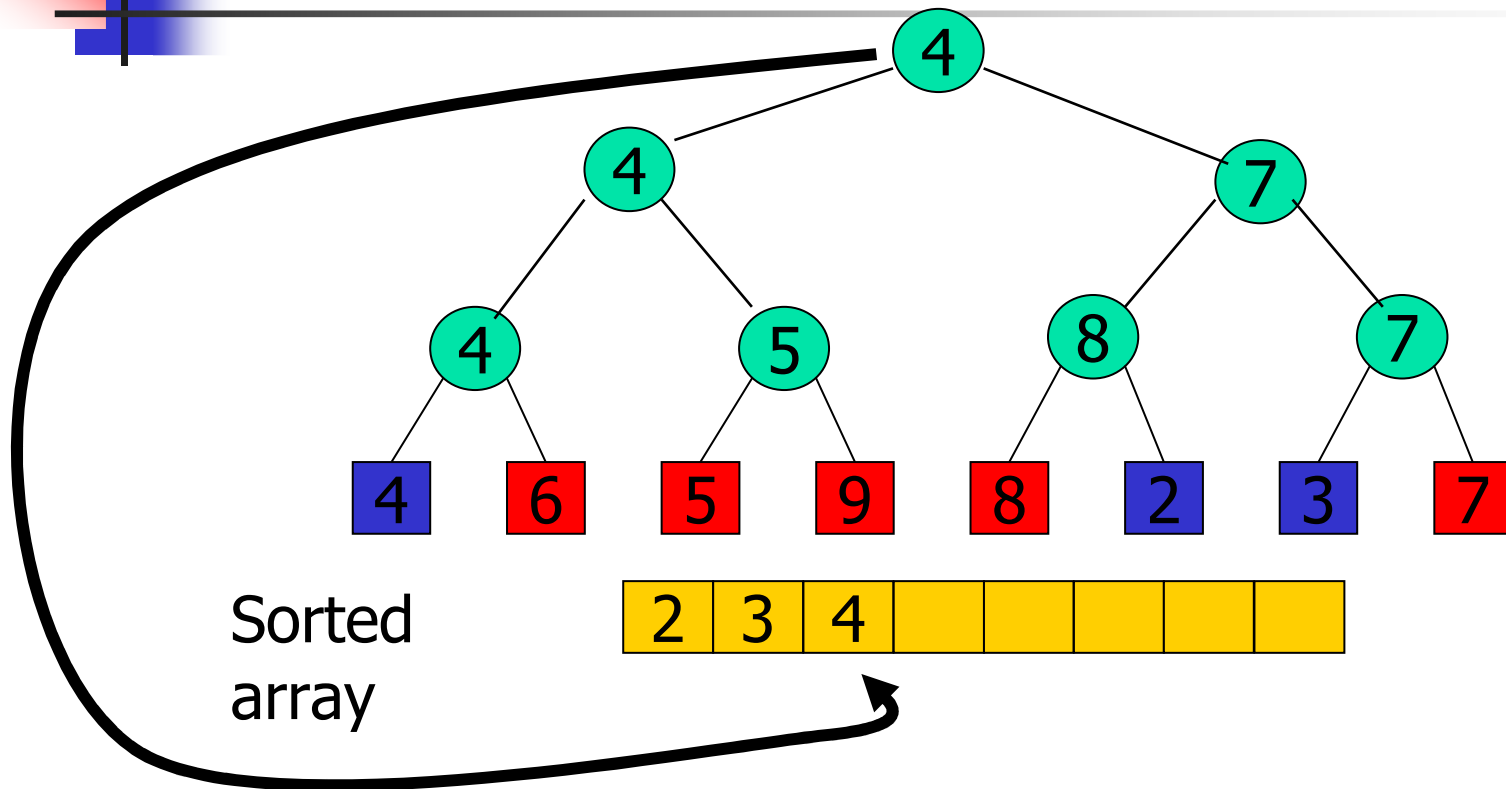
WT Applications – Sorting (9)



Sorted
array

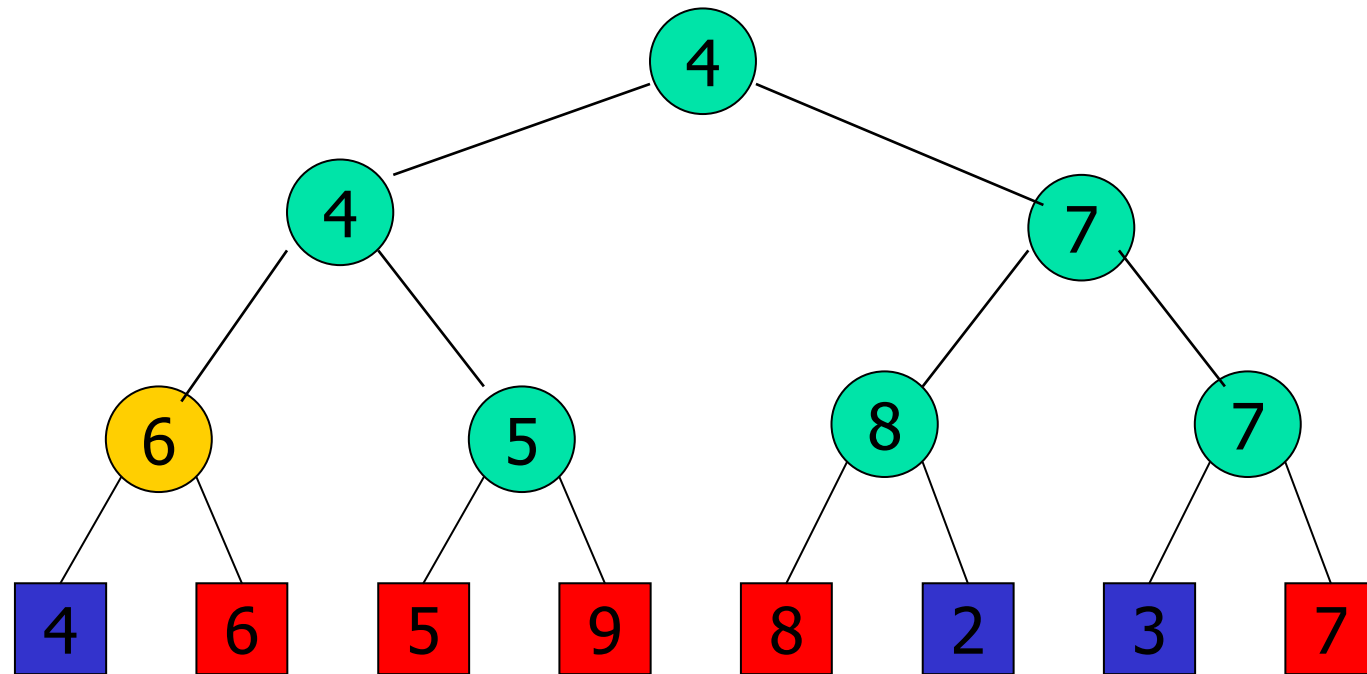


WT Applications – Sorting (10)



- Restructuring starts at the place where "4" is removed

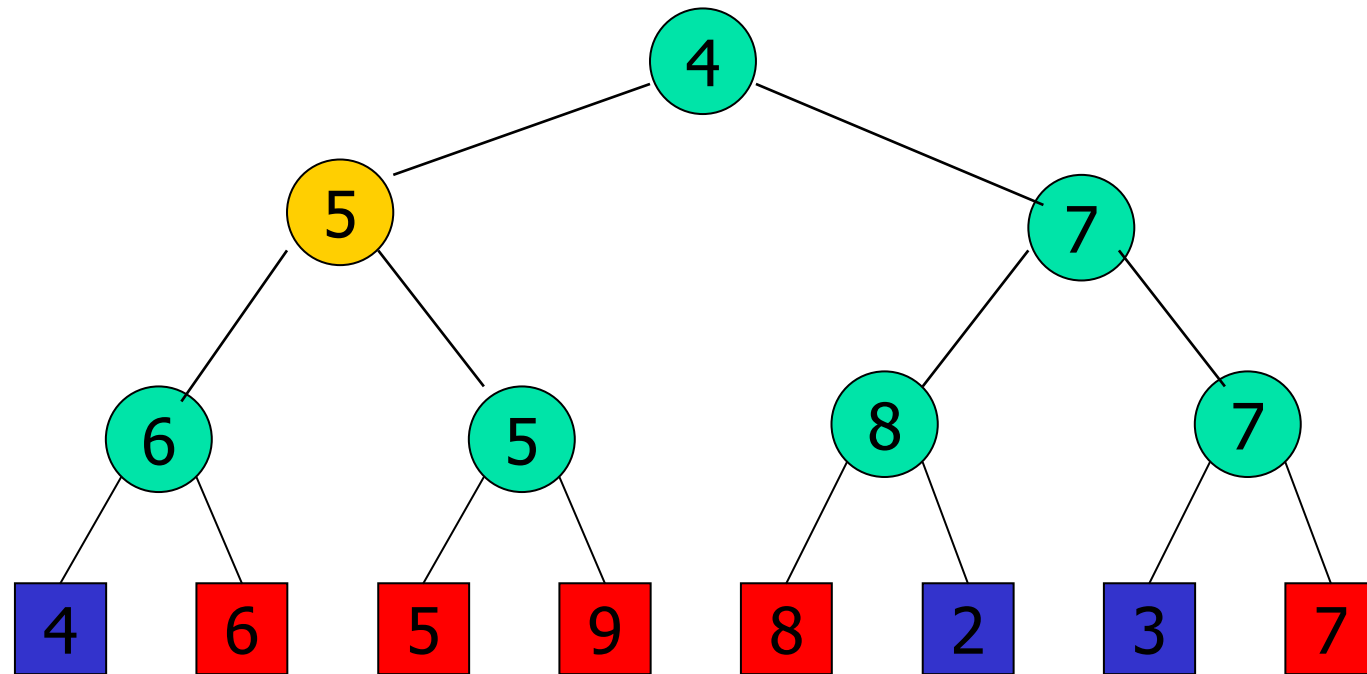
WT Applications – Sorting (11)



Sorted array



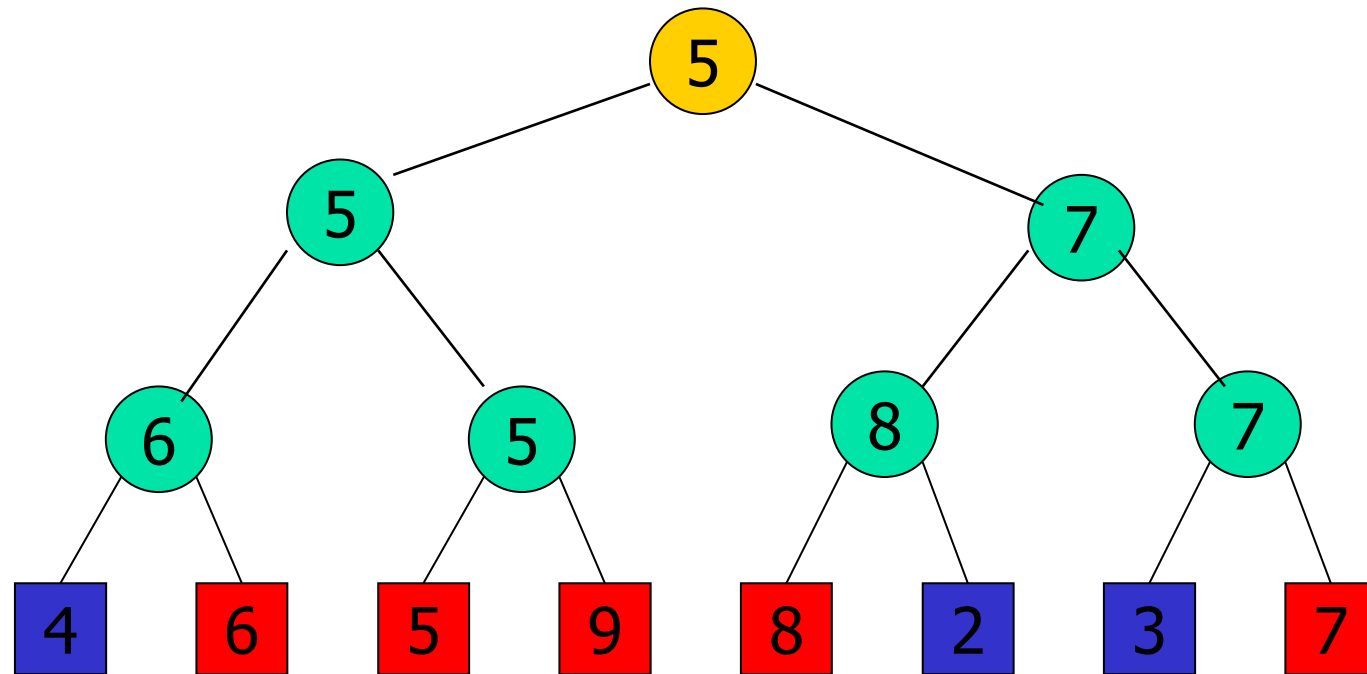
WT Applications – Sorting (12)



Sorted array



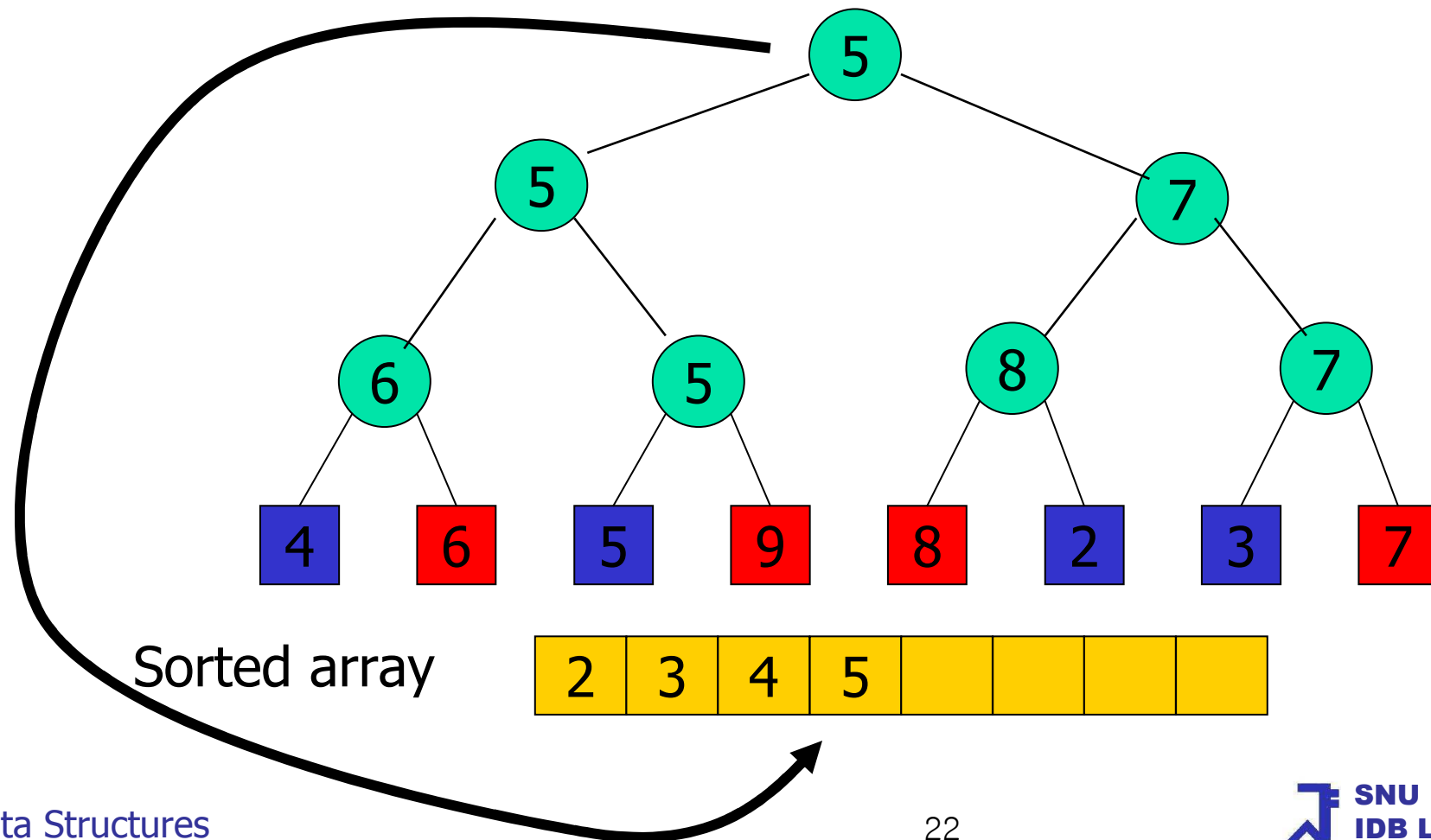
WT Applications – Sorting (13)



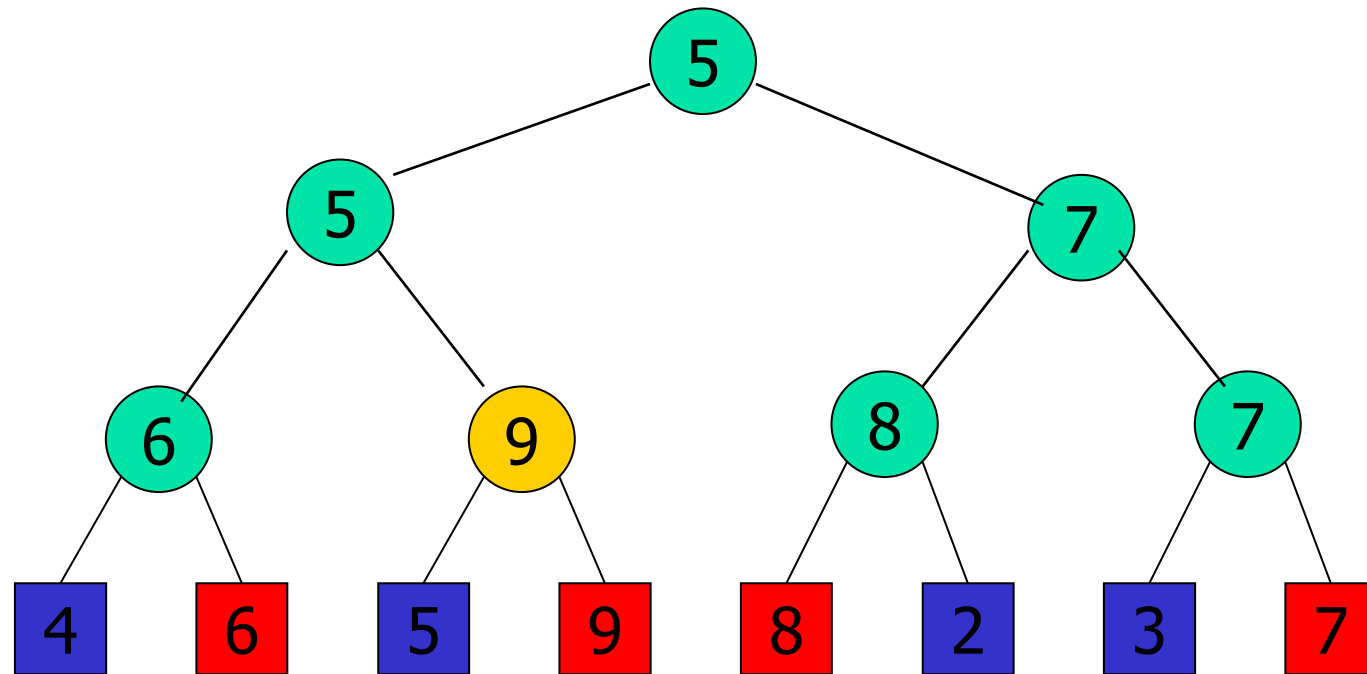
Sorted array



WT Applications – Sorting (14)



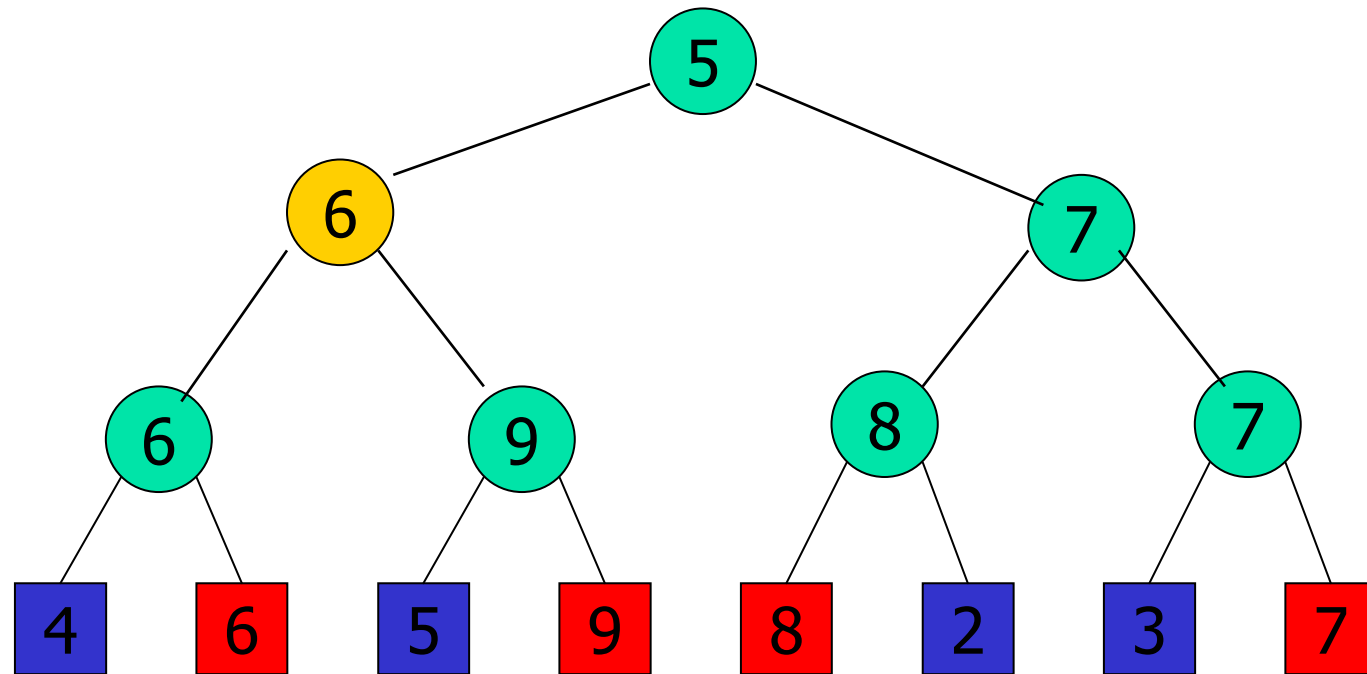
WT Applications – Sorting (15)



Sorted array



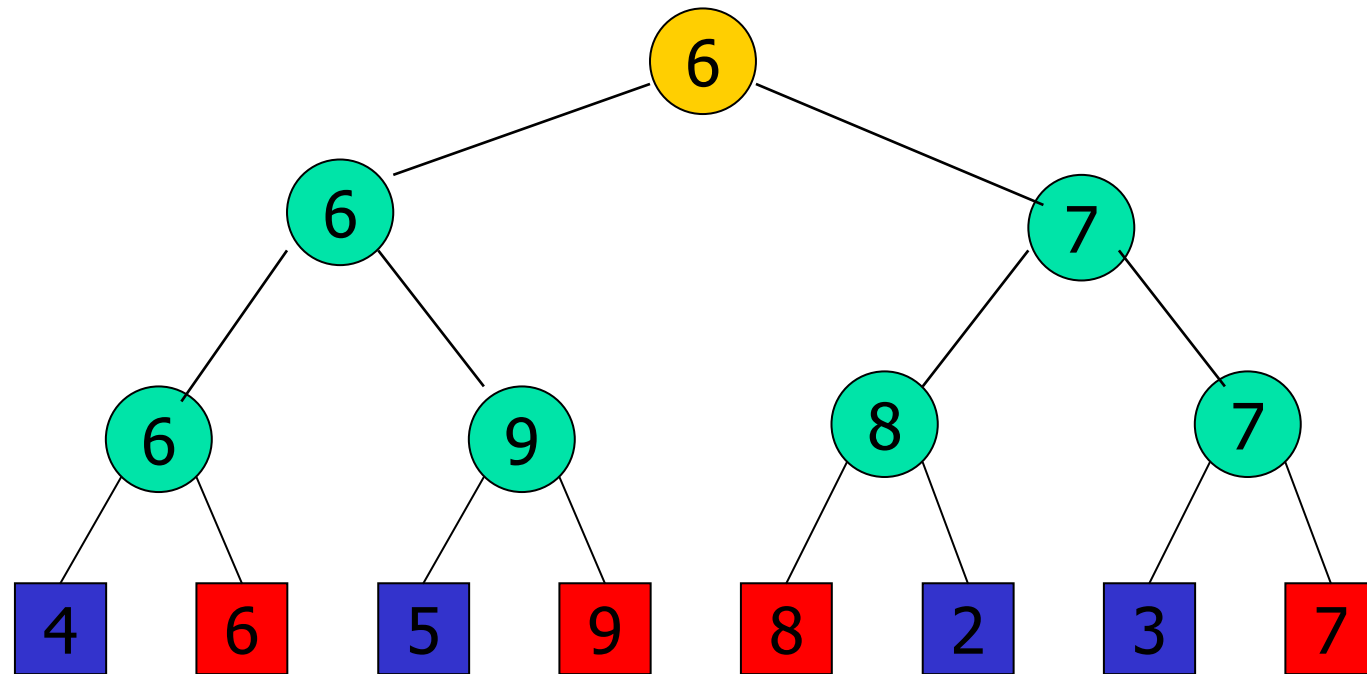
WT Applications – Sorting (16)



Sorted array



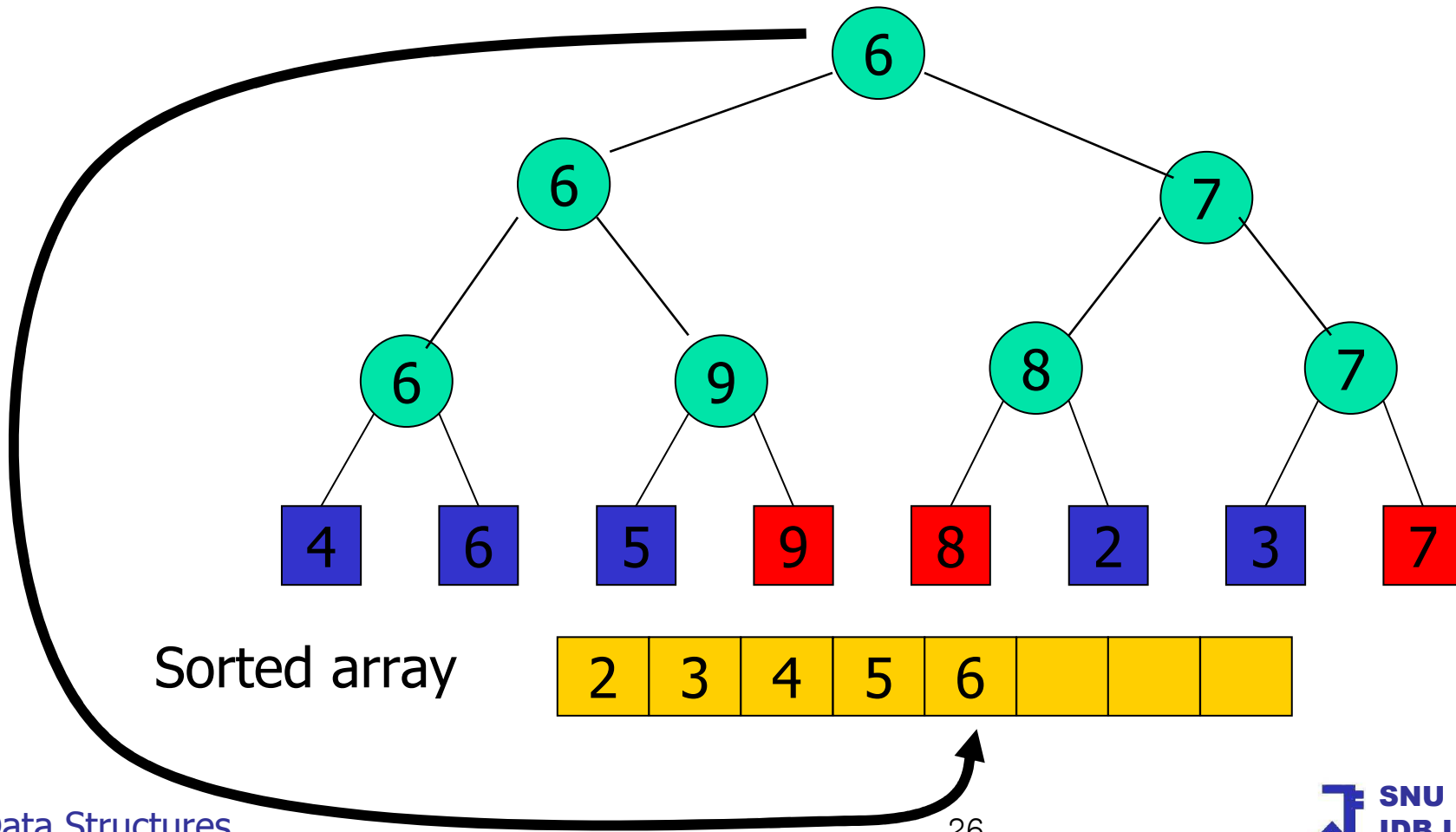
WT Applications – Sorting (17)



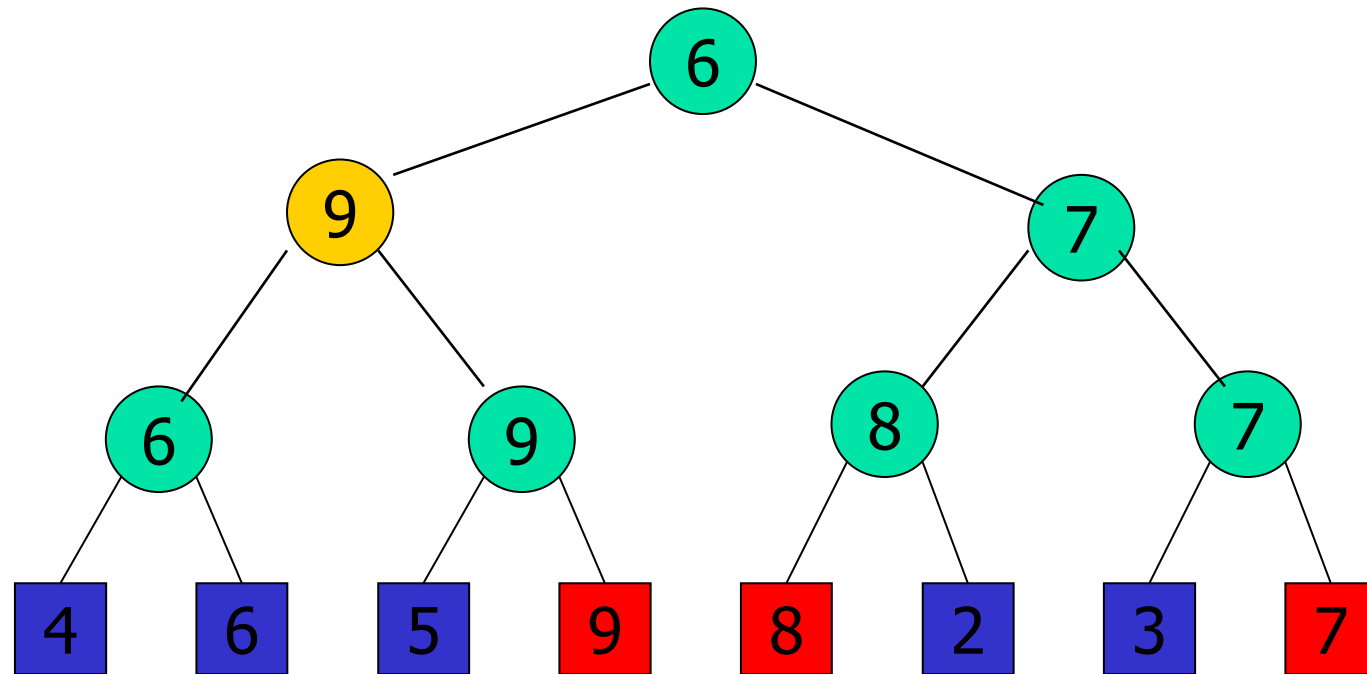
Sorted array



WT Applications – Sorting (18)



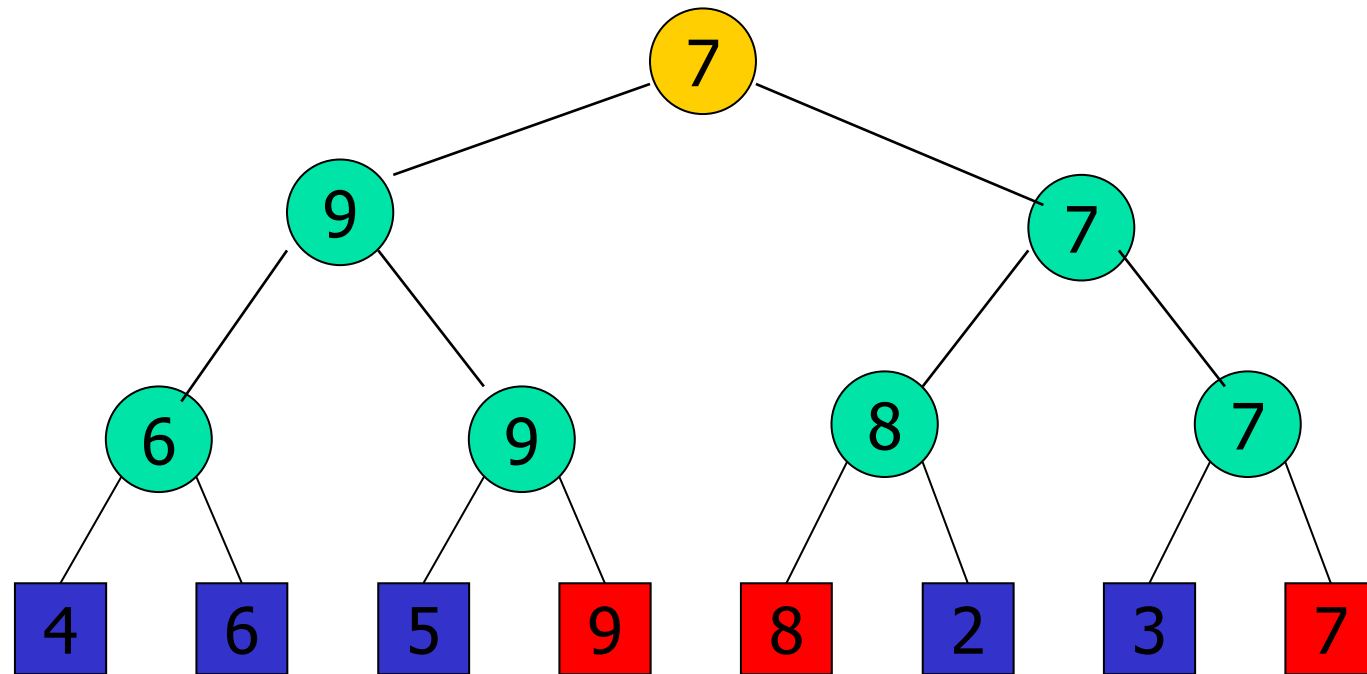
WT Applications – Sorting (19)



Sorted array



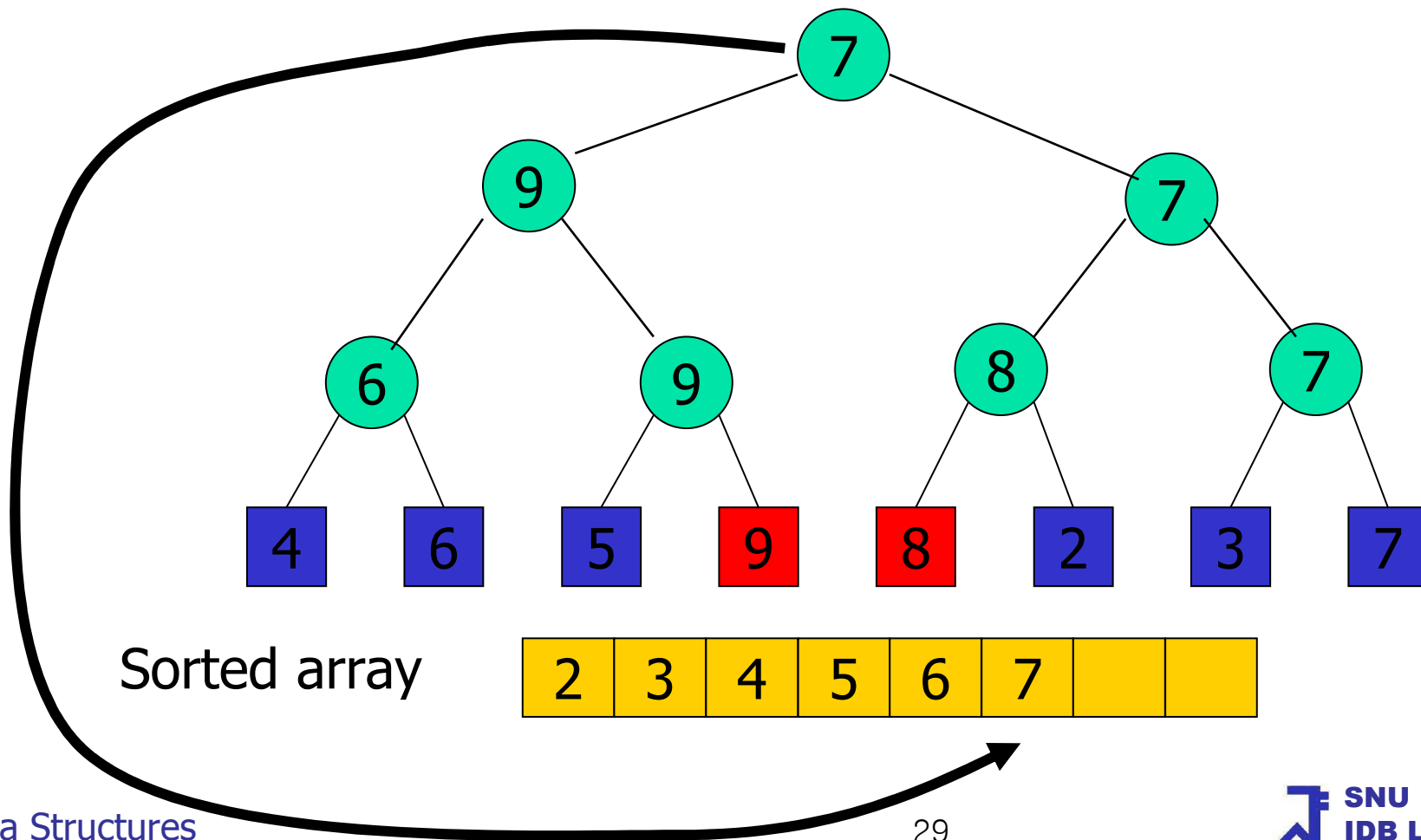
WT Applications – Sorting (20)



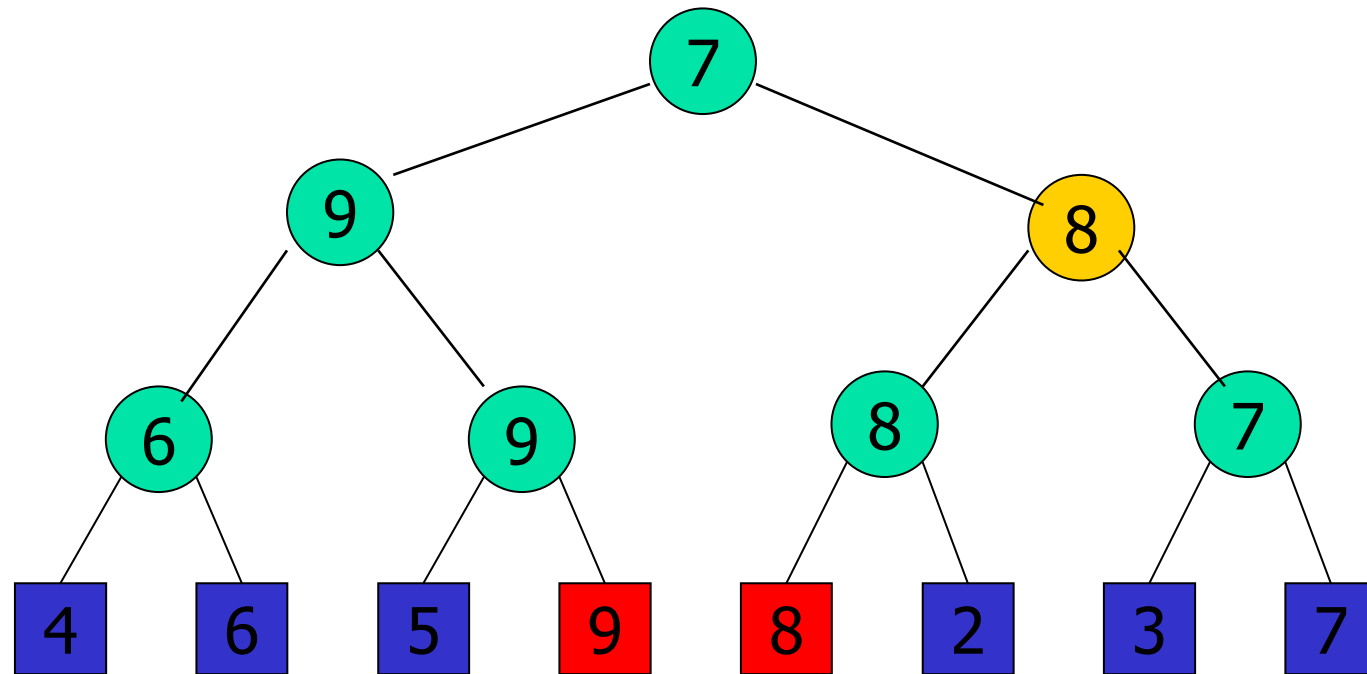
Sorted array



WT Applications – Sorting (21)



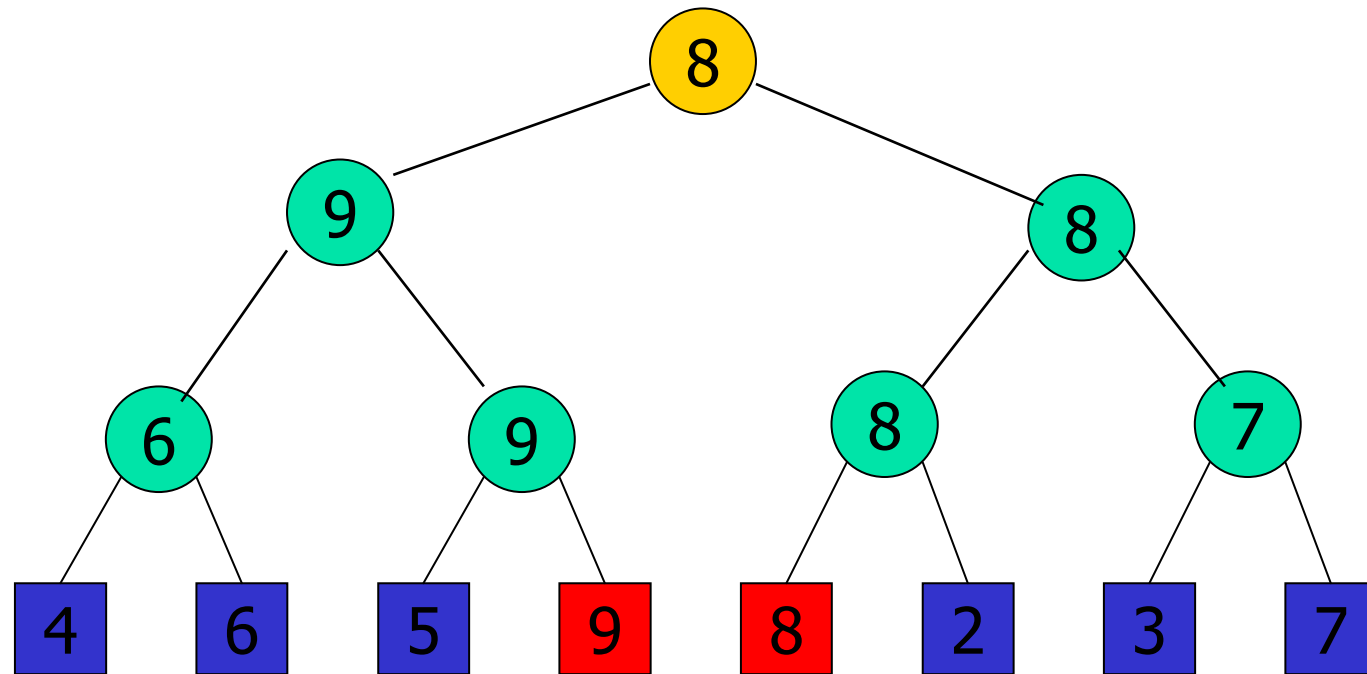
WT Applications – Sorting (22)



Sorted array



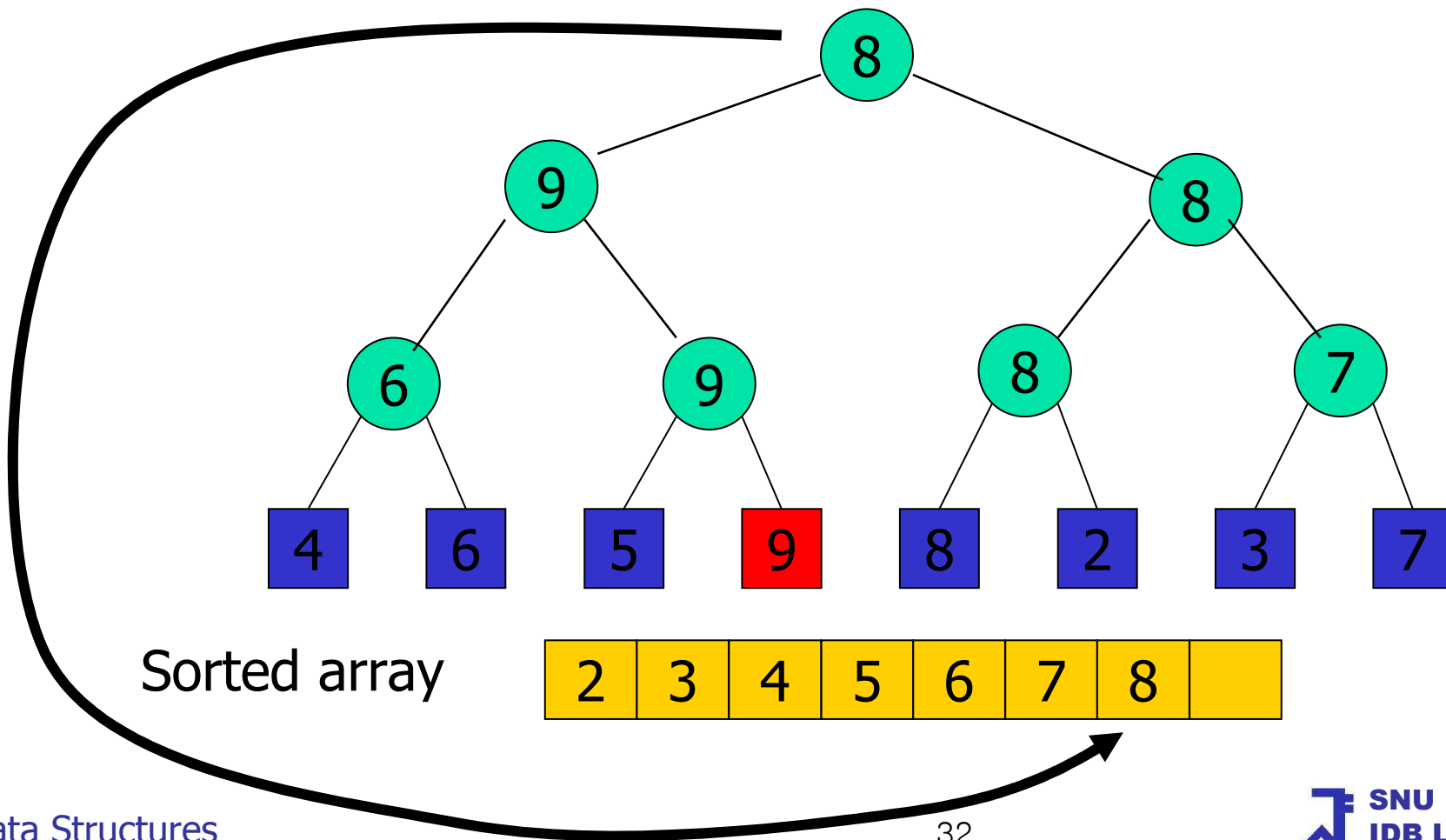
WT Applications – Sorting (23)



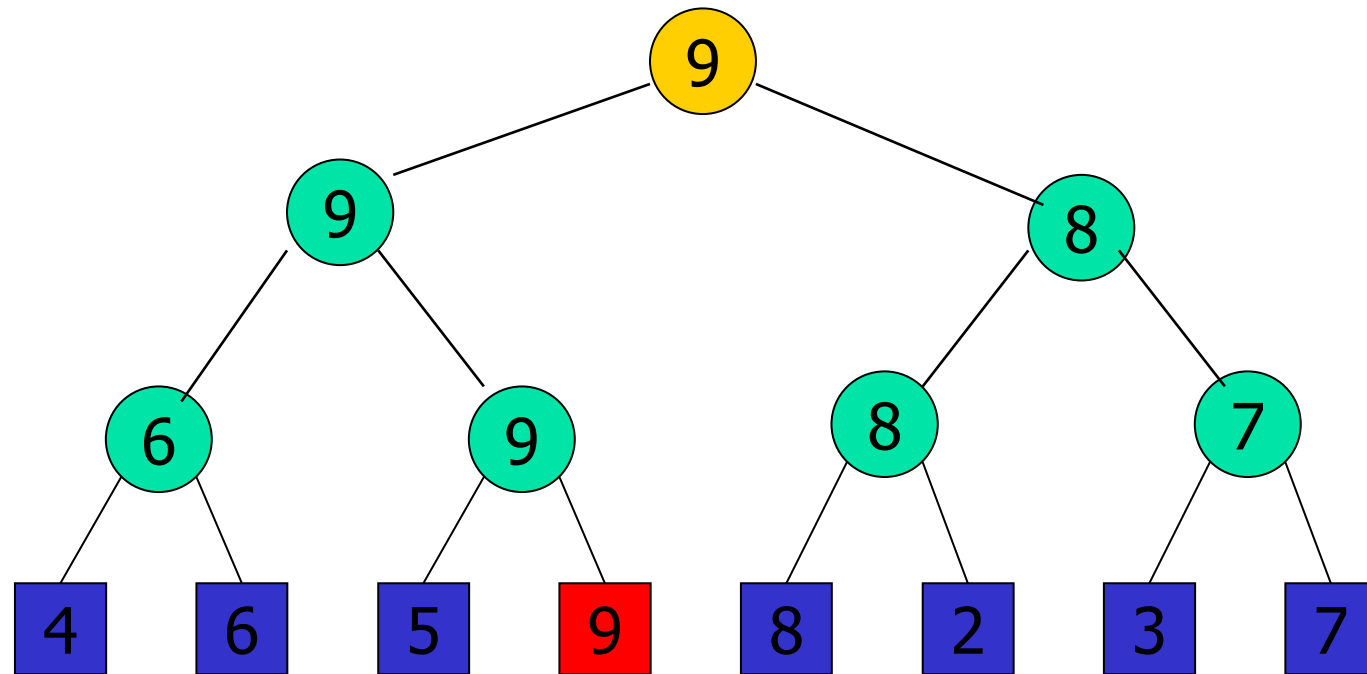
Sorted array



WT Applications – Sorting (24)



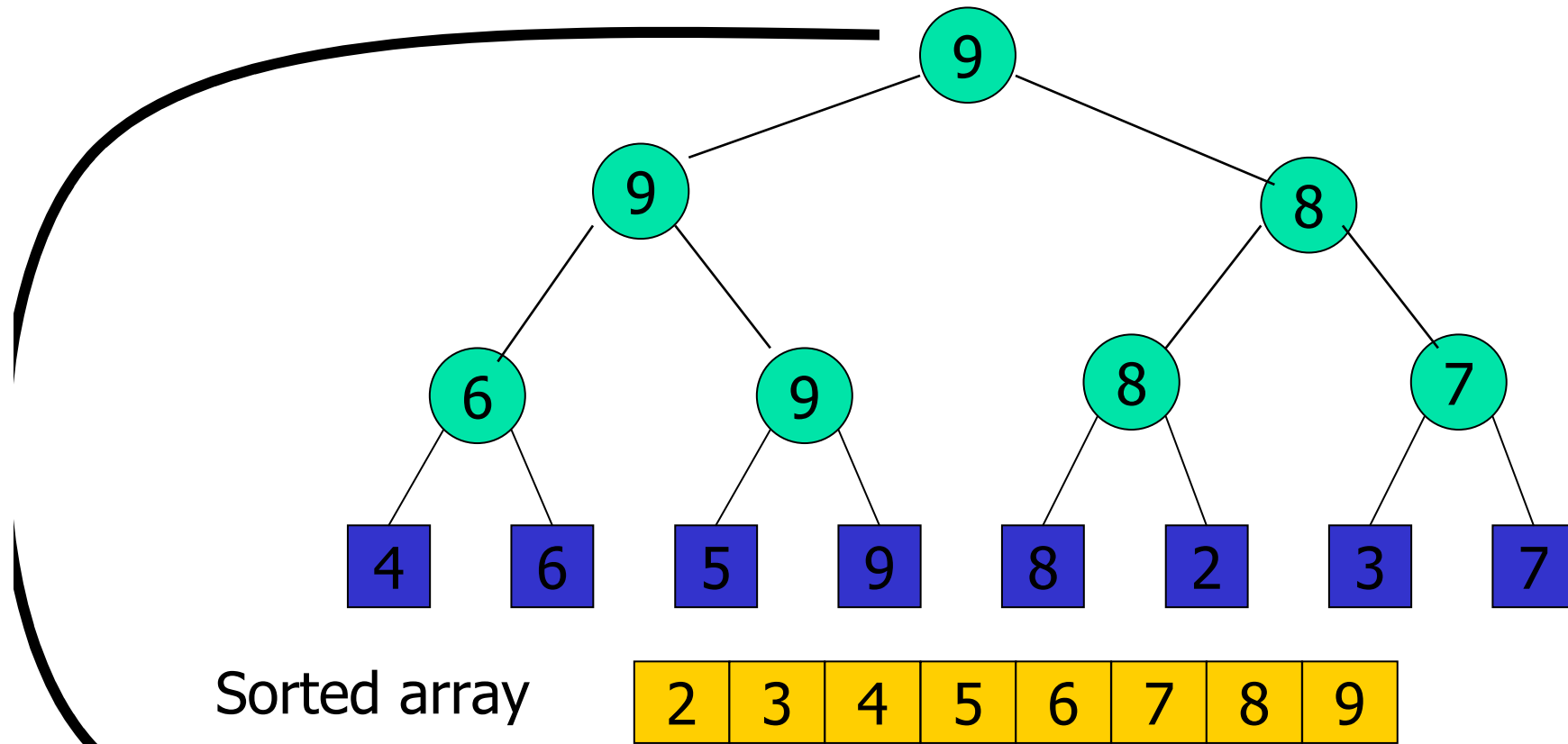
WT Applications – Sorting (25)



Sorted array



WT Applications – Sorting (26)





Complexity of Winner-Tree Sorting

- Initialize winner tree: $O(n)$ time
- Remove winner and replay: $O(\log n)$ time
- Remove winner and replay n times: $O(n \cdot \log n)$ time

- Total sort time is $O(n \cdot \log n)$
 - Actually $\Theta(n \cdot \log n)$



The ADT WinnerTree

AbstractDataType WinnerTree {

instances

complete binary trees with each node pointing to the winner of the match played there: the external nodes represent the players

operations

`initialize(a)` : initialize a winner tree for the players in array a

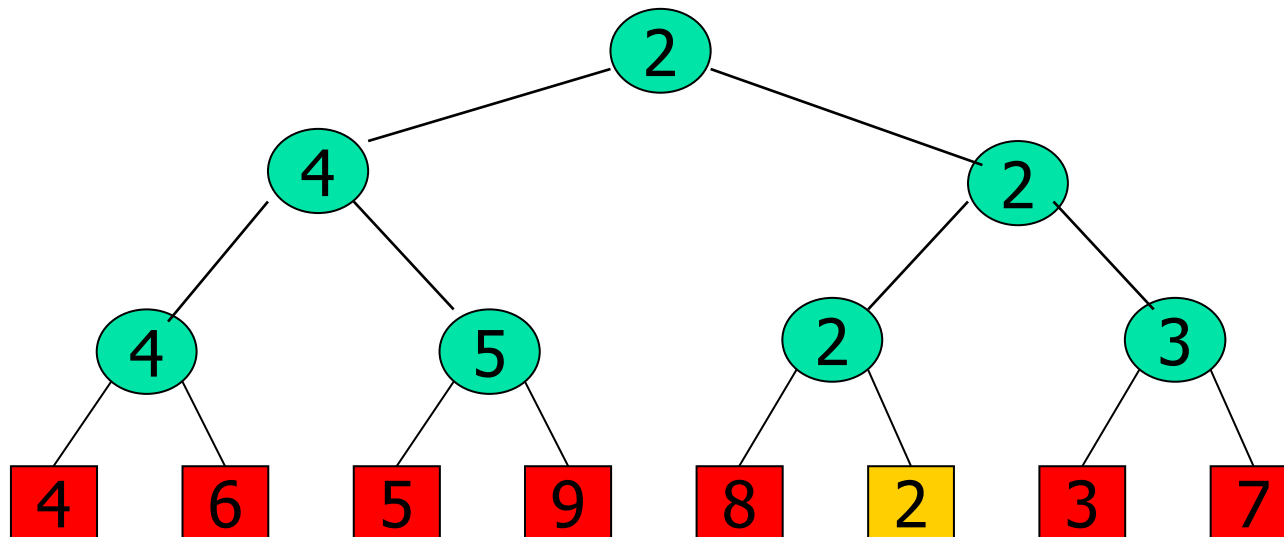
`getWinner()` : return the tournament winner

`rePlay(i)` : replay matches following a change in player i

}

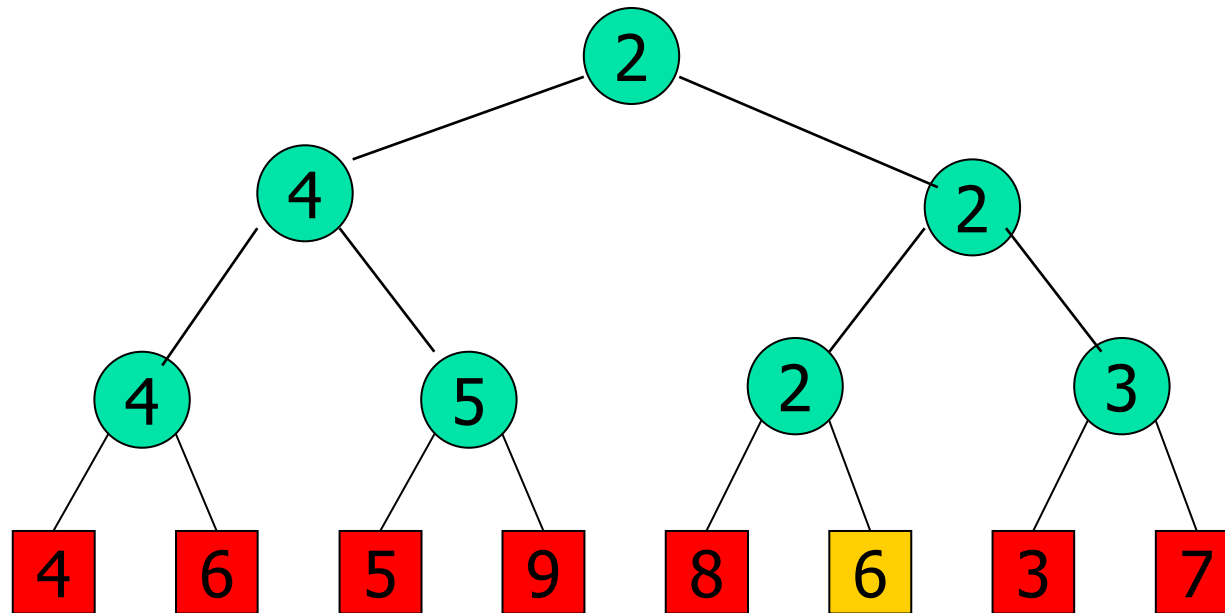
Replace Winner and Replay (1)

- Changing the the value of the winner requires a replay of all matches on the path from the winner's external node to the root
- Tree Height
- $O(\log n)$ time \rightarrow more precisely $\Theta(\log n)$



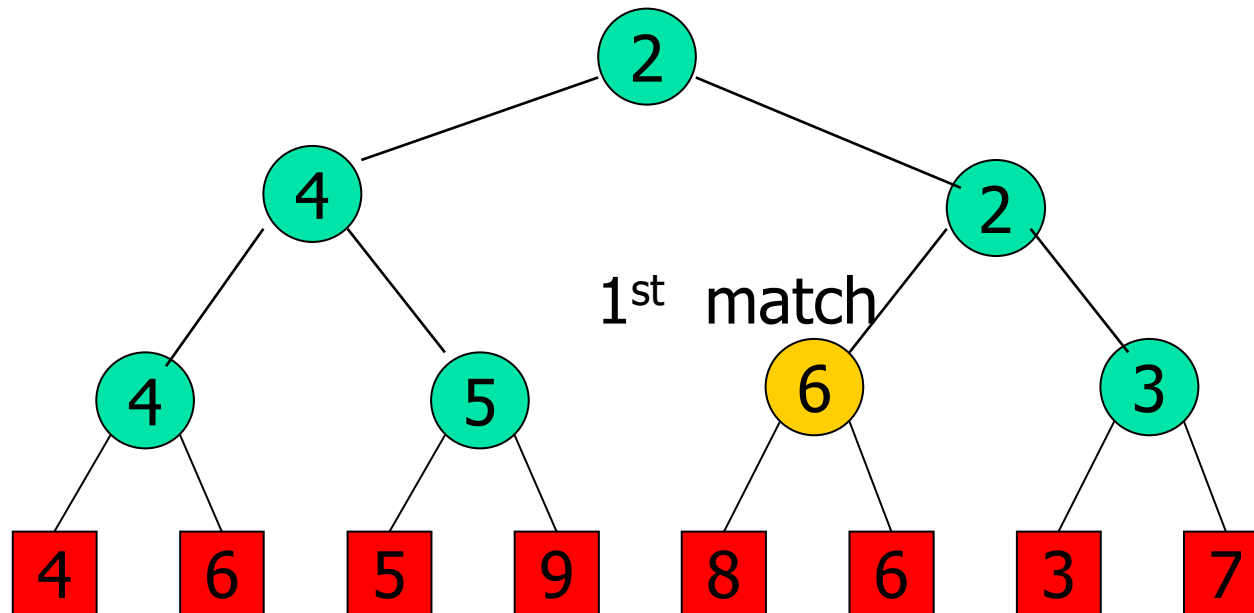
- Suppose replace winner “2” with the new value “6”

Replace Winner and Replay (2)



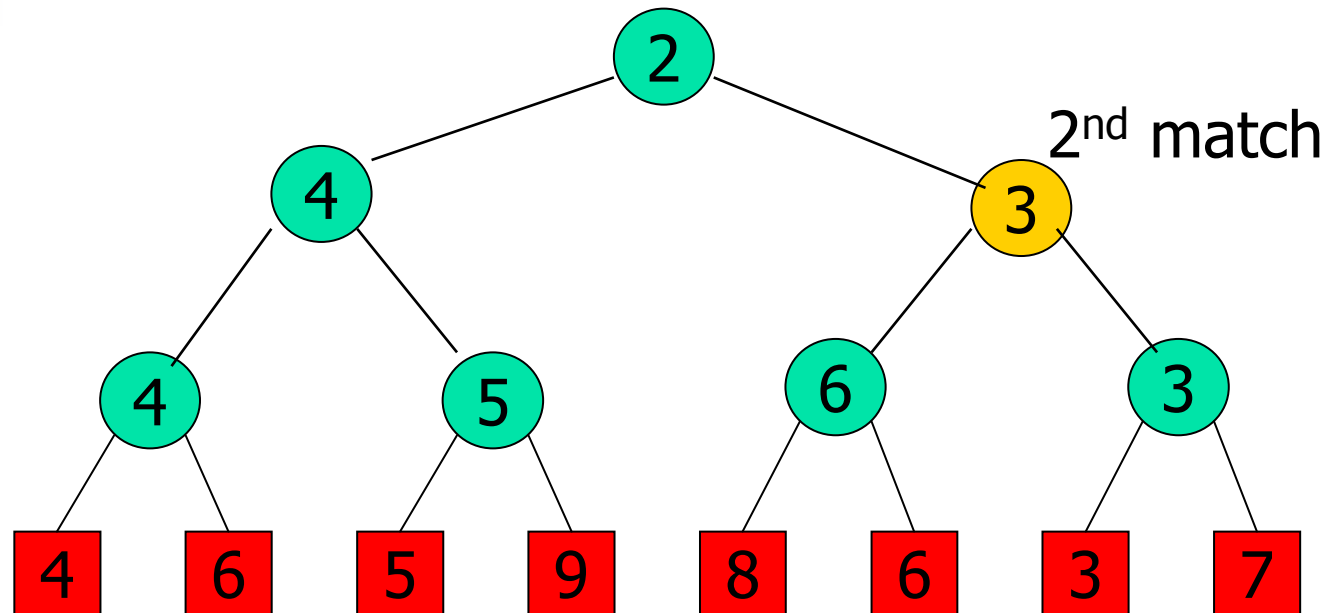
rePlay(6) : start rematch on player 6 whose value is now "6"

Replace Winner and Replay (3)



Player 6 is $a[12]$ in the array: the first match result between $a[11]$ and $a[12]$ is stored in $a[5]$

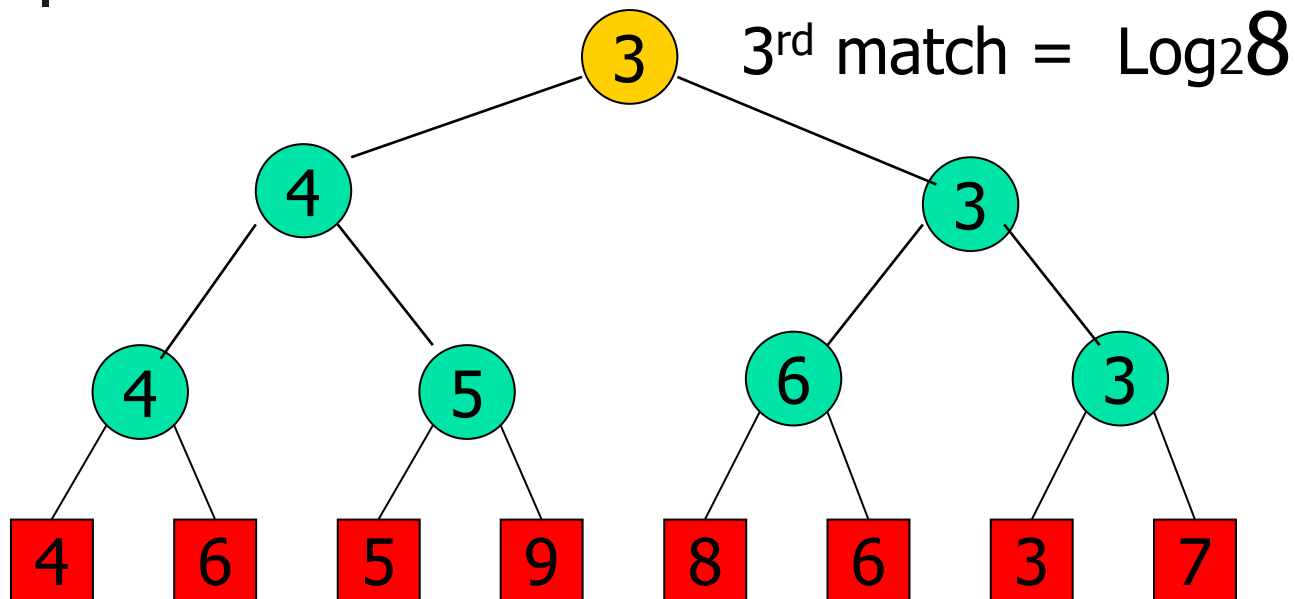
Replace Winner and Replay (4)



Change in $a[5]$ causes a rematch between $a[5]$ and $a[6]$.

The match result is stored in $a[2]$.

Replace Winner and Replay (5)



Change in $a[2]$ causes a rematch between $a[1]$ and $a[2]$.
The match result is stored in $a[0]$.

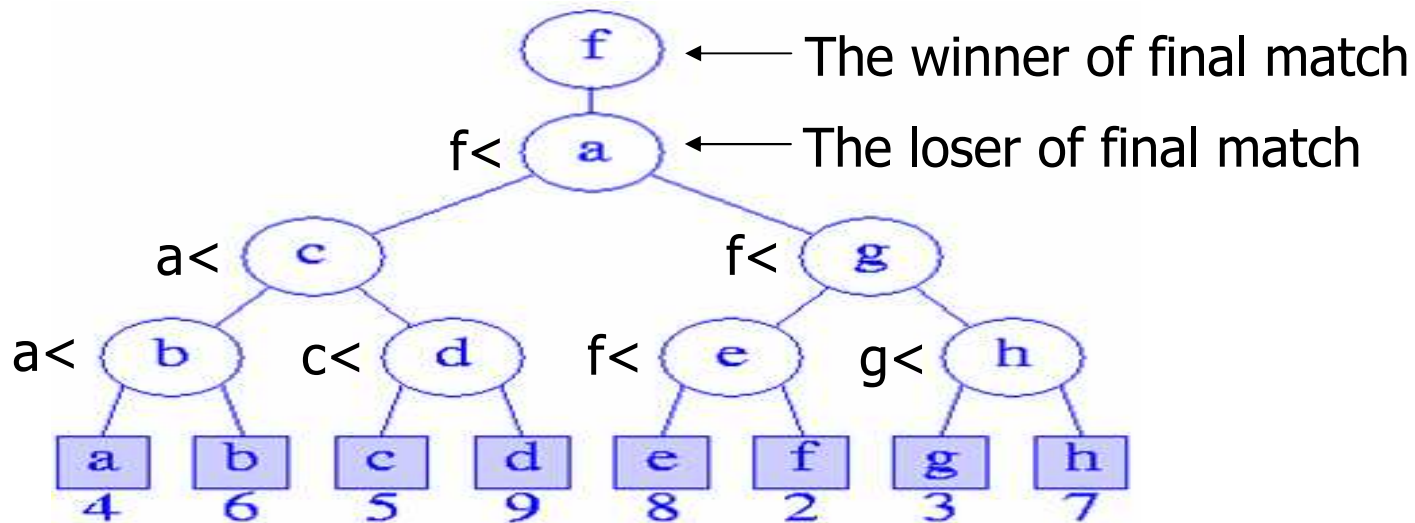


Table of Contents

- Winner Trees
- Loser Trees
- Tournament Tree Applications
 - Bin Packing Using First Fit (BPFF)
 - Bin Packing Using Next Fit (BPNF)

Loser Trees

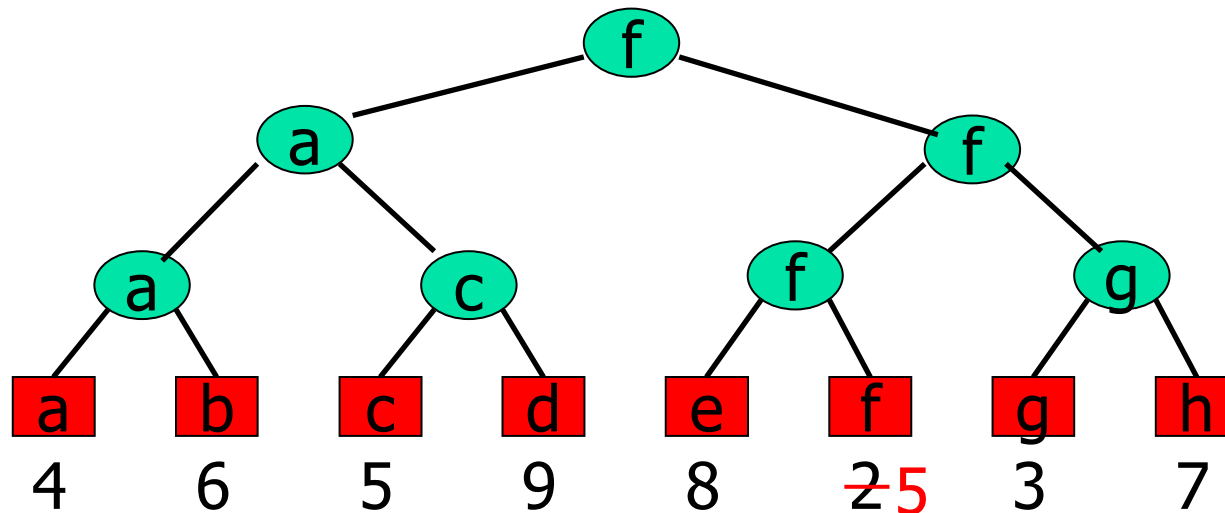
- Each match node stores the match loser rather than the match winner



Min Loser Tree

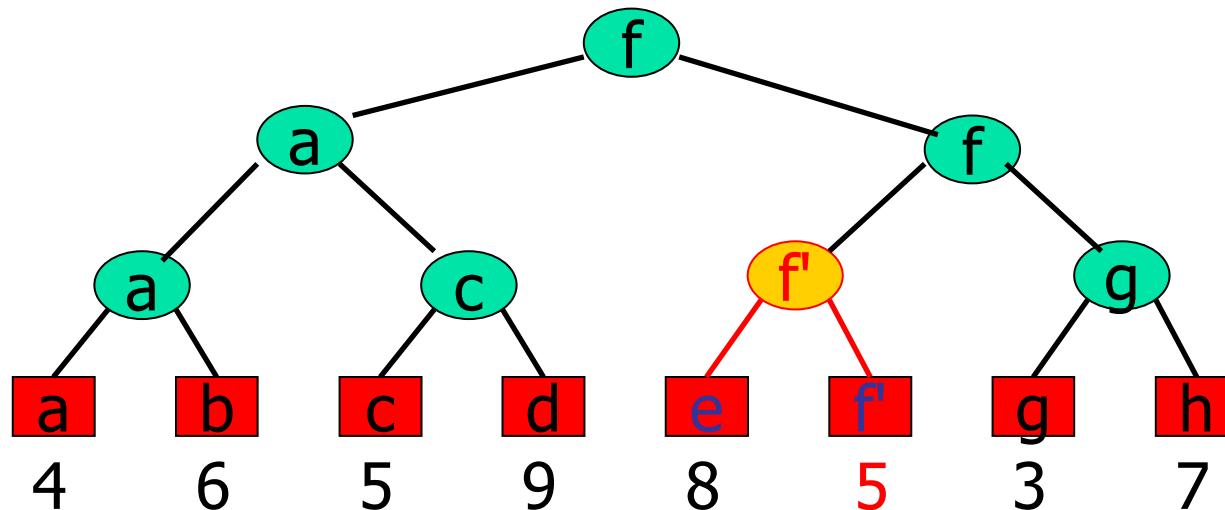
- Can reduce the work when the winner value is changed
 - Show the better performance than the winner tree

Replay in 8-Player Min Winner Tree (1)



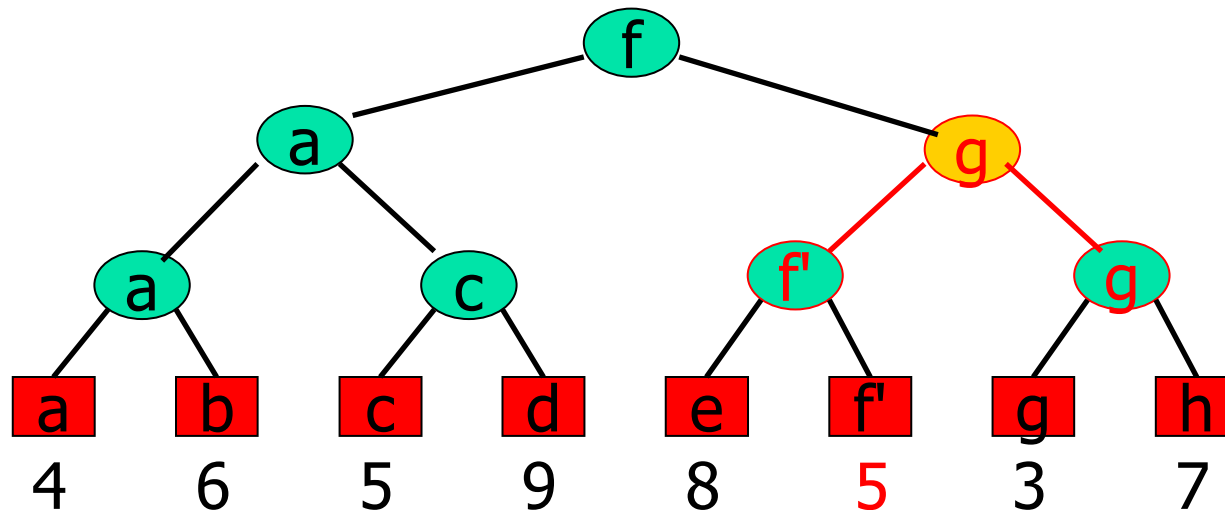
- Suppose we change f (key 2) with a new key 5

Replay in 8-Player Min Winner Tree (2)



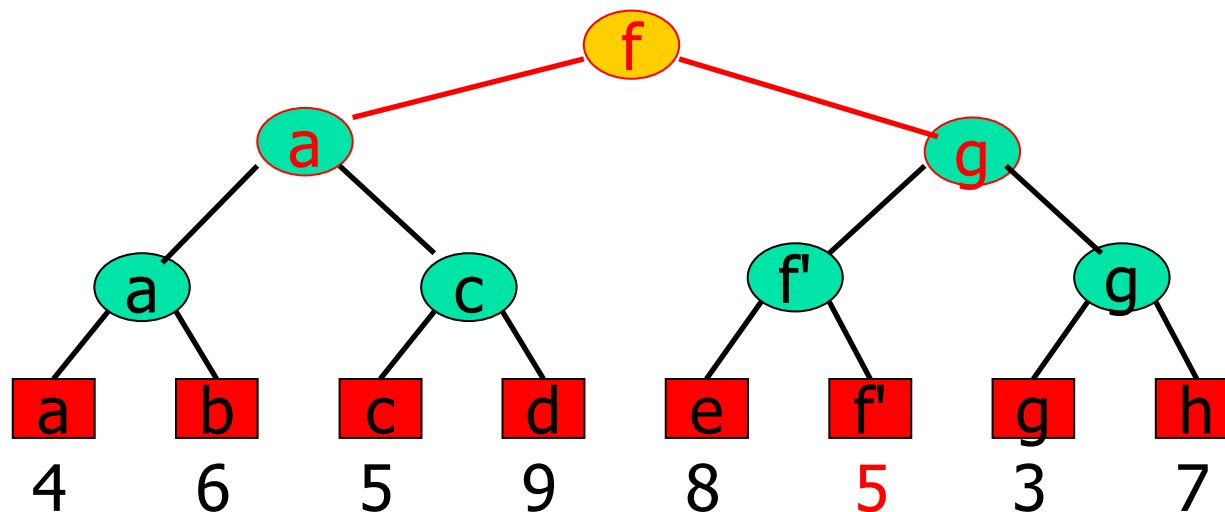
- We should compare f' with e (= **another child** of **parent** of f')
- Need referencing twice (self \rightarrow parent \rightarrow sibling)

Replay in 8-Player Min Winner Tree (3)



- We should compare f' with g (= **another child** of **parent** of f')
- Need referencing twice (self \rightarrow parent \rightarrow sibling)

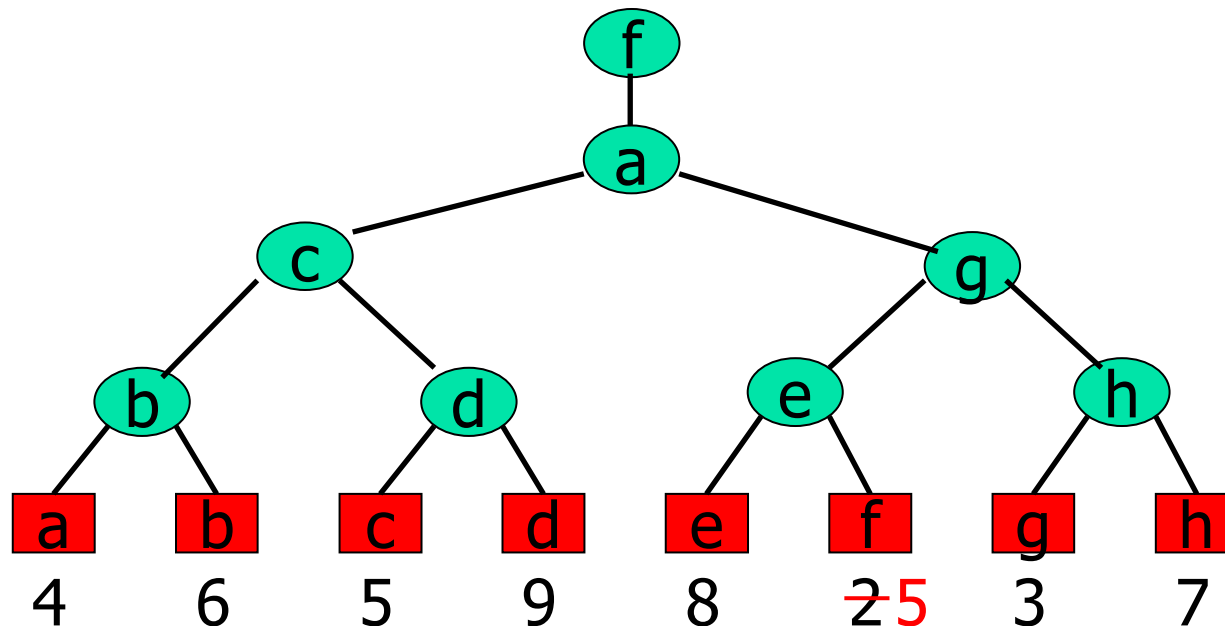
Replay in 8-Player Min Winner Tree (4)



- We should compare g with a (= **another child** of **parent** of g)
- Need referencing twice (self → parent → sibling)

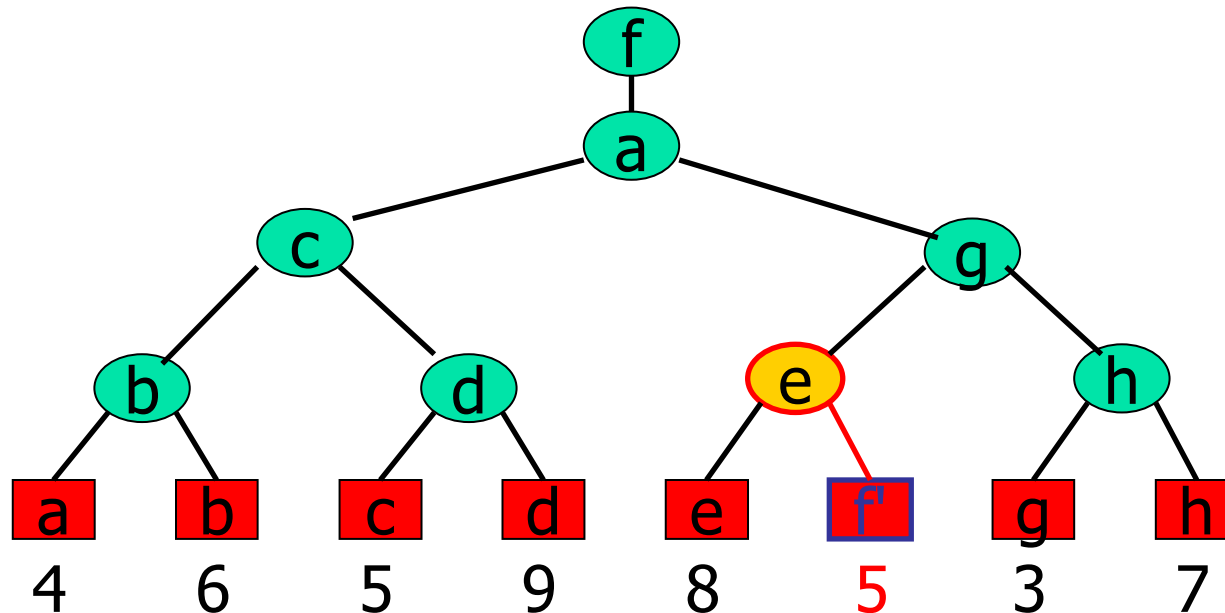
Replay in 8-Player Min Loser Tree (1)

- Special case: The key of *winner* is changed



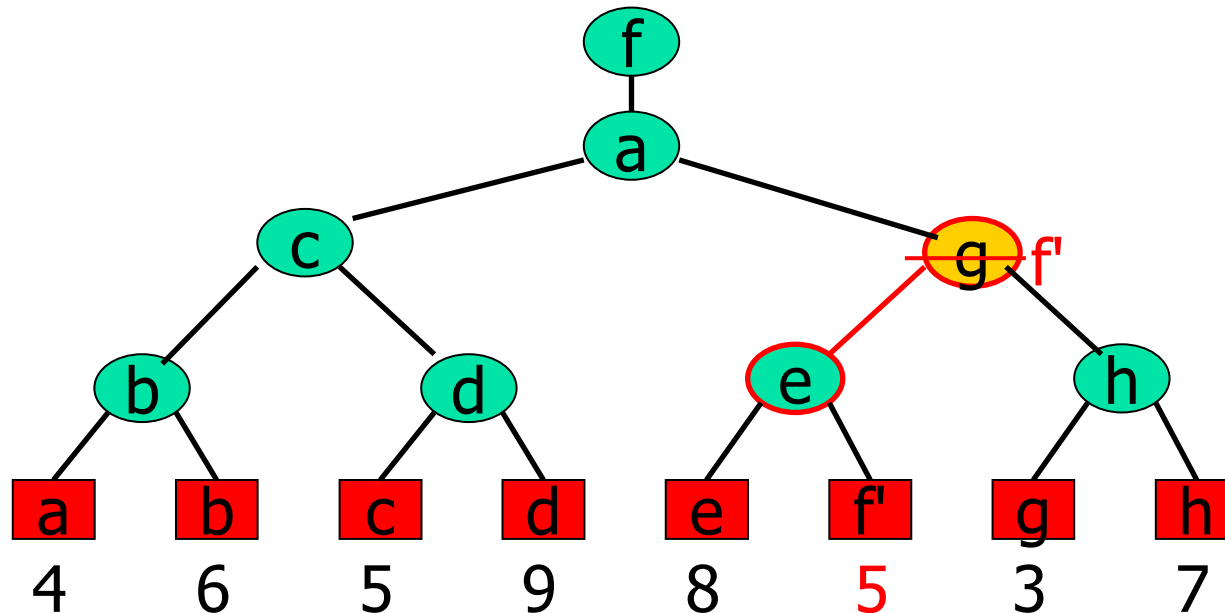
- Suppose we change winner *f* (key 2) with a new key 5

Replay in 8-Player Min Loser Tree (2)



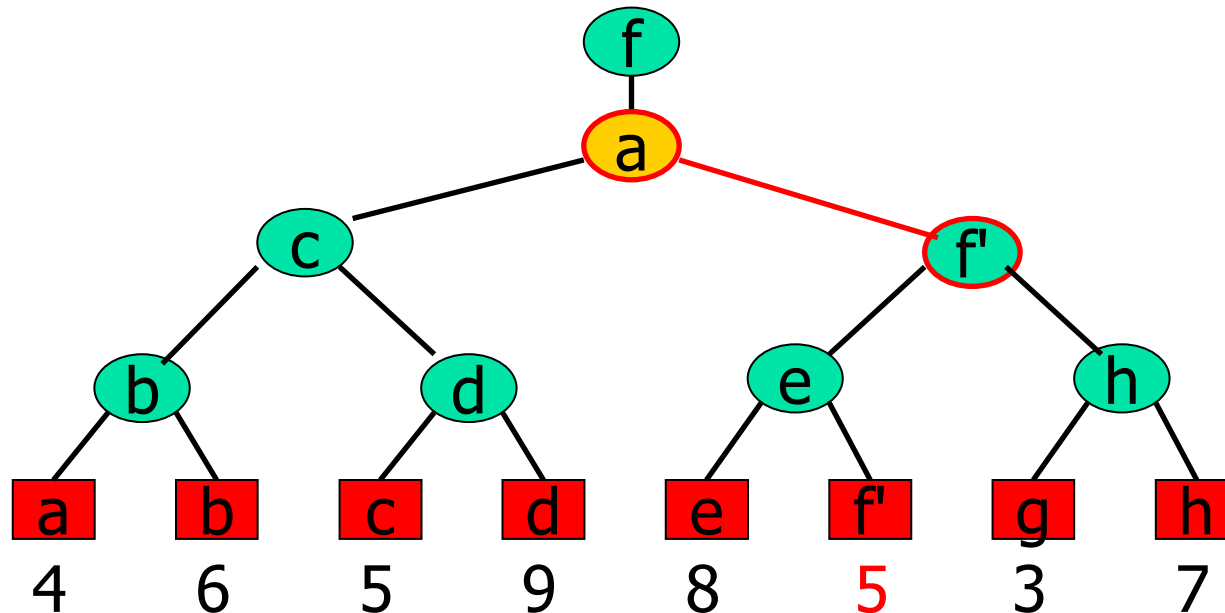
- We simply compare f' with its **parent** because previous loser is stored at parent node
- Need referencing only once (self \rightarrow parent)

Replay in 8-Player Min Loser Tree (3)



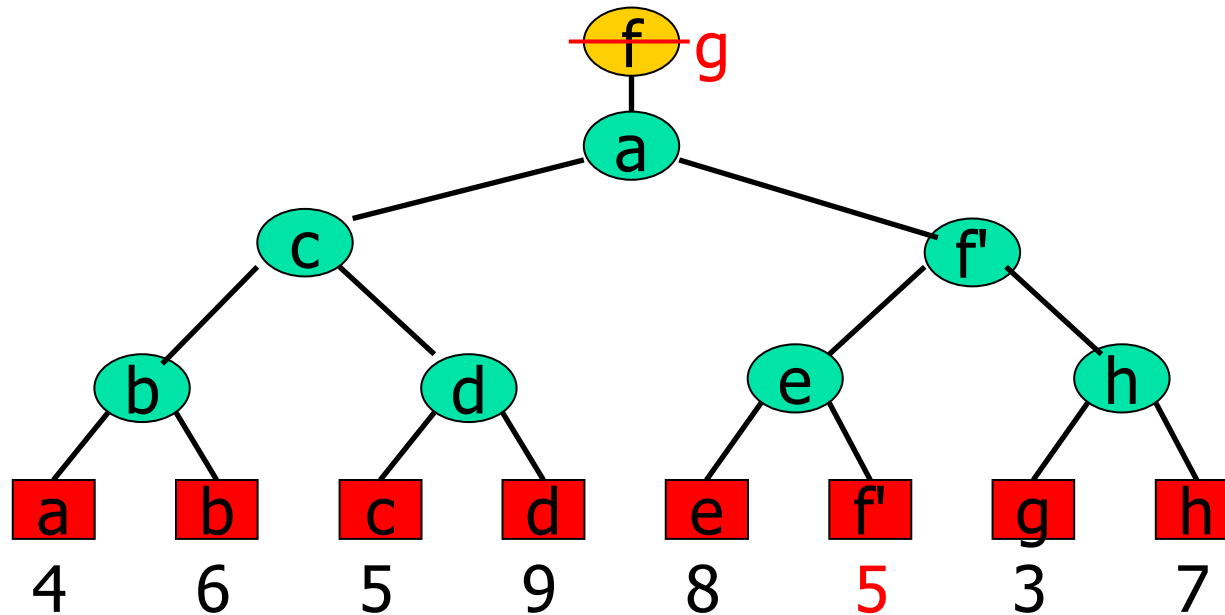
- We simply compare f' with its **parent** because previous loser is stored at parent node
- Need referencing only once (self \rightarrow parent)

Replay in 8-Player Min Loser Tree (4)



- We simply compare g with its **parent** because previous loser is stored at parent node
- Need referencing only once (self → parent)

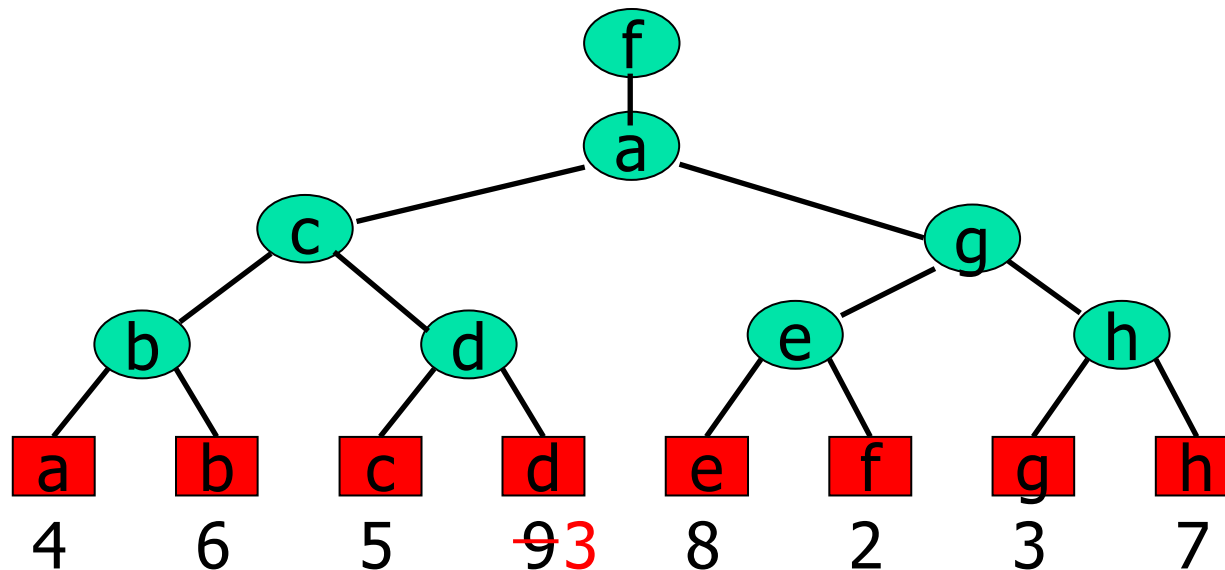
Replay in 8-Player Min Loser Tree (5)



- Winner changes to g

Replay in 8-Player Min Loser Tree (6)

- The loser tree is not effective if the changed node is not the previous *winner*



- Suppose we change d (key 9) with a new key 3
- We should compare d with its sibling(c), NOT parent(d)



Table of Contents

- Winner Trees
- Loser Trees
- Tournament Tree Applications
 - Bin Packing Using First Fit (BPFF)
 - Bin Packing Using Next Fit (BPNF)

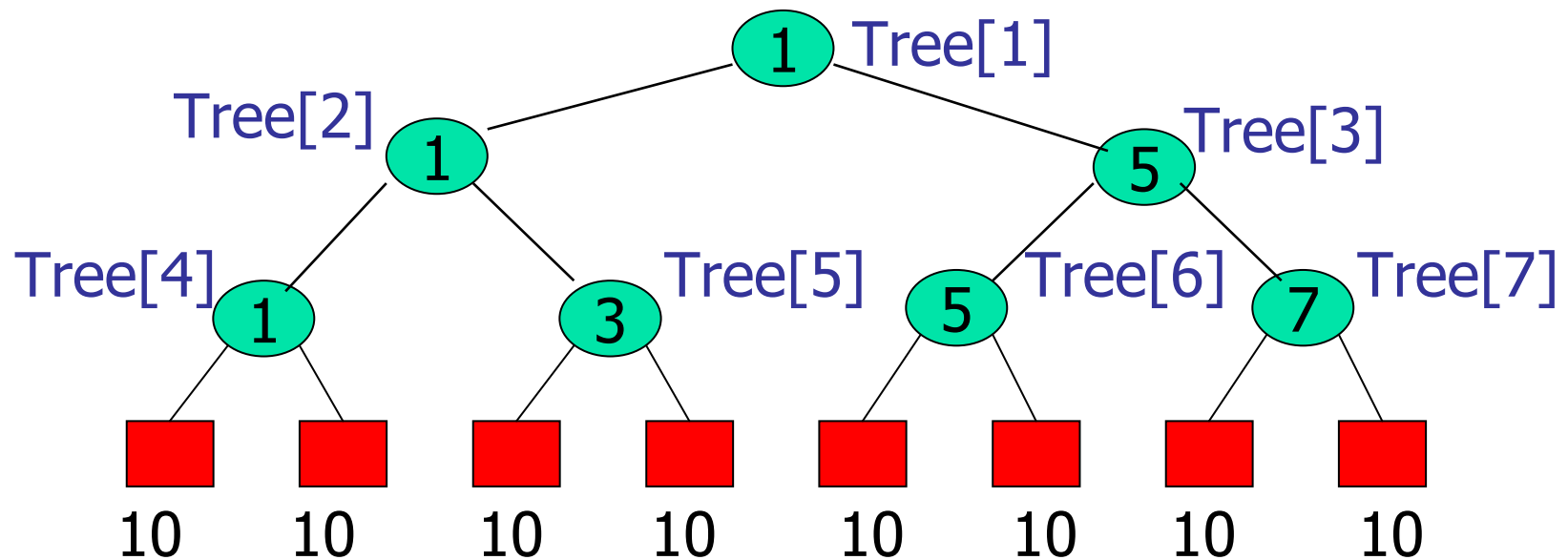


Bin Packing Problem

- Object i requires $\text{objectSize}[i]$ units of capacity
 - $0 < \text{objectSize}[i] \leq \text{binCapacity}$
- A **feasible packing** is an assignment of objects to bins so that no bin's capacity is exceeded
 - Optimal packing: A feasible packing using **the fewest number of bins**
- First-Fit: find the first available bin using **the winner tree**
- Next-Fit: A variant of First-Fit searching the next bins

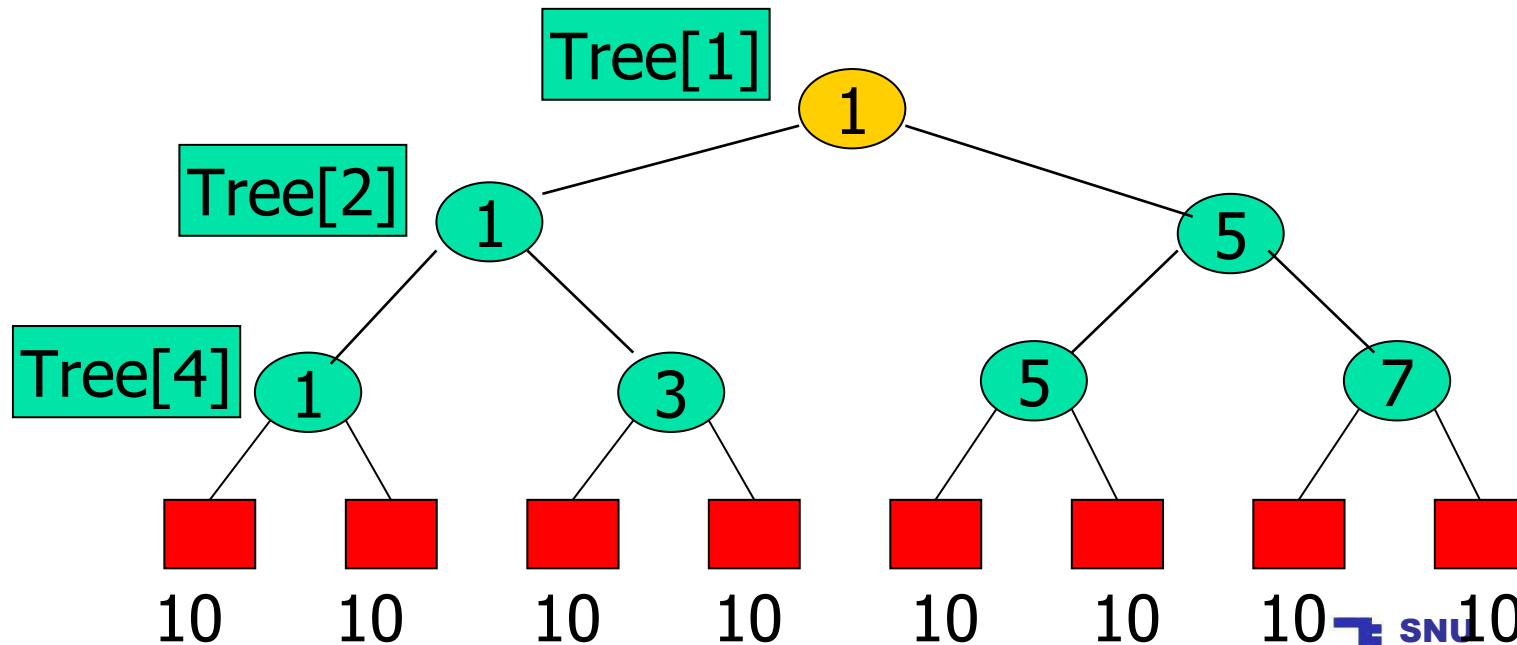
First-Fit and Winner Trees (1)

- Initialize: Winner tree of 8 bins and binCapacity = 10
 - If the players have the same value, the player with the small index is winner
- Suppose objects to be allocated are [8, 6, 5, 3]
- We want a feasible packing with a fewest number of bins



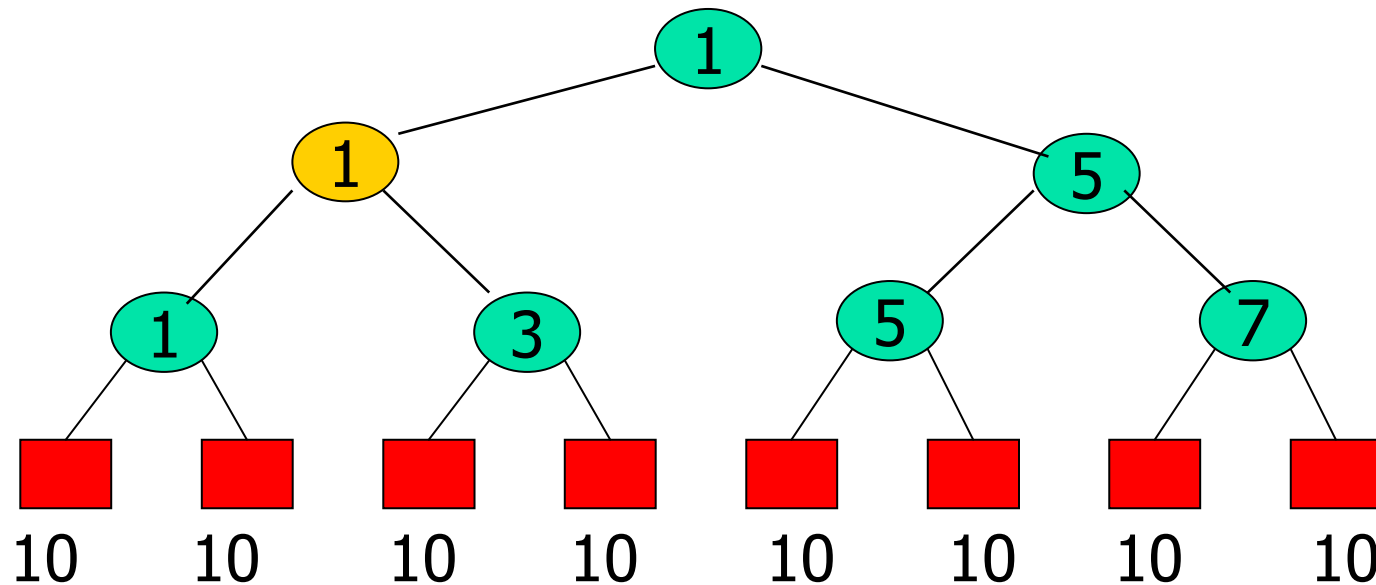
First-Fit and Winner Trees (2)

- Suppose objects to be allocated are [8, 6, 5, 3]
 - objectSize[1] is 8
- Bin[tree[1]].unusedCapacity \geq objectSize[1] \rightarrow go to left
 - $10 > 8$



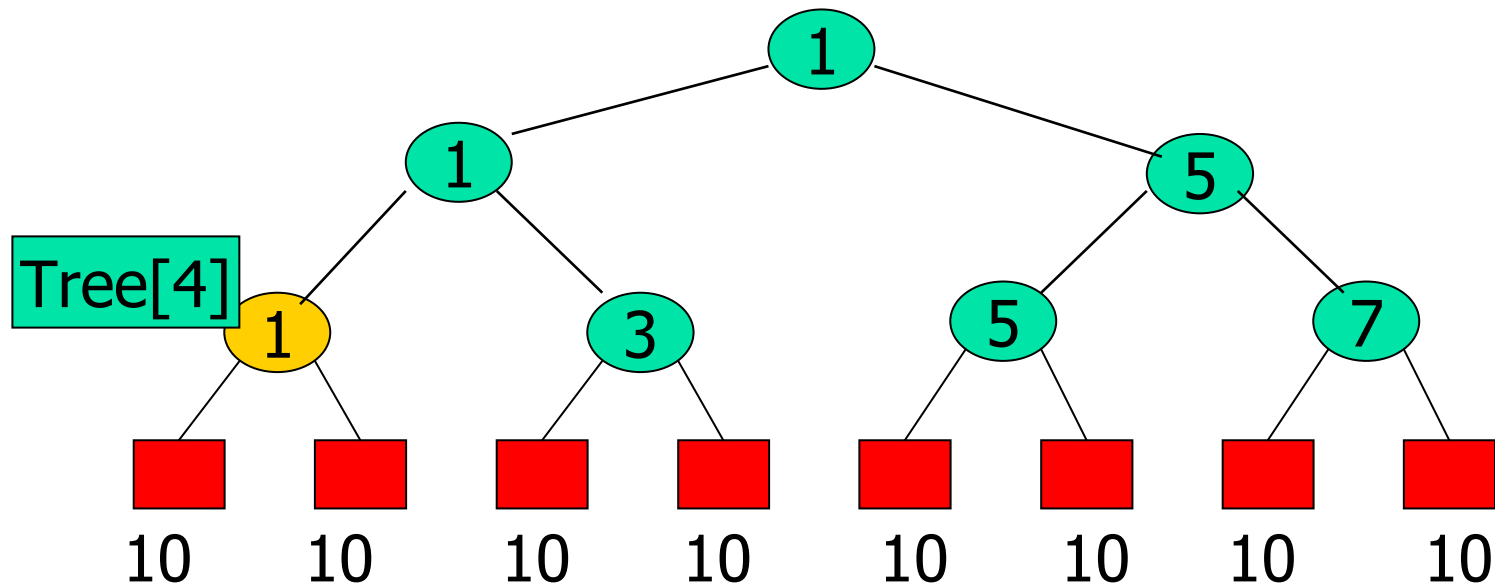
First-Fit and Winner Trees (3)

- $\text{Bin}[\text{tree}[2]].\text{unusedCapacity} \geq \text{objectSize}[1] \rightarrow \text{go to left}$
 - $10 > 8$



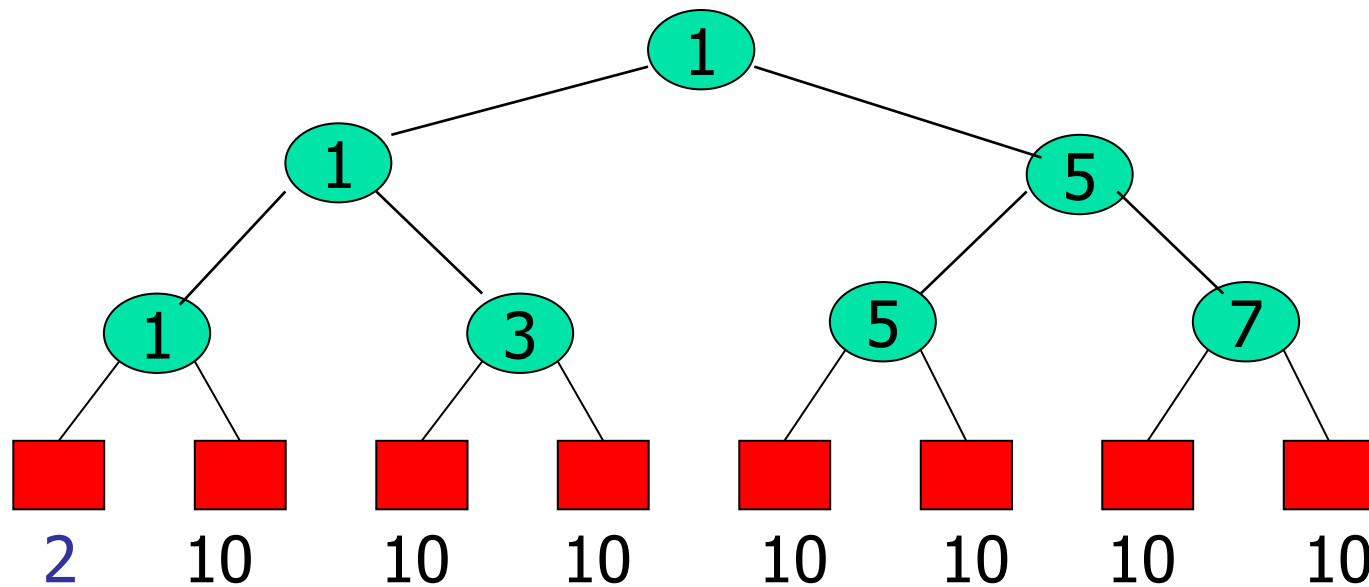
First-Fit and Winner Trees (4)

- $\text{Bin}[\text{tree}[4]].\text{unusedCapacity} \geq \text{objectSize}[1] \rightarrow \text{go to left}$
 - $10 > 8$



First-Fit and Winner Trees (5)

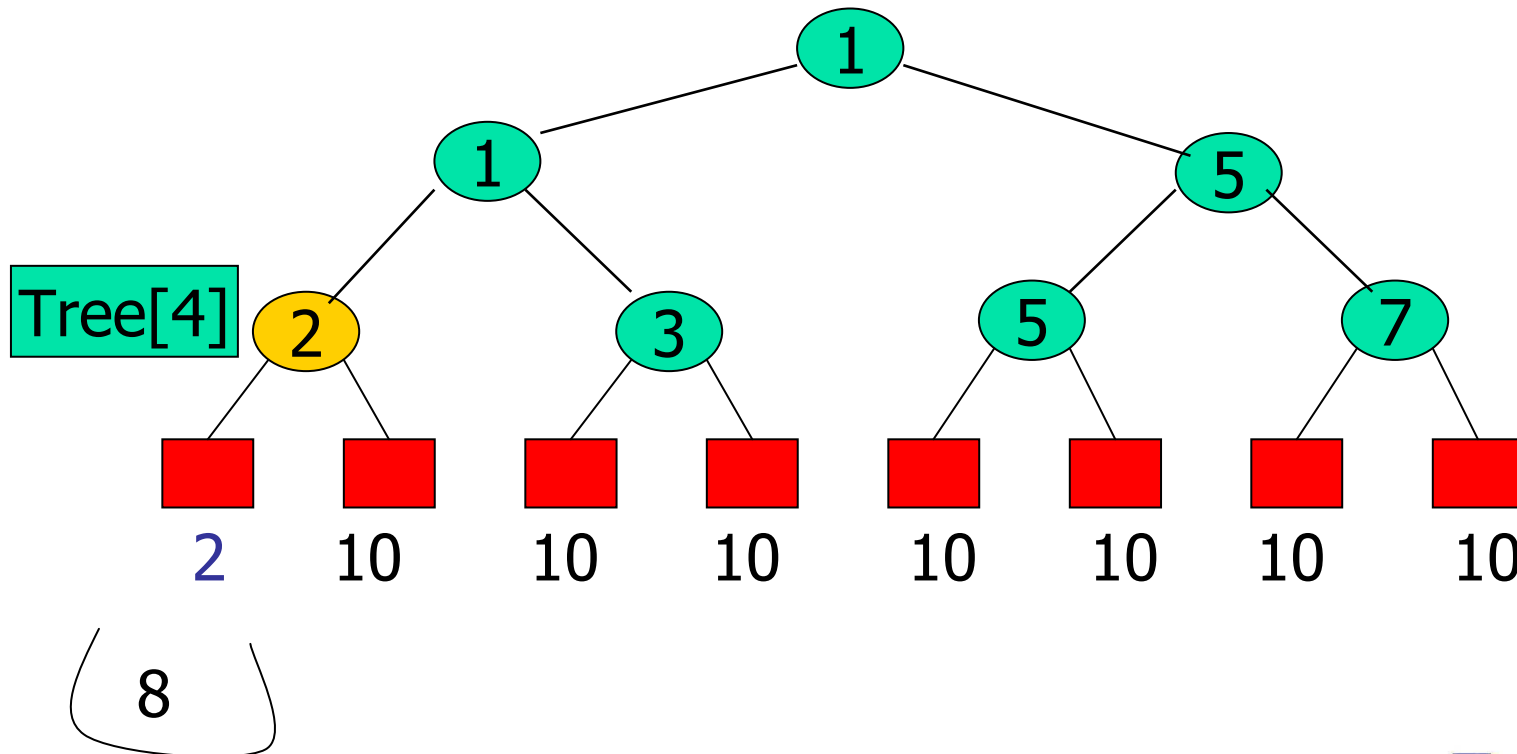
- Object with size "8" is now in Bin[1]
- Need to update the winner tree



8

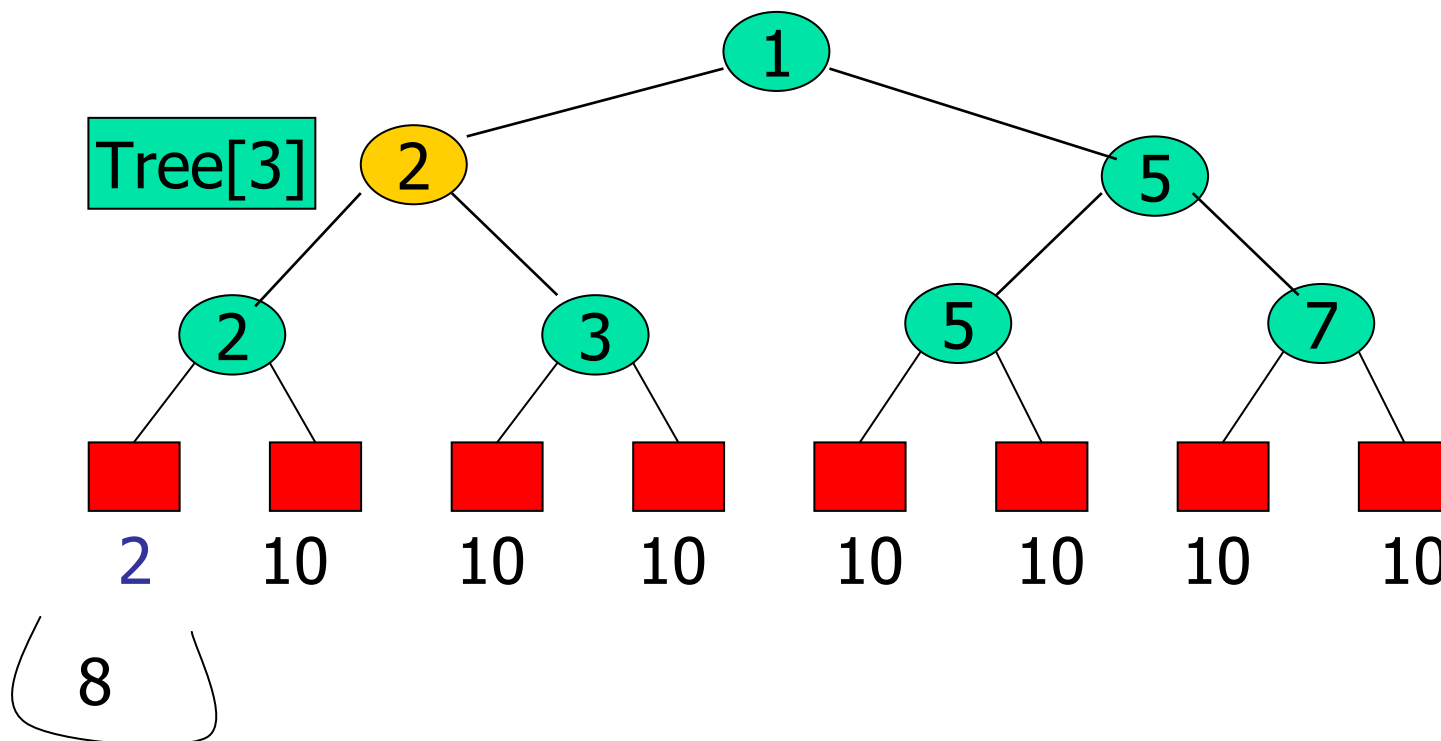
First-Fit and Winner Trees (6)

- Replay: start rematch at Tree[4]



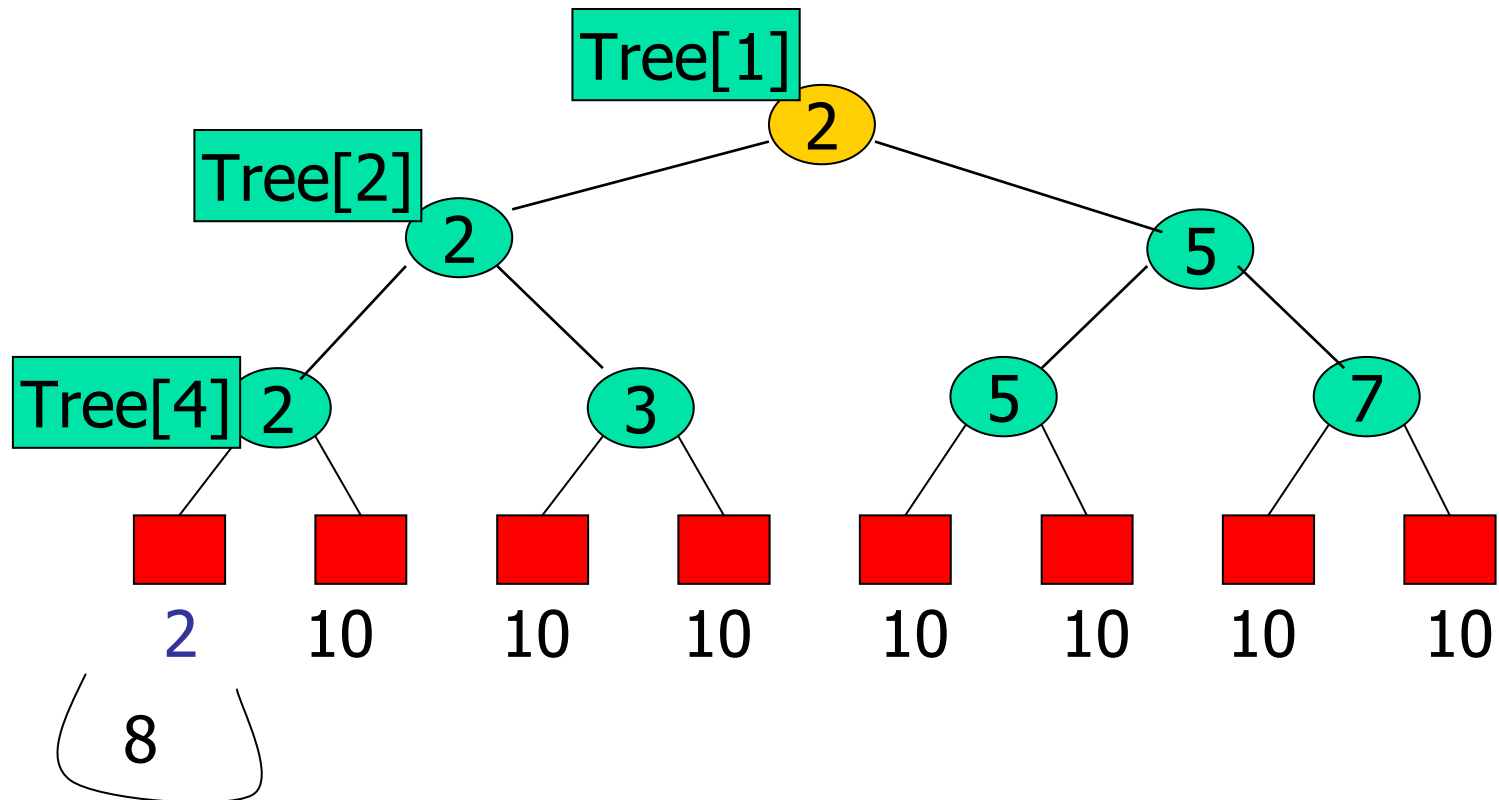
First-Fit and Winner Trees (7)

- Rematch at Tree[3]



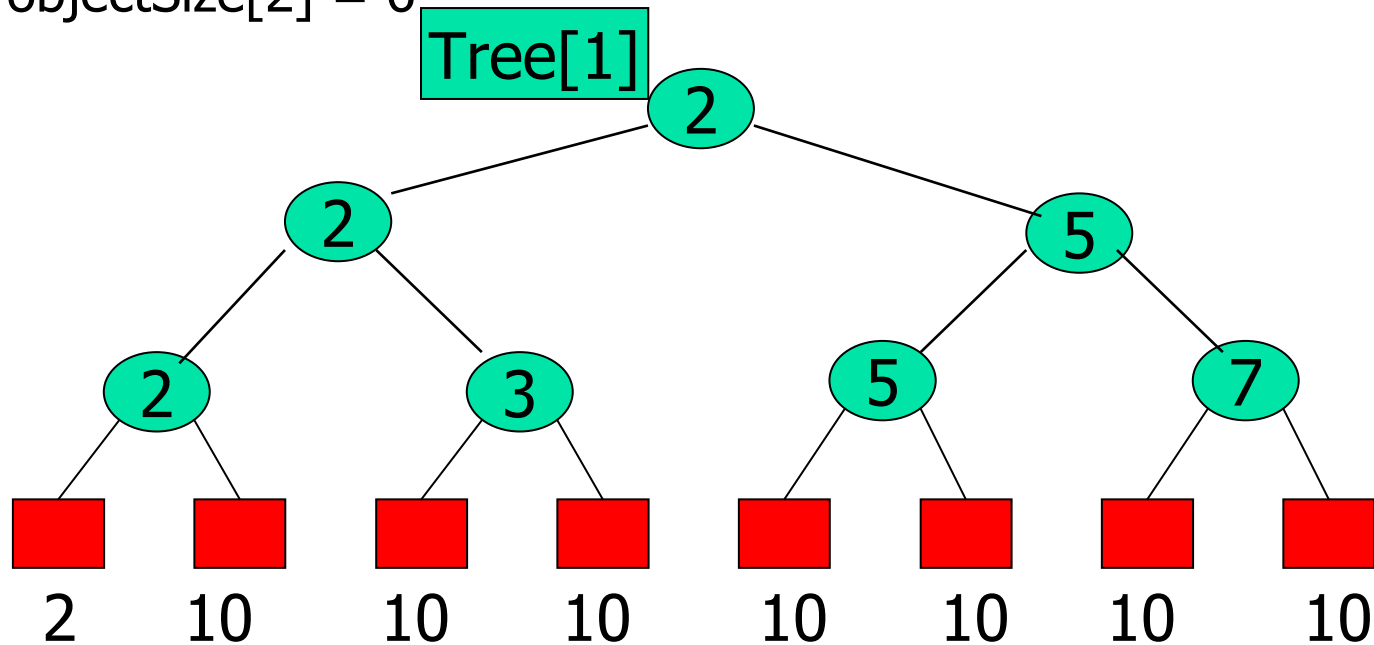
First-Fit and Winner Trees (8)

- Rematch at Tree[1]



First-Fit and Winner Trees (9)

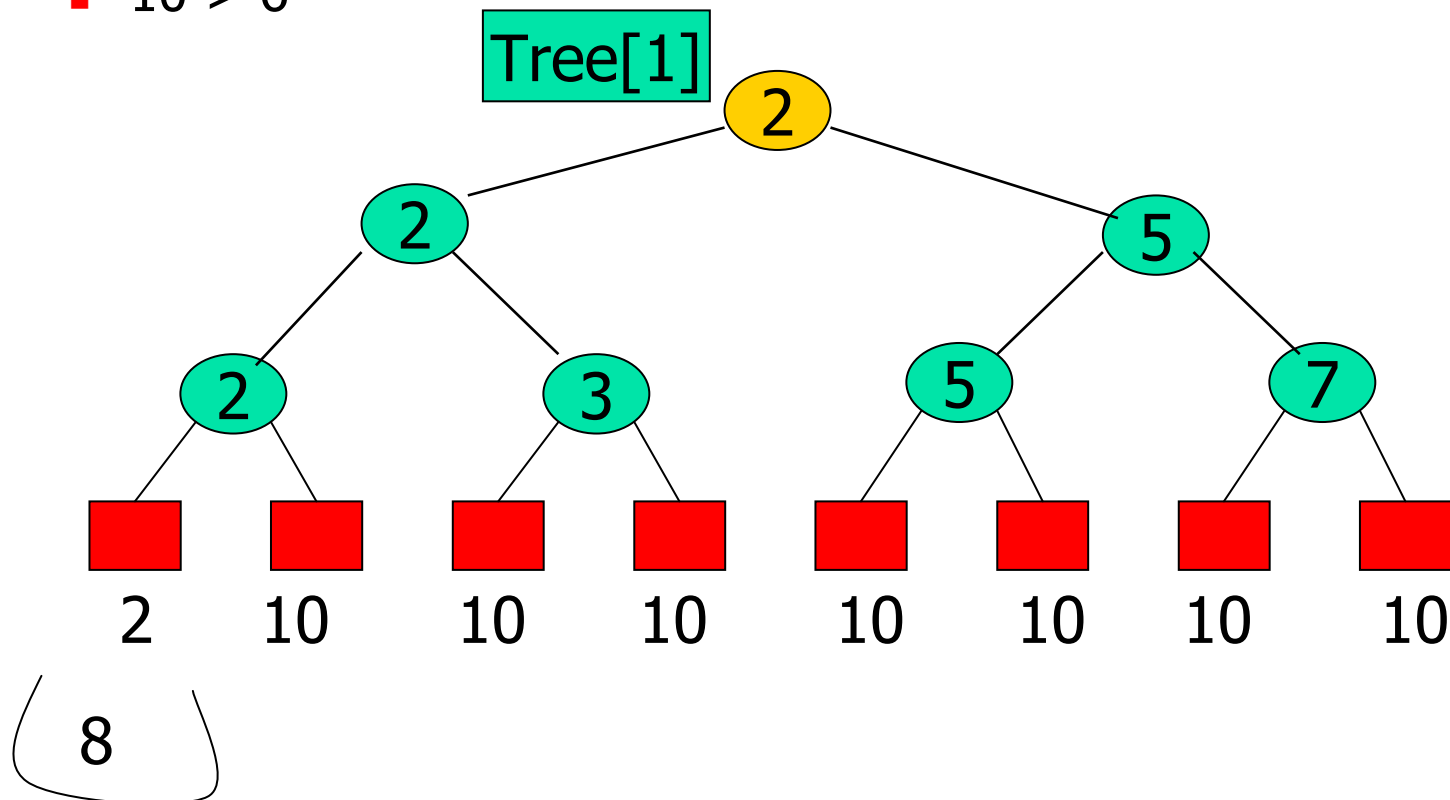
- Suppose objects to be allocated are [8, 6, 5, 3]
 - objectSize[2] = 6



8

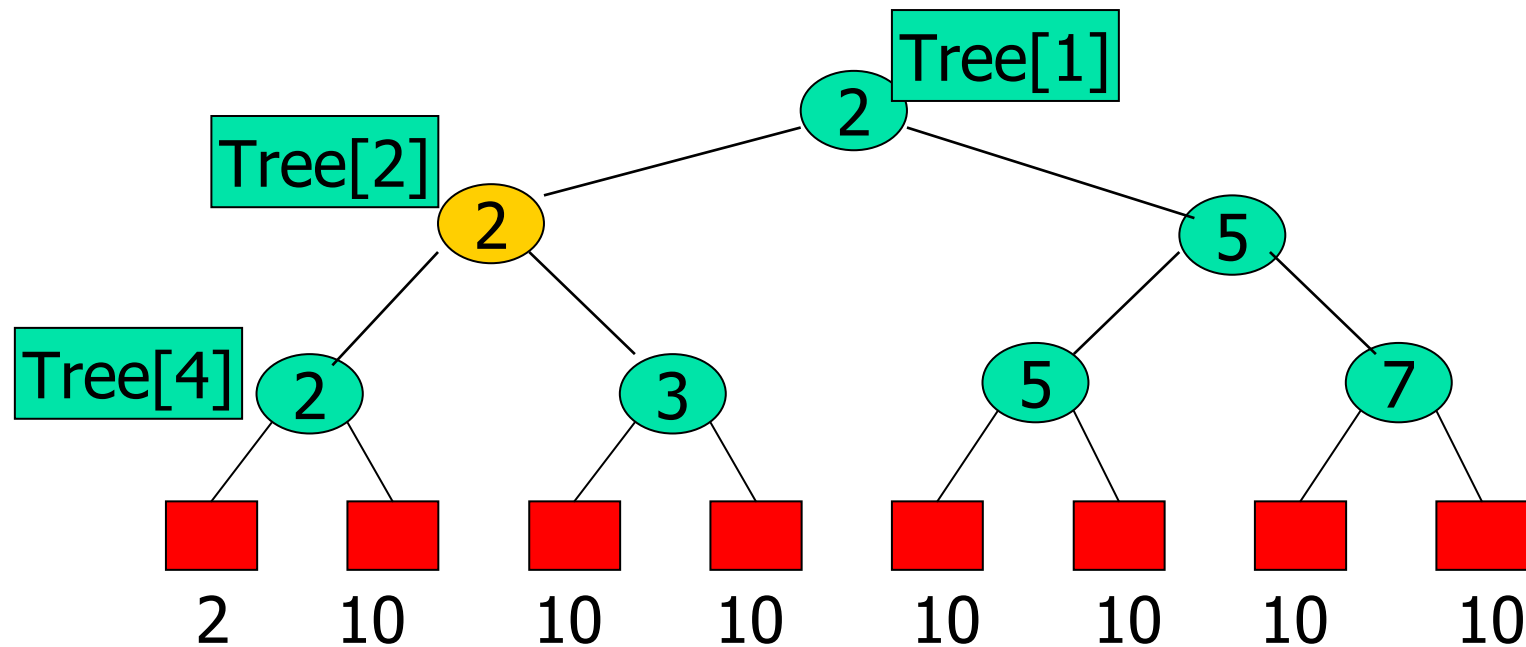
First-Fit and Winner Trees (10)

- $\text{Bin}[\text{tree}[1]].\text{unusedCapacity} \geq \text{objectSize}[2]$
 - $10 > 6$



First-Fit and Winner Trees (11)

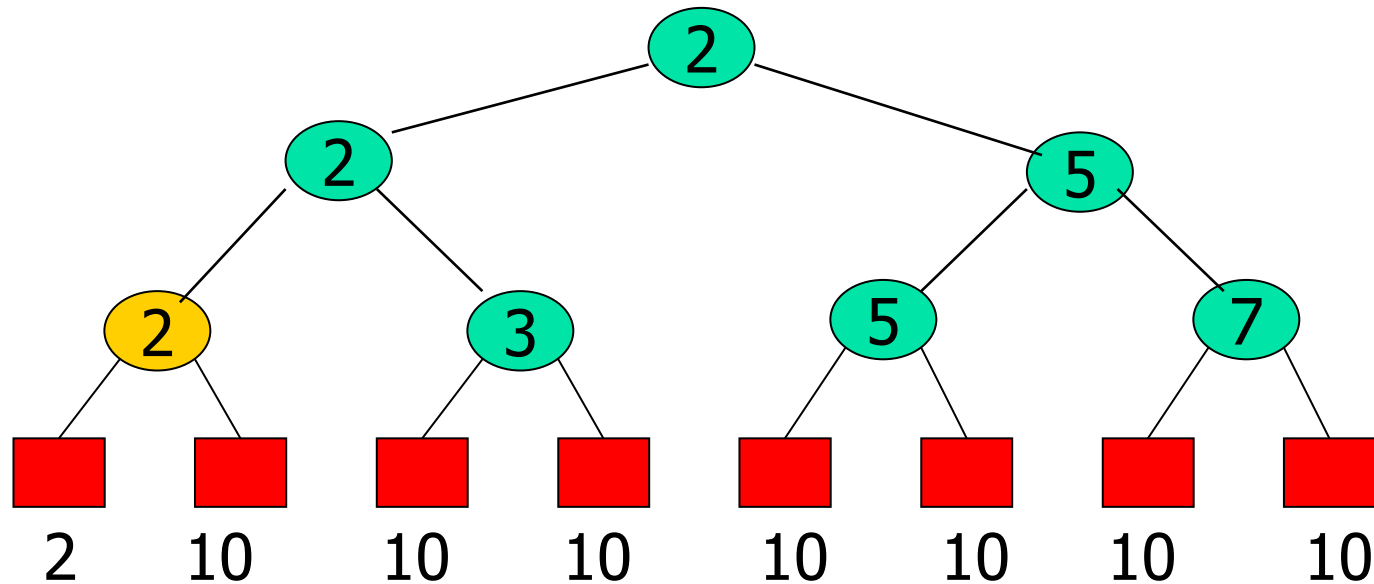
- `Bin[tree[2]].unusedCapacity >= objectSize[2]`
 - `10 > 6`



8

First-Fit and Winner Trees (12)

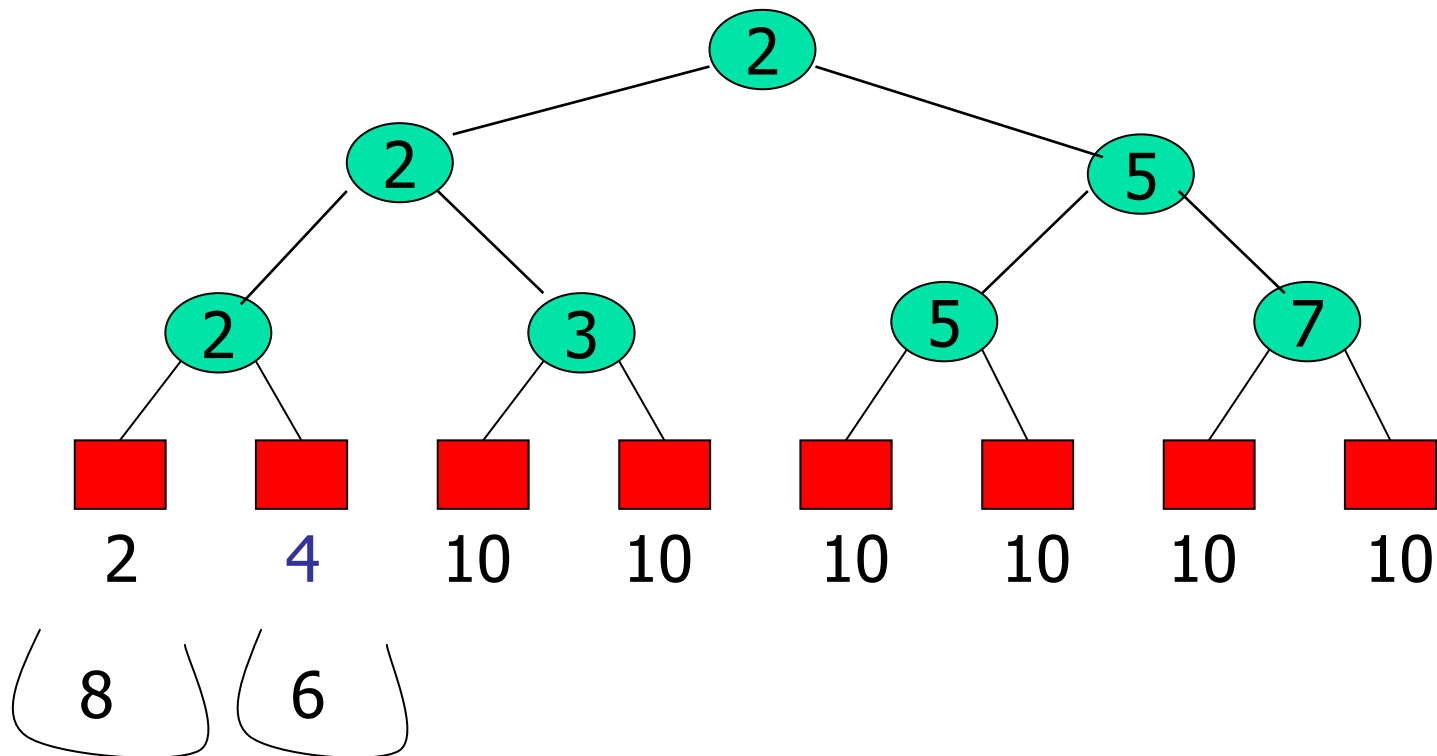
- `Bin[tree[4]].unusedCapacity >= objectSize[2]`
 - `10 > 6`



8

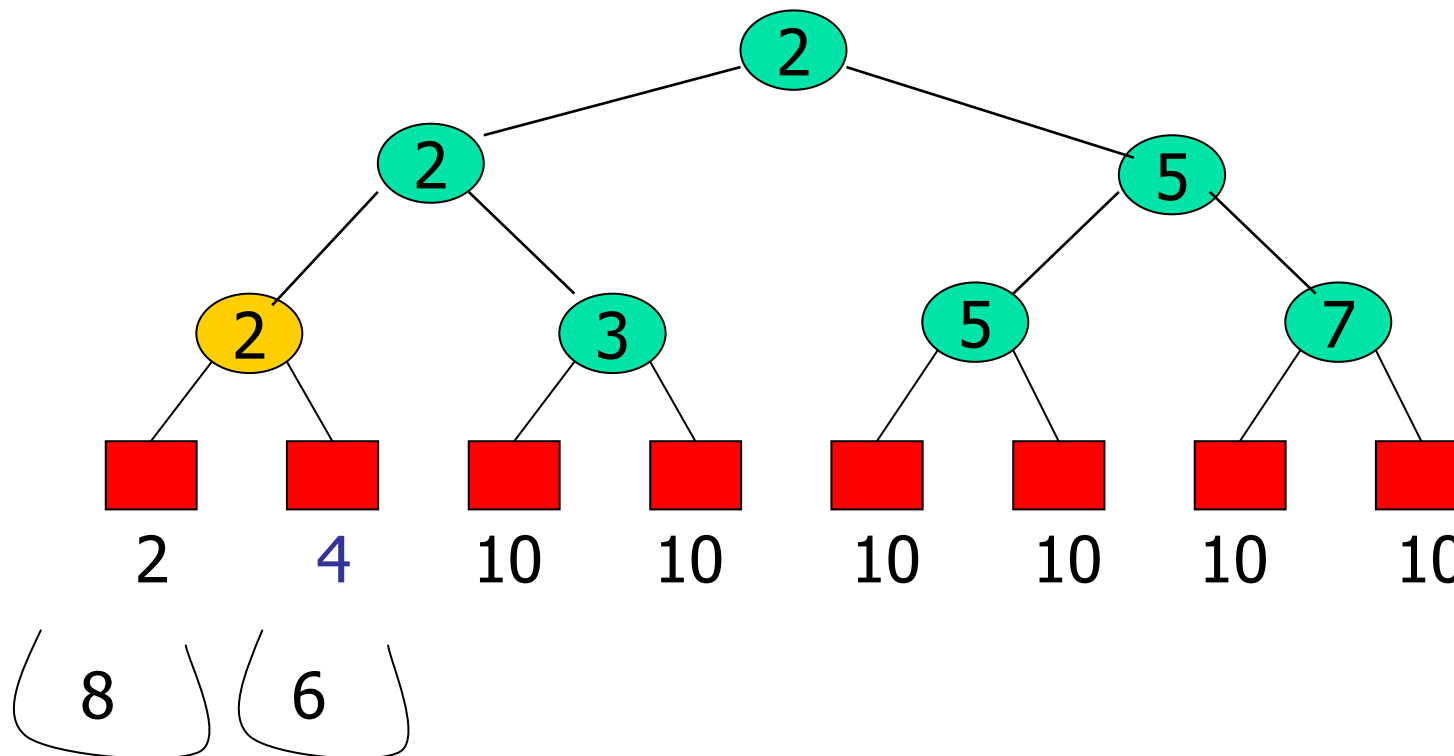
First-Fit and Winner Trees (13)

- Object with size "6" is now in Bin[2]



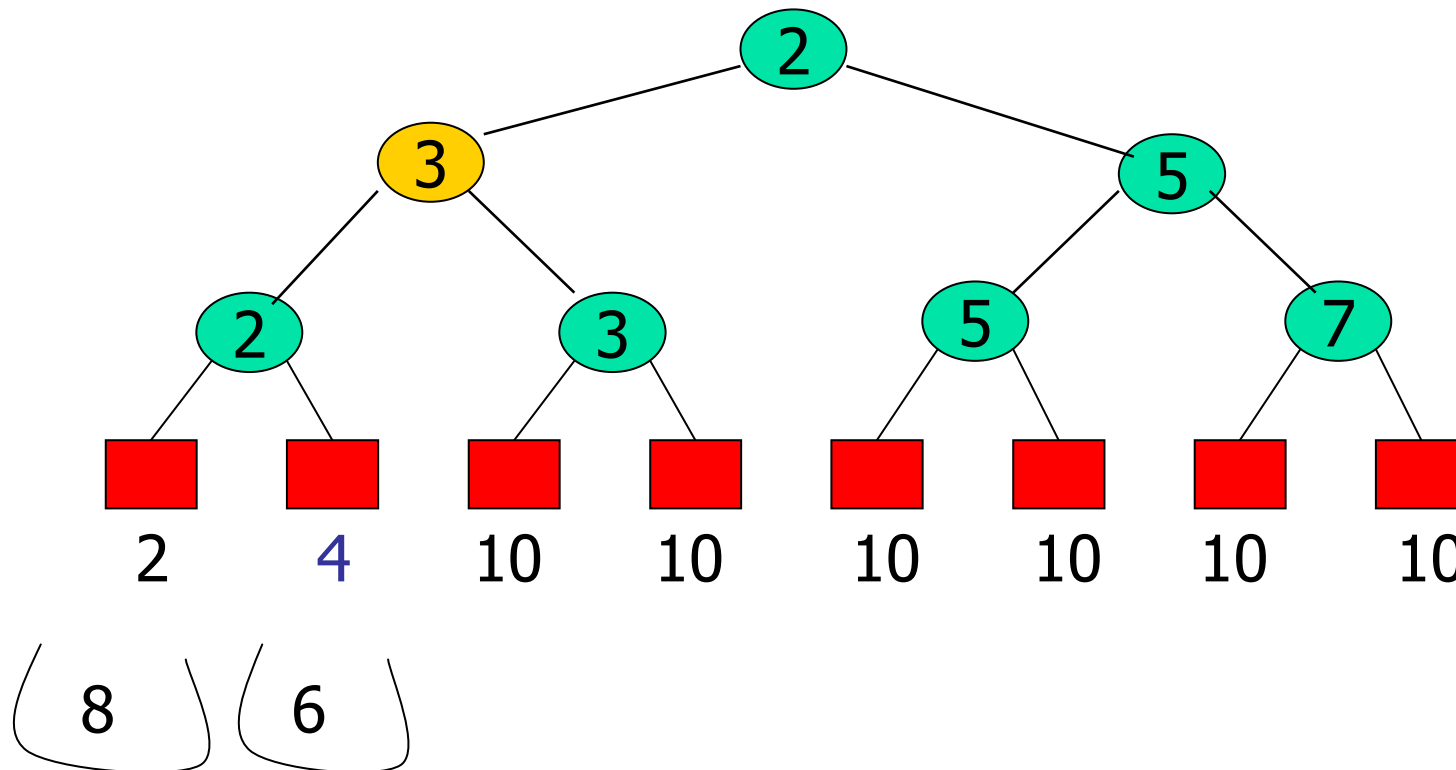
First-Fit and Winner Trees (14)

- Update the winner tree



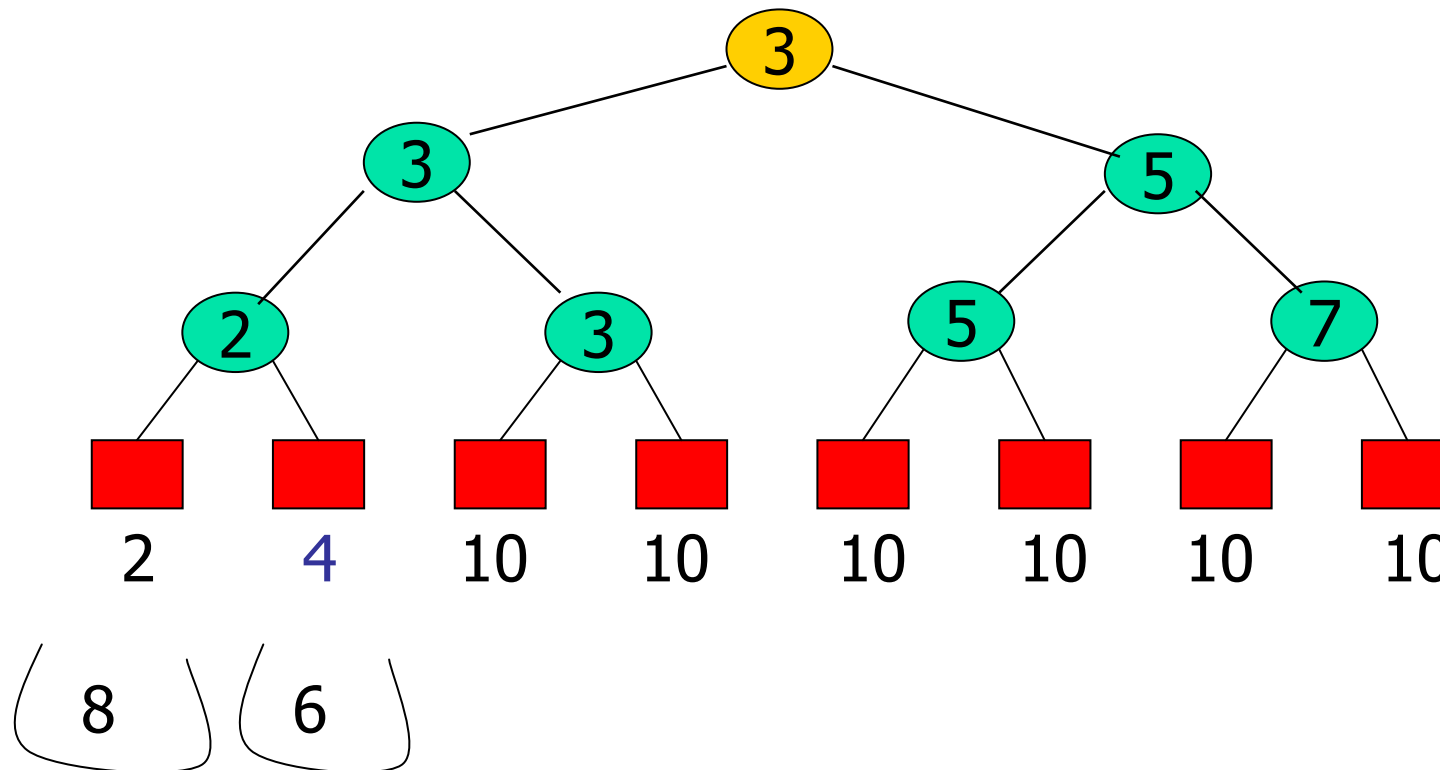
First-Fit and Winner Trees (15)

- Update the winner tree



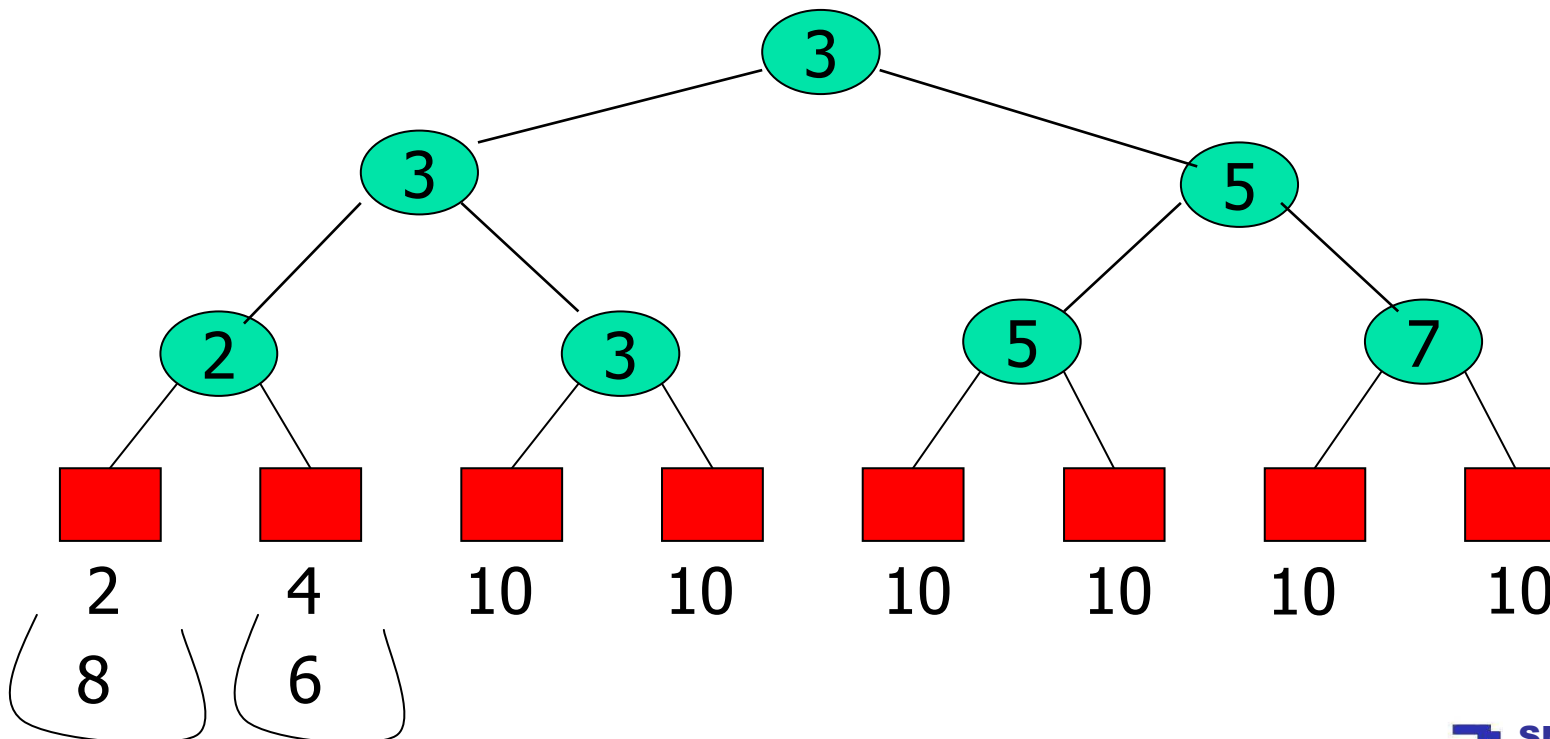
First-Fit and Winner Trees (16)

- Update the winner tree



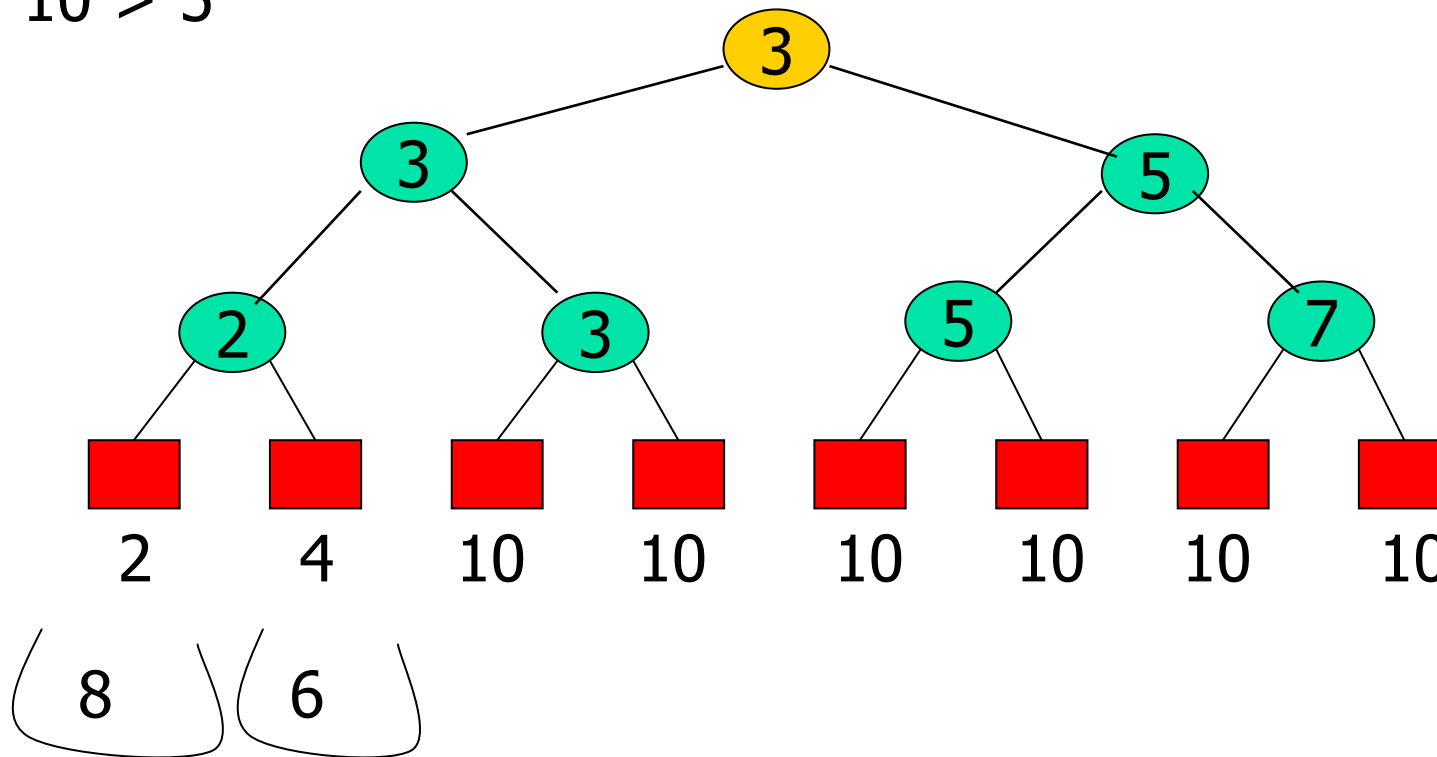
First-Fit and Winner Trees (17)

- Suppose objects to be allocated are [8, 6, 5, 3]
 - `objectSize[3] = 5`



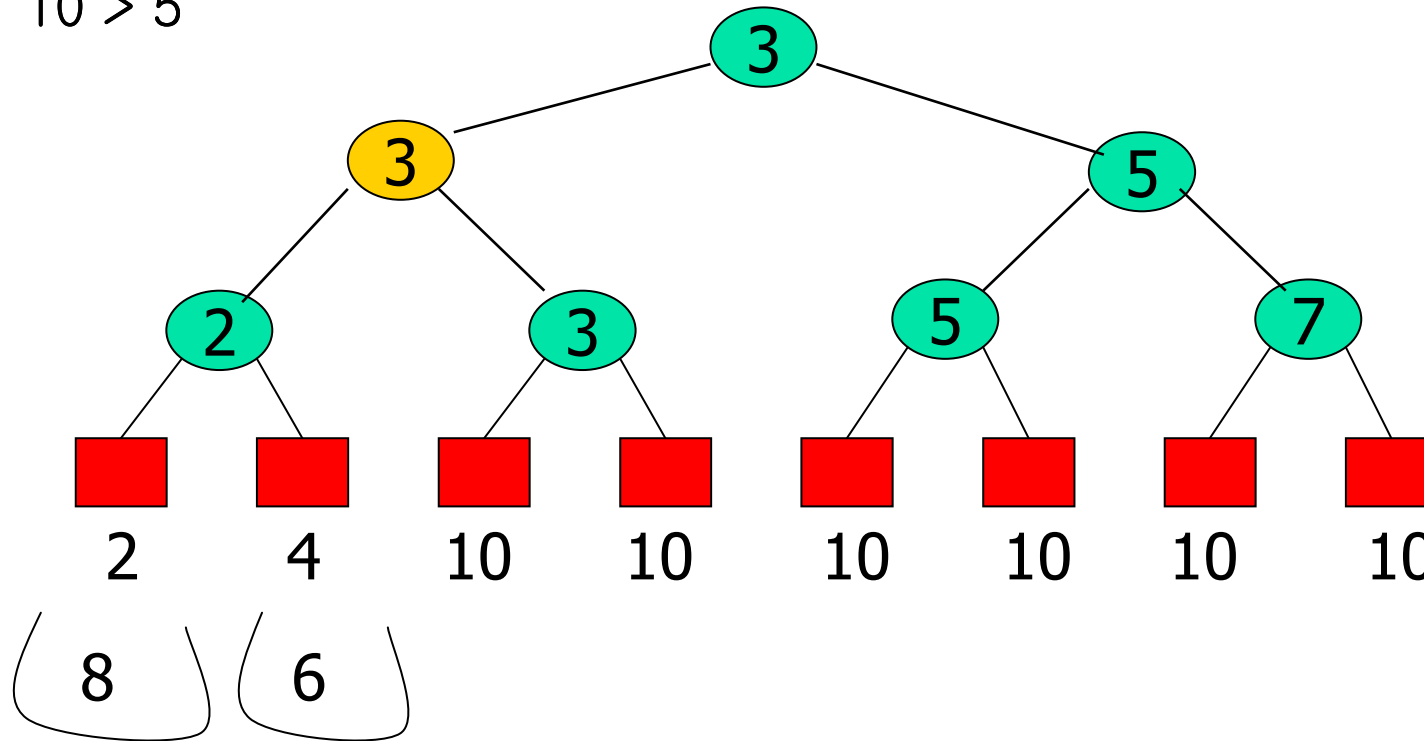
First-Fit and Winner Trees (18)

- `Bin[tree[1]].unusedCapacity >= objectSize[3]`
 - $10 > 5$



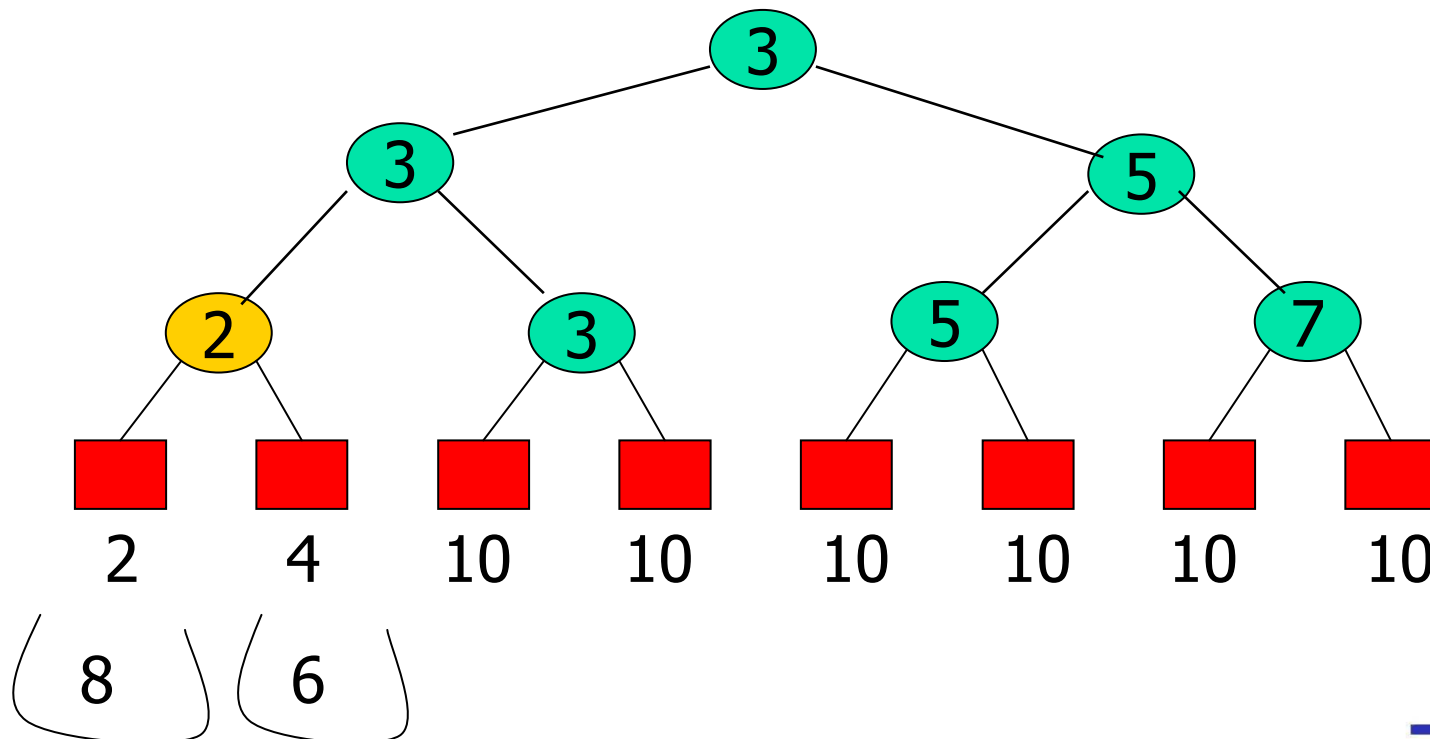
First-Fit and Winner Trees (19)

- `Bin[tree[2]].unusedCapacity >= objectSize[3]`
 - $10 > 5$



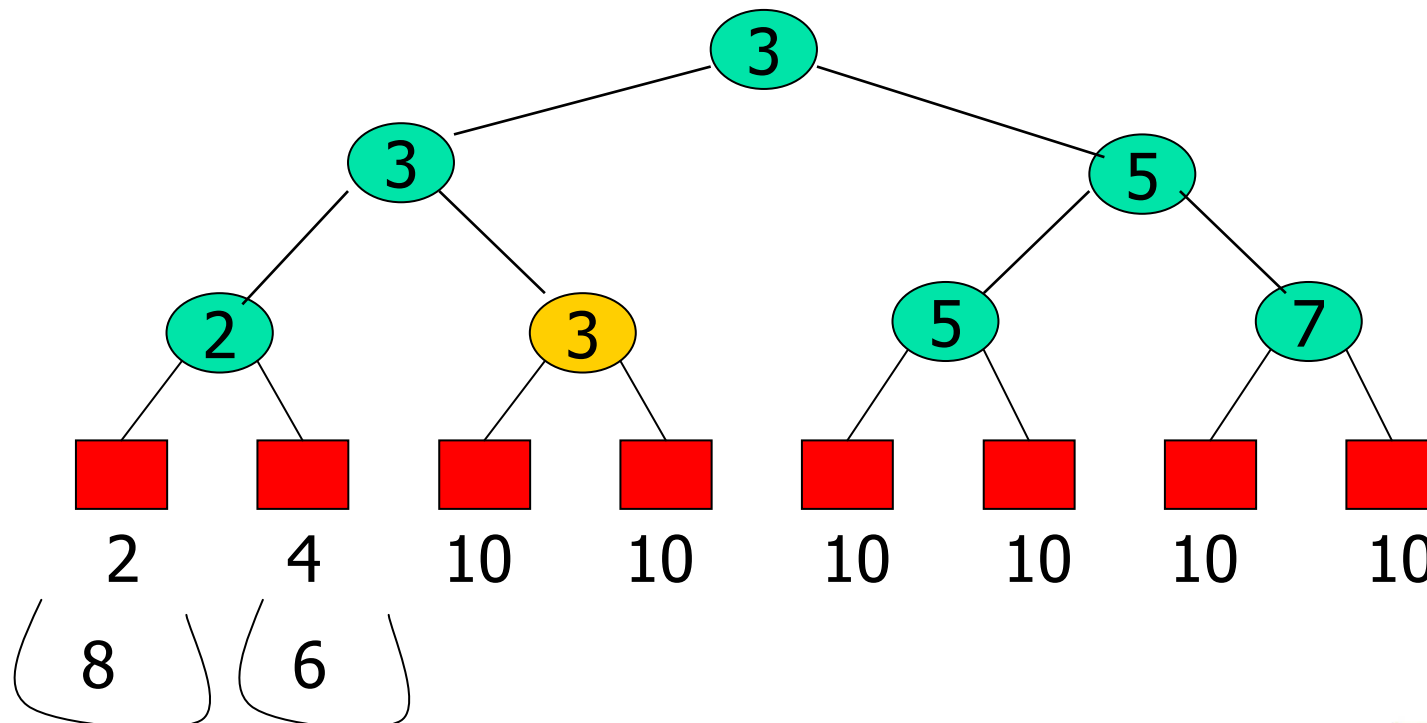
First-Fit and Winner Trees (20)

- `Bin[tree[4]].unusedCapacity < objectSize[3]`
 - $4 < 5$



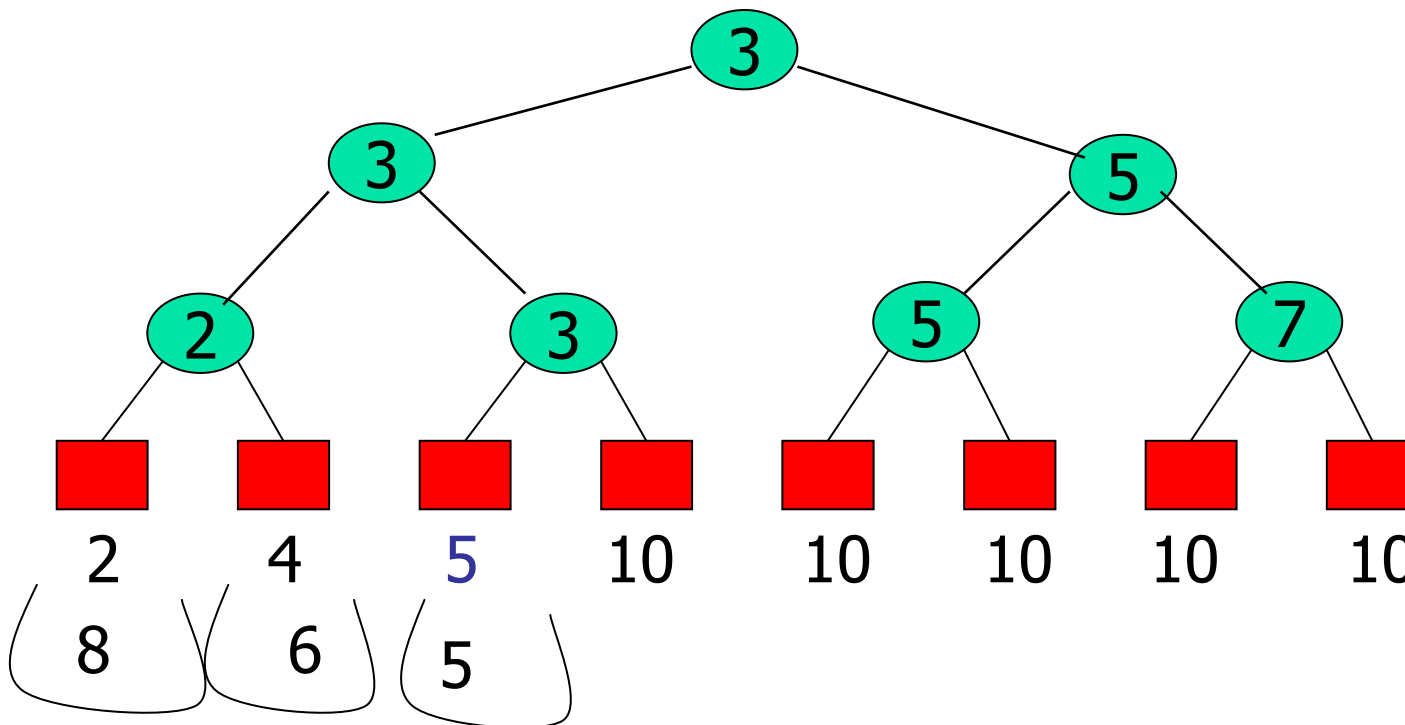
First-Fit and Winner Trees (21)

- Go to sibling: $\text{Bin}[\text{tree}[5]].\text{unusedCapacity} \geq \text{objectSize}[3]$
 - $10 > 5$



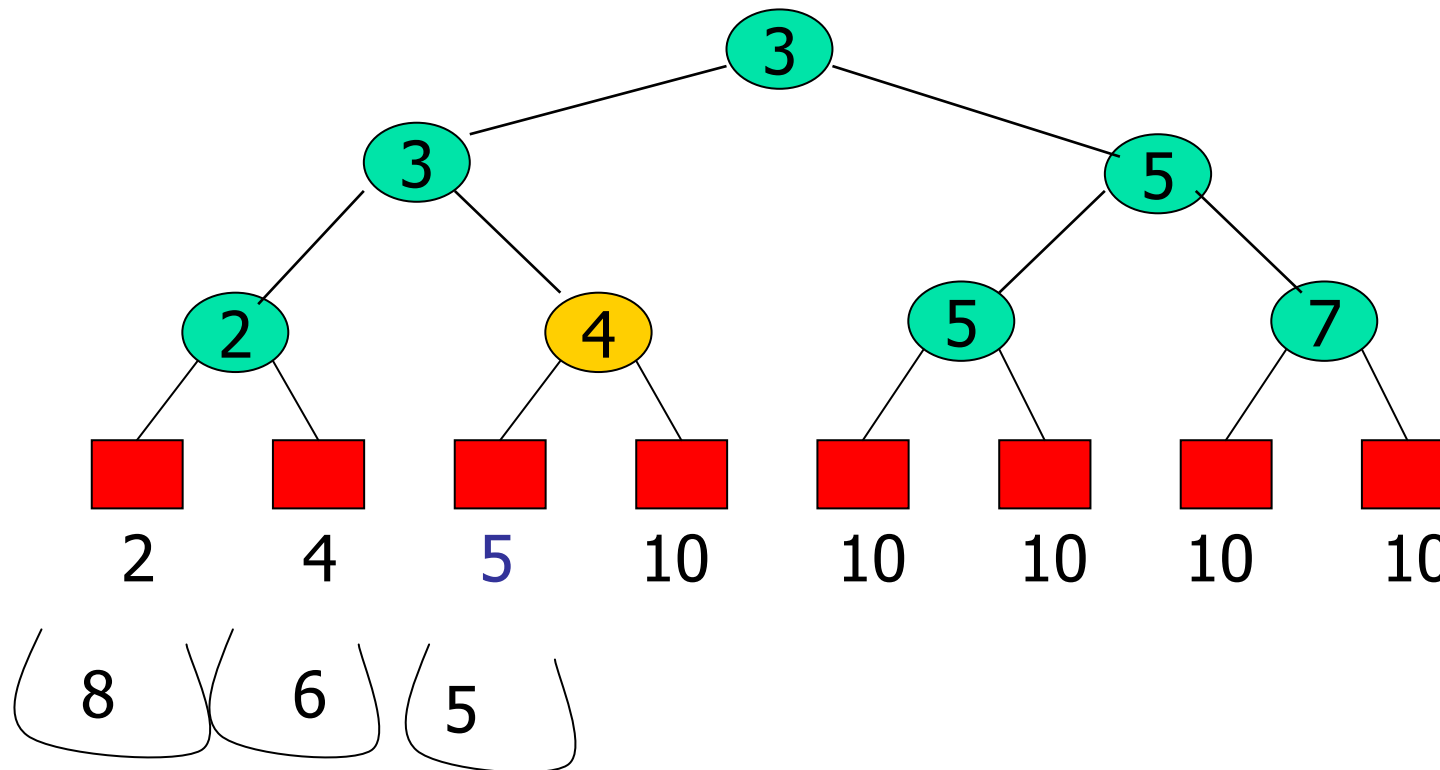
First-Fit and Winner Trees (22)

- Object with size "5" is now in Bin[3]



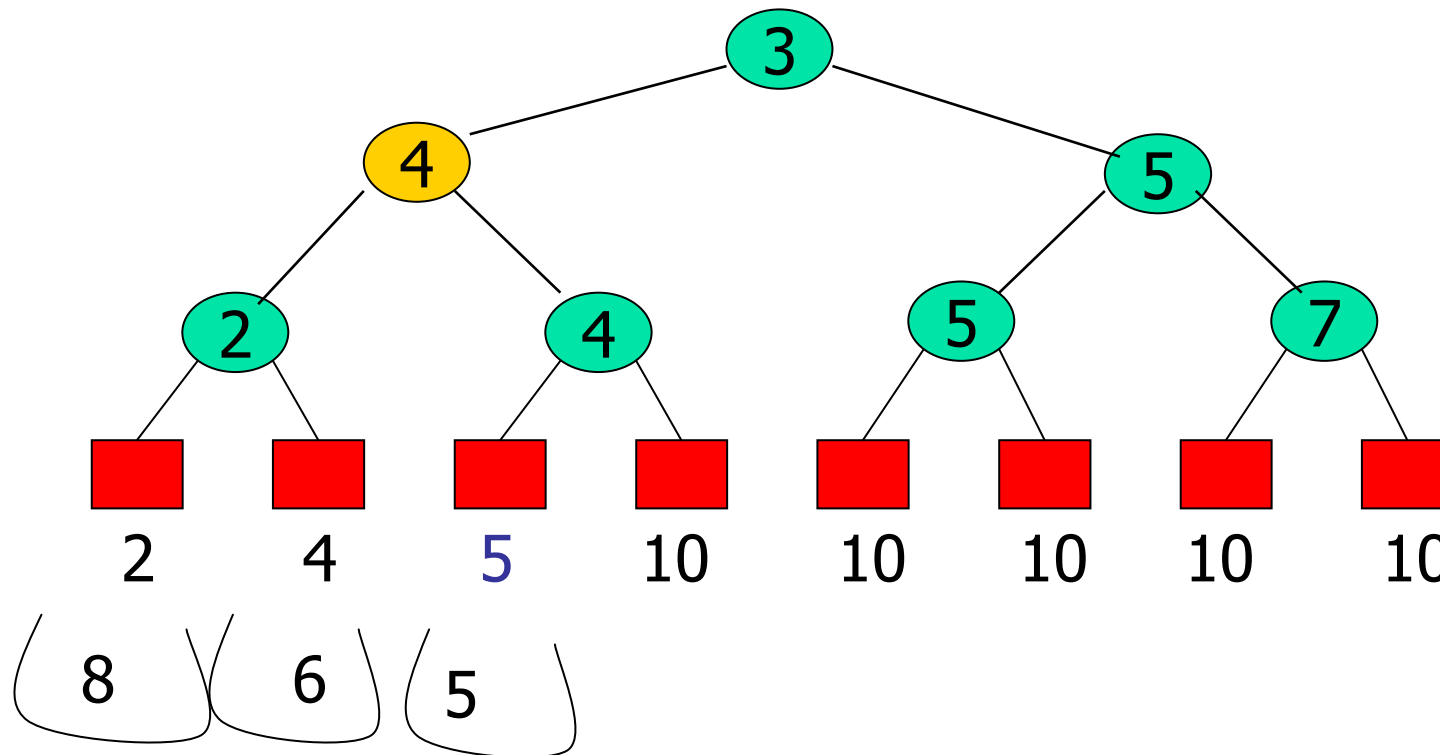
First-Fit and Winner Trees (23)

- Update the winner tree



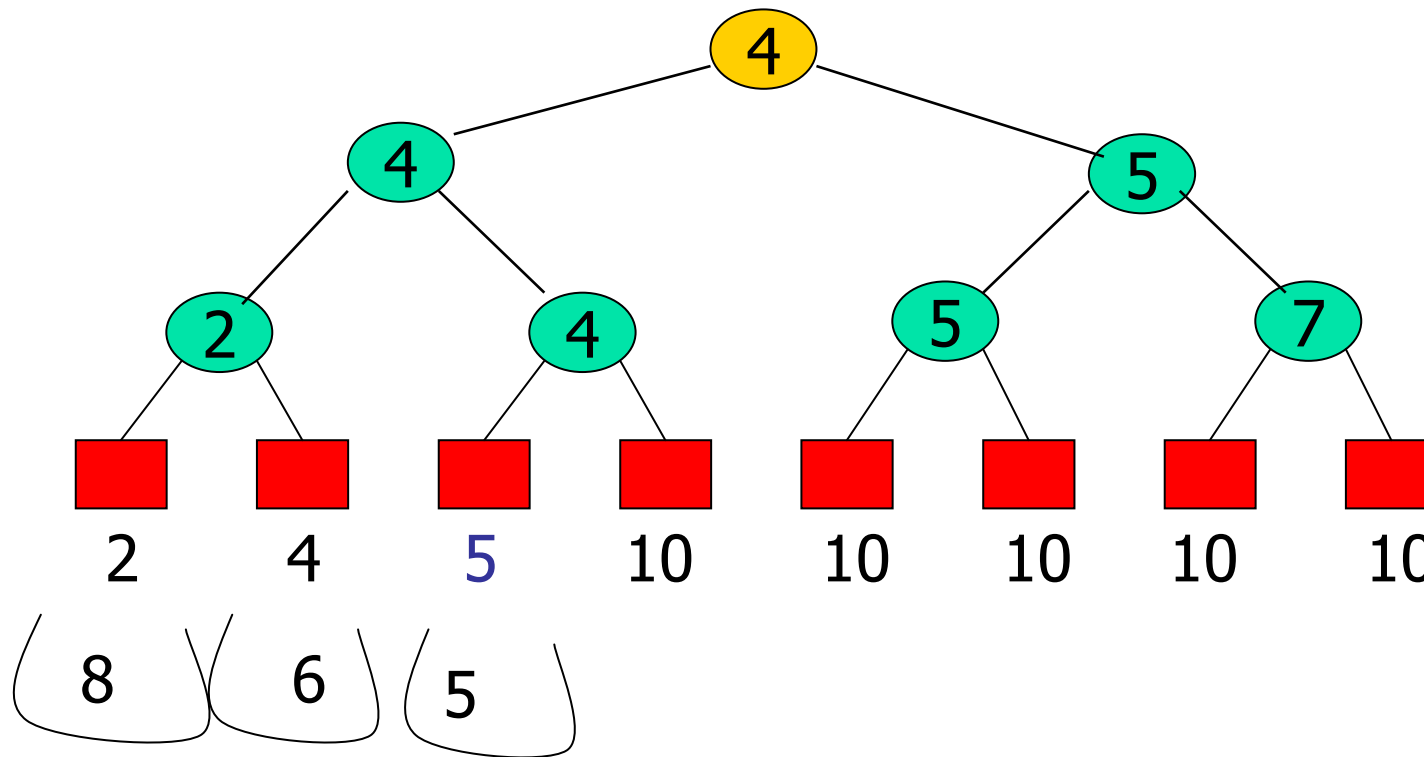
First-Fit and Winner Trees (24)

- Update the winner tree



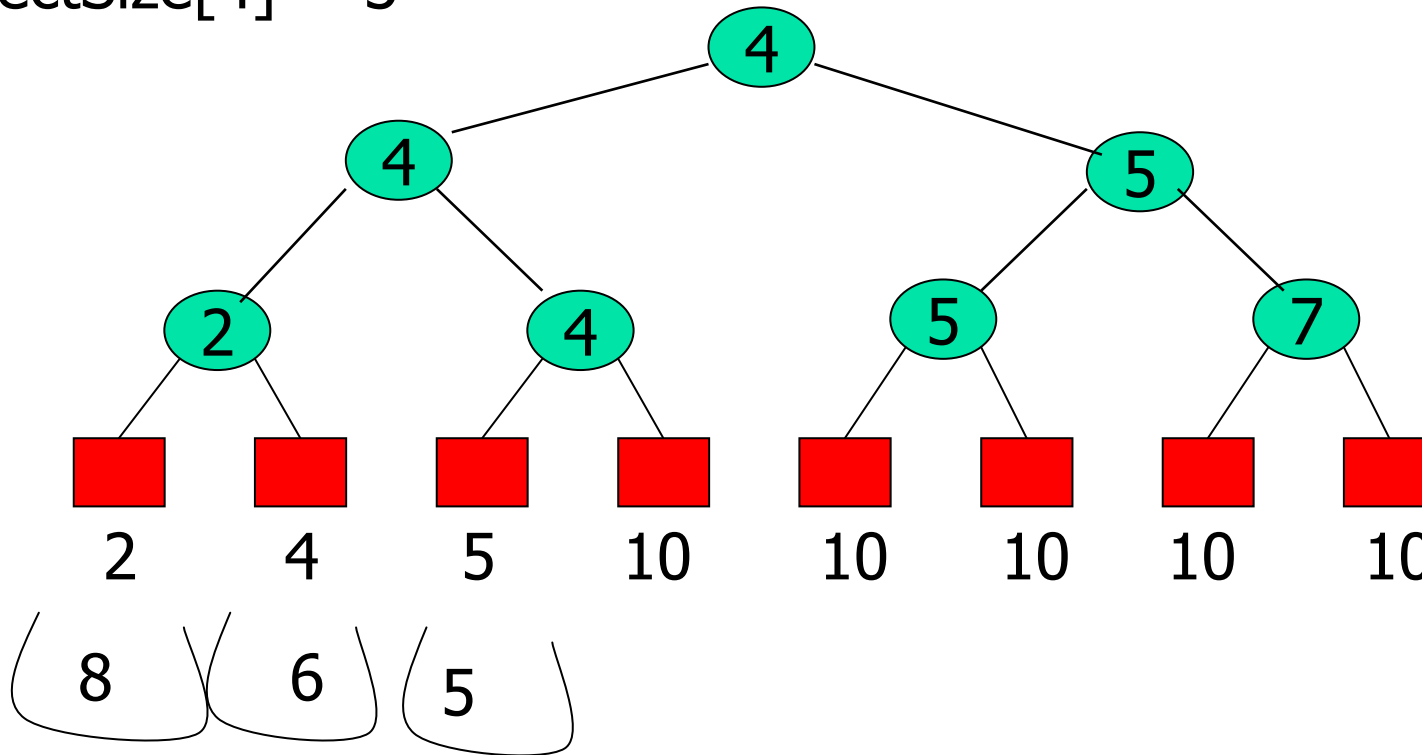
First-Fit and Winner Trees (25)

- Update the winner tree



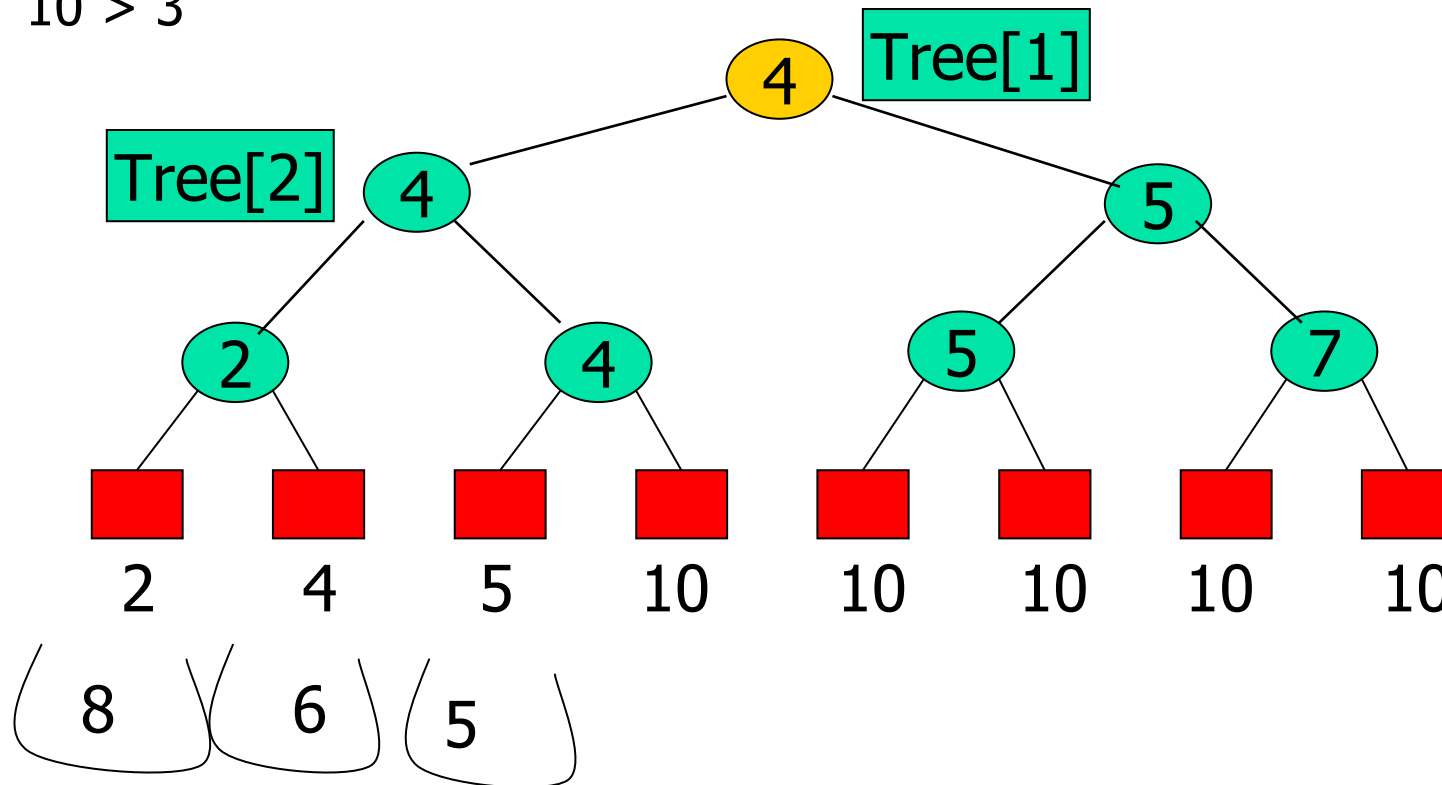
First-Fit and Winner Trees (26)

- Suppose objects to be allocated are [8, 6, 5, 3]
- $\text{objectSize}[4] = 3$



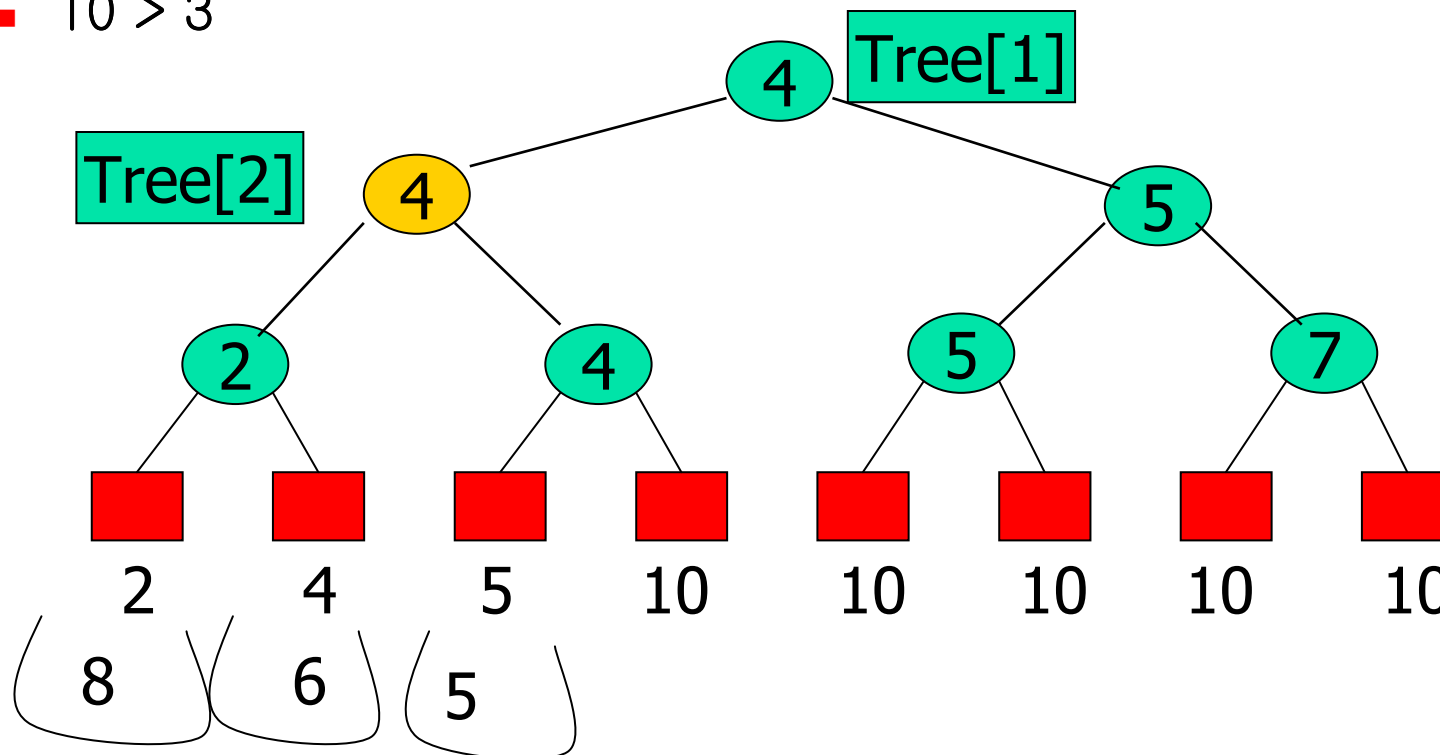
First-Fit and Winner Trees (27)

- `Bin[tree[1]].unusedCapacity >= objectSize[4]`
 - $10 > 3$



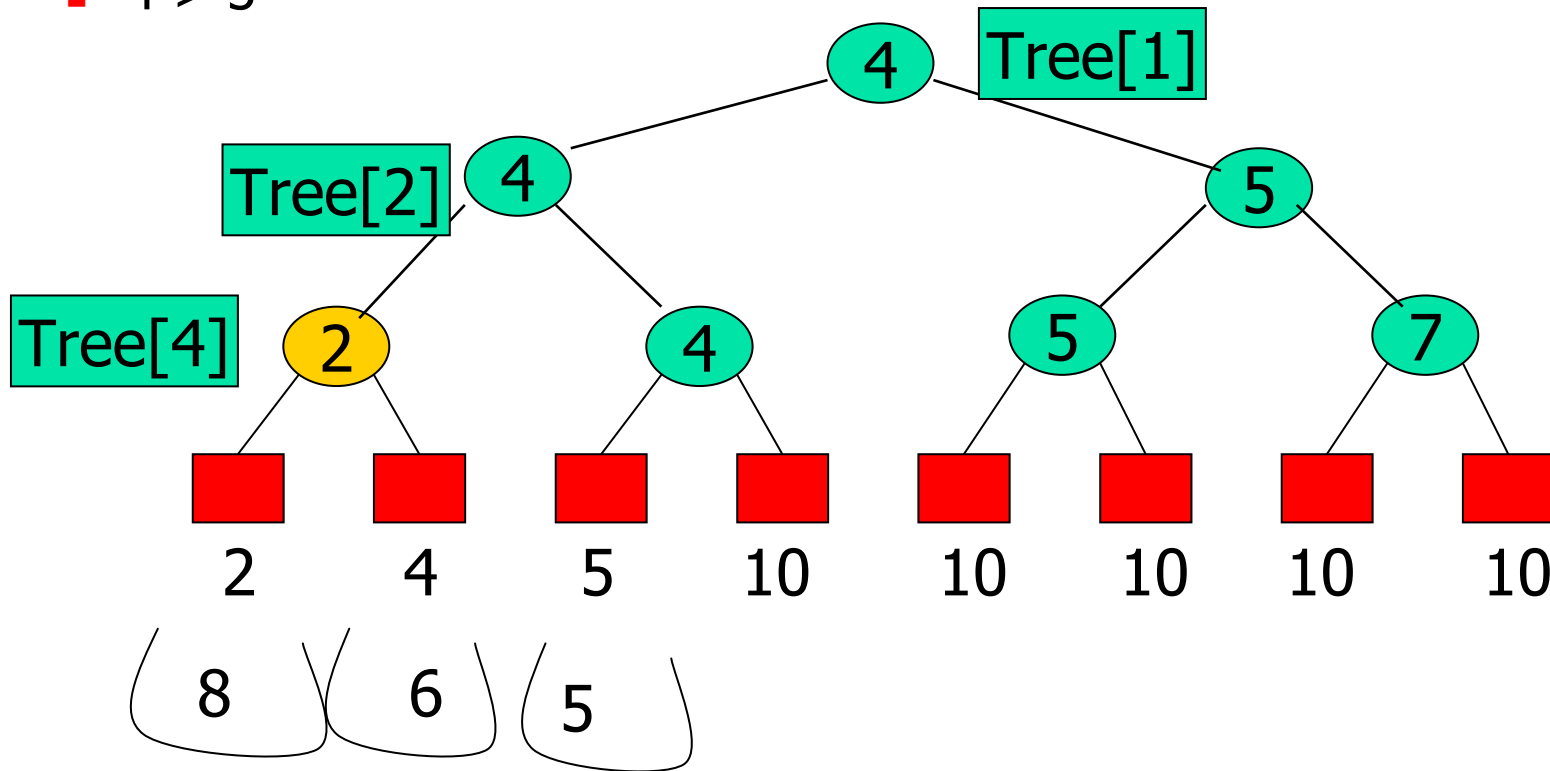
First-Fit and Winner Trees (28)

- $\text{Bin}[\text{tree}[2]].\text{unusedCapacity} \geq \text{objectSize}[4]$
 - $10 > 3$



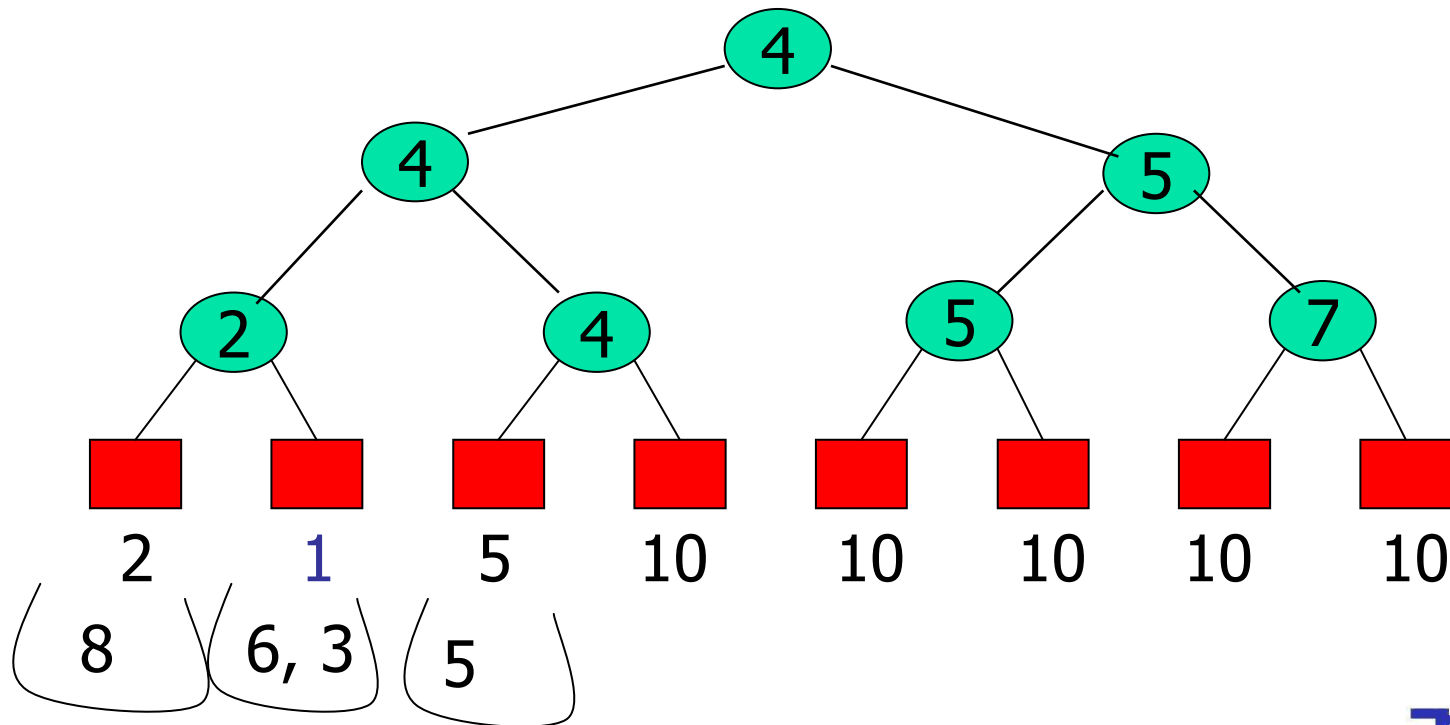
First-Fit and Winner Trees (29)

- `Bin[tree[4]].unusedCapacity >= objectSize[4]`
 - $4 > 3$



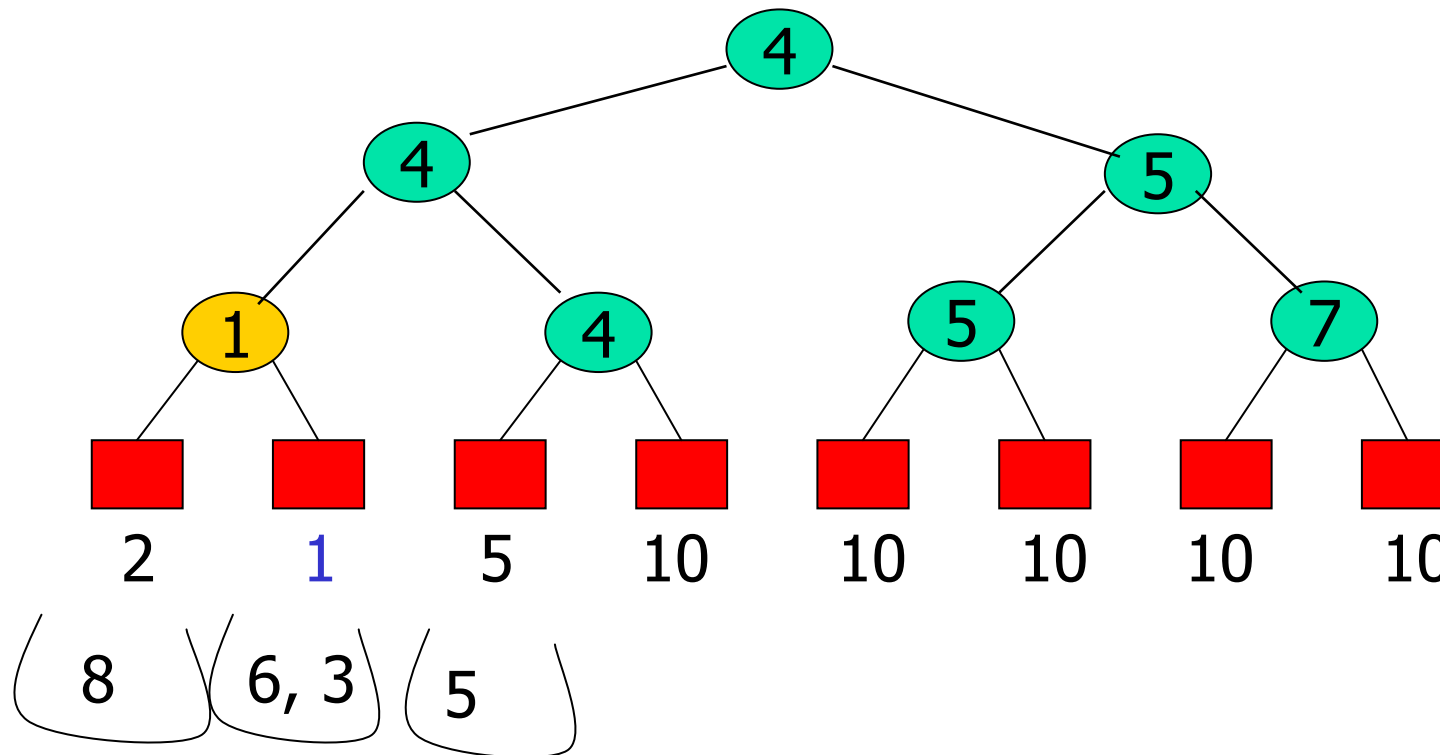
First-Fit and Winner Trees (30)

- Object with size "3" is now in bin[2]



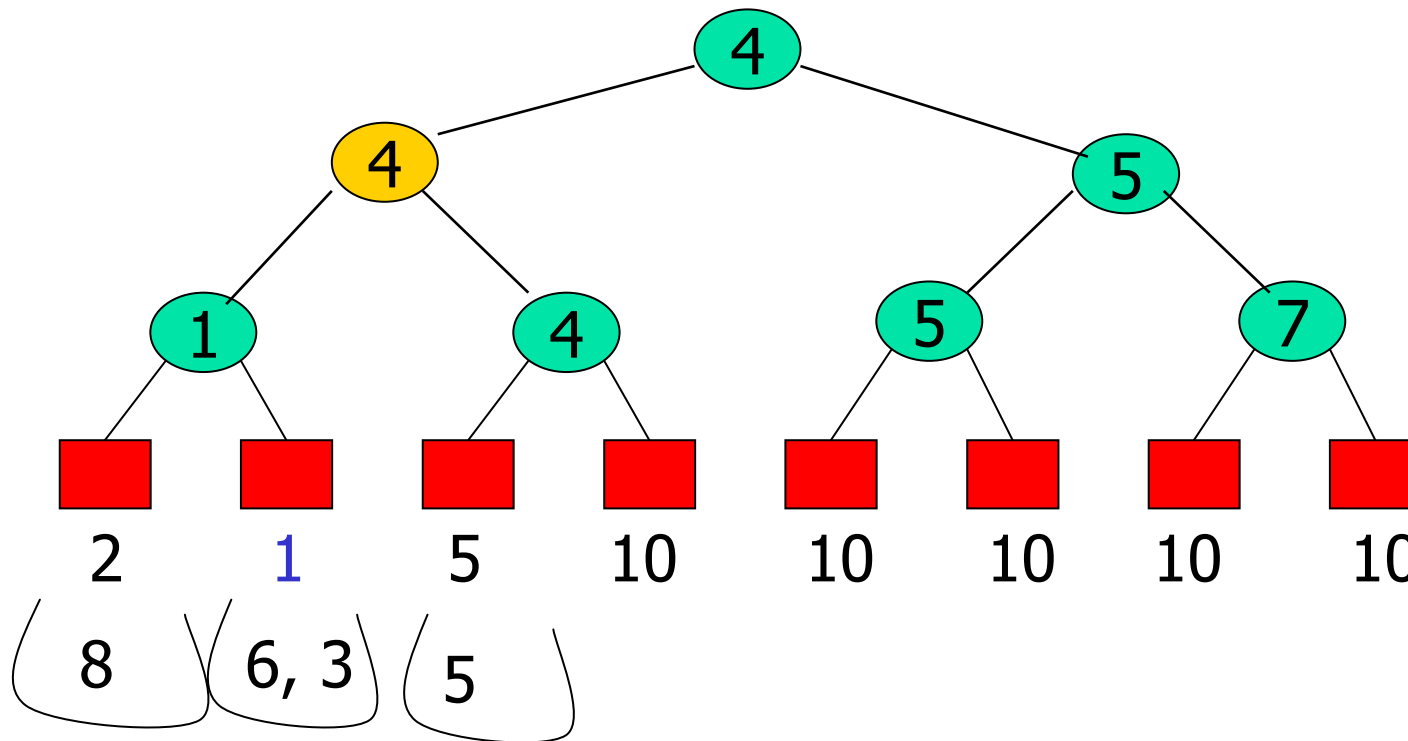
First-Fit and Winner Trees (31)

- Update the winner tree



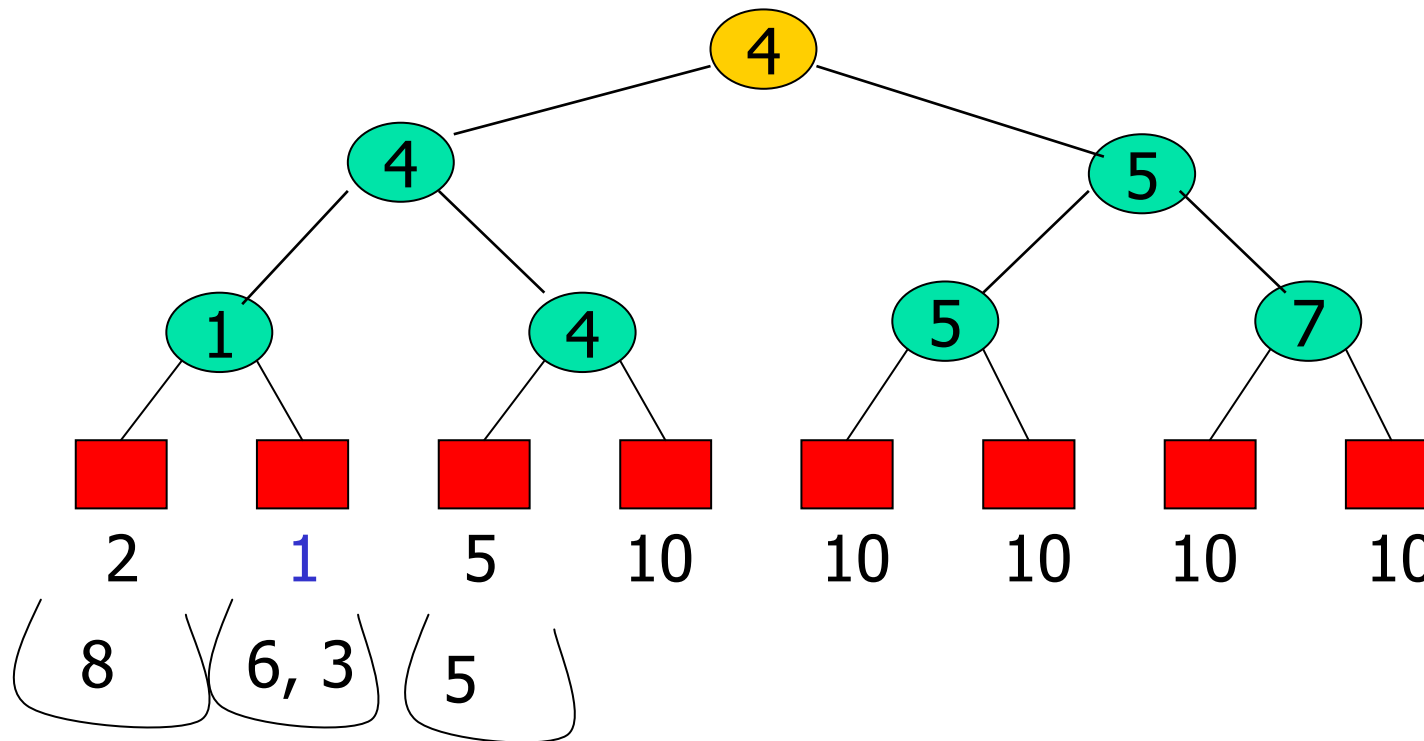
First Fit and Winner Trees (32)

- Update the winner tree



First-Fit and Winner Trees (33)

- Update the winner tree





The method firstFitPack() (1)

```
public static void firstFitPack(int [] objectSize, int binCapacity) {
    int n = objectSize.length - 1; // number of objects
    Bin [] bin = new Bin [n + 1]; // bins
    ExtendedCWTree winTree = new ExtendedCWTree();
    for (int i = 1; i <= n; i++) // initialize n bins and winner tree
        bin[i] = new Bin(binCapacity); // initial unused capacity
    winTree.initialize(bin);
    // put objects in bins
    for (int i = 1; i <= n; i++) { // put object i into a bin
        // find first bin with enough capacity
        int child = 2; // start search at left child of root
        while (child < n) {
            int winner = winTree.getWinner(child);
            if (bin[winner].unusedCapacity < objectSize[i])
                child++; // first bin is in right subtree
            child *= 2; // move to left child }
        }
    }
}
```



The method firstFitPack() (2)

```
int binToUse;           // will be set to bin to use
child /= 2;            // undo last left-child move
if (child < n) { // at a tree node
    binToUse = winTree.getWinner(child);
    // if binToUse is right child, need to check bin binToUse-1.
    // No harm done by checking bin binToUse-1 even if binToUse is left child.
    if (binToUse > 1 && bin[binToUse - 1].unusedCapacity >= objectSize[i])
        binToUse--;
}
else binToUse = winTree.getWinner(child / 2); // arises when n is odd
System.out.println("Pack object " + i + " in bin " + binToUse);
bin[binToUse].unusedCapacity -= objectSize[i];
winTree.rePlay(binToUse);
}
}
** O(nlogn) time using a winner tree
```



Table of Contents

- Winner Trees
- Loser Trees
- Tournament Tree Applications
 - Bin Packing Using First Fit (BPFF)
 - Bin Packing Using Next Fit (BPNF)



Next-Fit

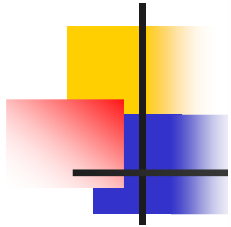
- For the new object, we determine **the next nonempty bin** that can accommodate the object by polling the bins in a round robin fashion
 - We want a feasible packing with **a fewest number of bins**
- 3 bins of size 7 and six objects [3,5,3,4,2,1]
 - 3 goes to bin[1]
 - 5 goes to bin[2] // the candidate is bin[2] & check the left side; bin[2] is OK.
 - 3 goes to bin[1] // the candidate is bin[3] & check the left side; bin[1] is better
 - 4 goes to bin[3] // the candidate is bin[3] & check the left side; bin[3] is OK
 - 2 goes to bin[2] // first check bin[1], but not qualified; bin[2] is qualified
 - 1 goes to bin[3] // first check bin[3], but not qualified; bin[3] is qualified
- Idea ($O(n)$ for one assignment)
 - Search for the next bin of the last used bin which can accommodate the new object
 - If the candidate bin is not empty, use it
 - Otherwise search for the left-most bin which can accommodate the new object



Next-Fit with Winner Tree

- **Step 1 (Figure 14.8 in textbook)**
 - Search the suitable bin **with help of winner tree**
 - If the found bin is empty, go to step 2
 - $O(\log(n))$

- **Step 2 (actually First-Fit)**
 - Search the left-most suitable bin **with help of winner tree**
 - $O(\log(n))$



```
// Find nearest bin to right of
// lastBinUsed into which object i fits.
j = lastBinUsed + 1;
if (bin[j].unusedCapacity >= objectSize[i])
    return j;
if (bin[j+1].unusedCapacity >= objectSize[i])
    return j + 1;

p = parent of bin[j];
if (p == n - 1)
{ // special case
    let q be the external node to the right of tree[p];
    if (bin[q].unusedCapacity >= objectSize[i])
        return q;
}

// move toward root looking for first right
// subtree that has a bin with enough capacity
// subtree to right of p is p+1

p /= 2; // move to parent
while (bin[tree[p+1]].unusedCapacity < objectSize[i])
    p /= 2;

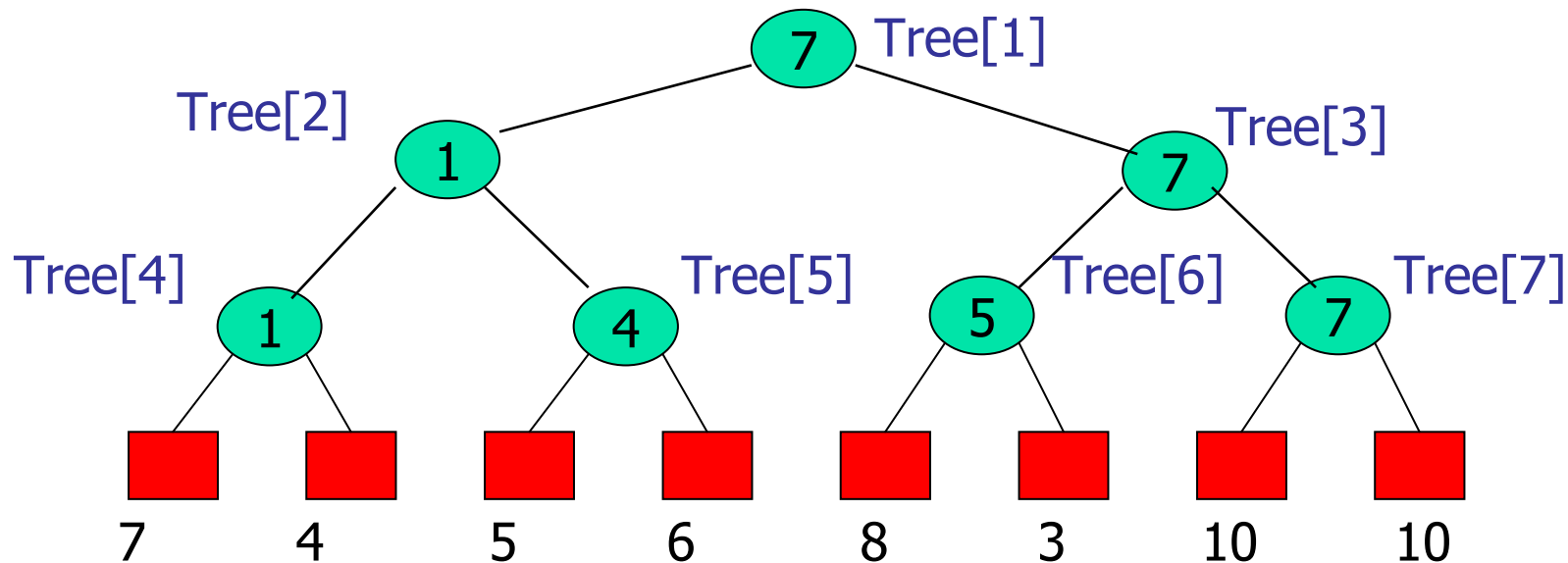
return first bin in subtree p+1 into which object i fits;
```

p

Figure 14.8 Pseudocode for step 1

Next-Fit with Winner Tree (1)

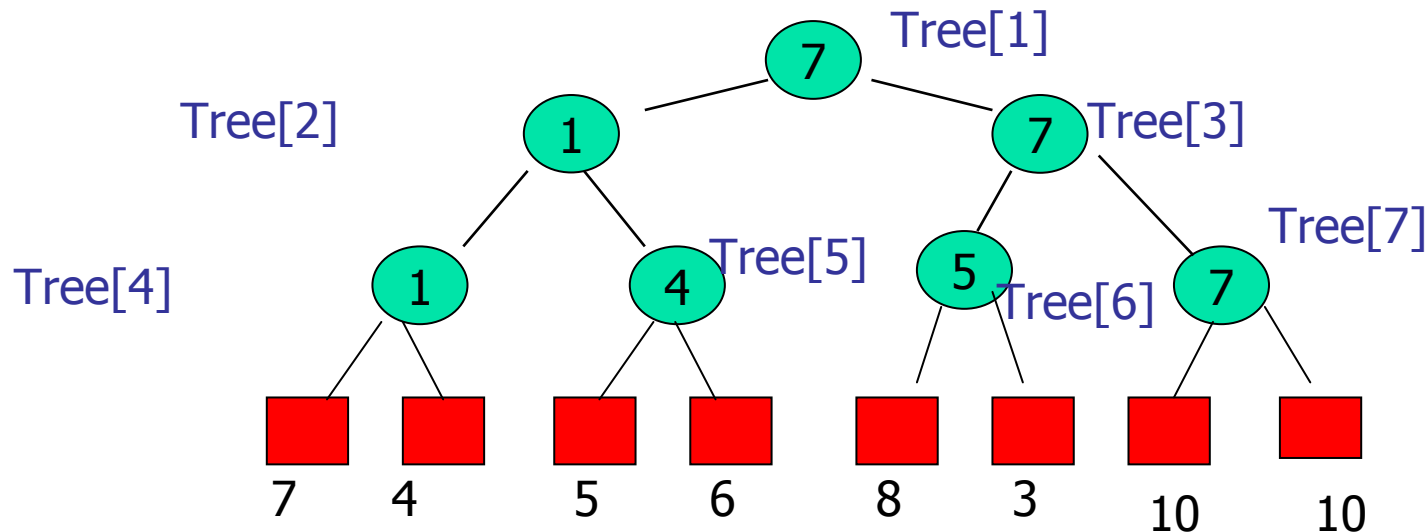
- Suppose the size of a new object to be allocated is 7 & LastUsedBin is bin[1]



- Start from bin[2] & bin[3]; go to parent of bin[2] (which is tree[4])
- go to tree[2] & try tree[3];
- The unusedCapacity of tree[3] is 10, try to find the first bin of tree[3] which can accommodate "7"

Next-Fit and Winner Tree (2)

- Suppose the size of a new object to be allocated is 9 & LastUsedBin is bin[3]



- Start from bin[4] & bin[5]; go to the parent of bin[4] (which is tree[5])
- Go to tree[2] and try tree[3]
- The unusedCapacity of tree[3] is 10, try to find the first bin of tree[3] which can accommodate "9" → bin[7] is the candidate, but empty
- We check the left-most bin which can accommodate "9" → No such bin → bin[7] is the bin to use



Summary

- A **tournament tree** is a complete binary tree that is most efficiently stored by using the array-based binary tree
- Study two varieties of tournament trees
 - Winner tree
 - Loser tree
- Tournament Tree Application
 - Bin Packing Using First Fit (BPFF)
 - Bin Packing Using Next Fit (BPNF)