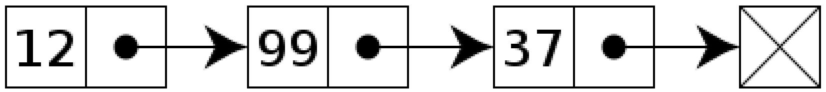
CS218: design and analysis of algorithms

Data Structures

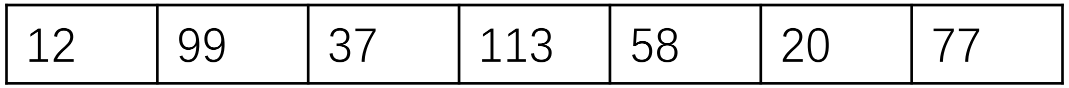
Yan Gu

What data structures have we learned before?

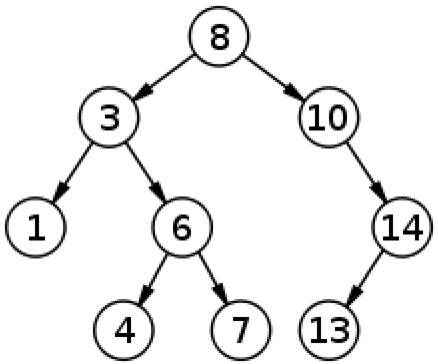
- **Linked list**

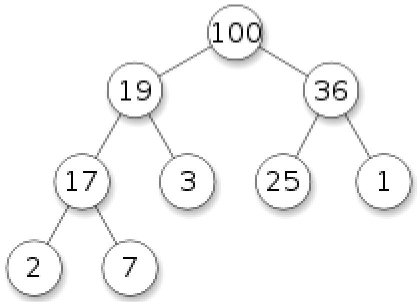


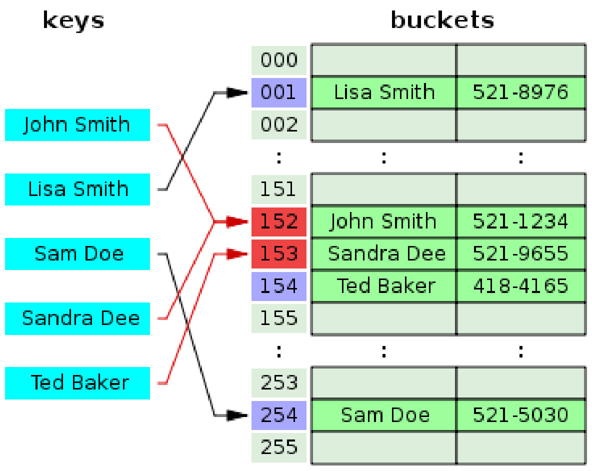
- **Array**
  - 1D, 2D, …



- **Search tree**
  - Binary search tree/multiway search tree/balanced search tree/AVL tree/red-black tree
- **Heap**
  - Binary heap, Fibonacci heap, …
- **Hash Table**
  - Open addressing, close addressing, …







All pictures are from Wikipedia

Construction of Huffman Tree

- **Huffman(C)**
  - n=|C|
  - Q=C // construct a **priority queue** of all character's frequency
  - for i = 1 to n-1
    - allocate a new node z
    - z.left = x = **Extract-Min**(Q)
    - z.right = y = **Extract-Min**(Q)
    - z.freq = x.freq + y.freq
    - **Insert**(Q, z)
  - return **Extract-Min**(Q) // Root of the tree

What priority queue will you use?

Construct Huffman tree using priority queues

Operation needed	Function name
Construct a priority queue (initialize) with n elements	construct (array A)
Find the smallest element and delete it	extract_min()
Insert an element	insert(x)

This is "**priority queue**", it is an "**abstract data type**" (**ADT**).
It specifies an interface of functions

- In Huffman tree construction, we need a "**priority queue**"
- we call construct once
- extract_min and insert n times

Constructe Huffman tree using priority queues

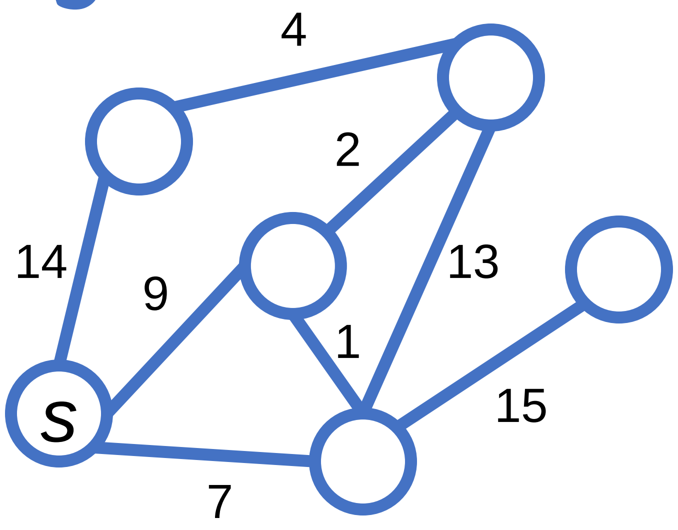
To implement a "**priority queue**", we need some "**data structures**".
Here are some possible implementations.

Data Structure	construct (array A)	extract_min()	insert(x)
Binary heap	Construct a heap from an array (heapify)	Read the root and delete it	Insert x into the heap
	$O(n)$	$O(\log n)$	$O(\log n)$
Balanced binary tree (e.g., AVL tree)	Construct a tree from an array	Chase the left-most branch to find min and delete it	Insert into the tree
	$O(n \log n)$	$O(\log n)$	$O(\log n)$
Sorted array	Sort the array	Find the first element in the array and mark it deleted	Insert x into the middle of the array to keep the order
	$O(n \log n)$	$O(1)$	$O(n)$
Unsorted array	Put everything in the array (nothing to do)	Traverse the array to find the min and mark it deleted	Put x at the end
	$O(1)$	$O(n)$	$O(1)$

Is the abstraction of “priority queue” useful?

- Is it only used in Huffman tree construction?
- Do you know other algorithms using priority queues?
- Indeed, a lot of greedy algorithms essentially use priority queues

Dijkstra's algorithm and Prim's algorithm



- Given a graph and a source vertex s, find the shortest distance from s and all the other vertices

- $\delta(u) = \infty$ for all $u \in V$, 0 for $\delta(s)$
- $S = \emptyset$
- $Q = \{s\}$
- while $Q \neq \emptyset$
  - $u =$ Extract-Min(Q)
  - $S = S \cup \{u\}$
  - for each $v \in N(u)$
    - $\delta(v) = \min\{\delta(v), \delta(u) + w(s, v)\}$
    - Insert/Update $\delta(v)$ in Q

- $\delta(u)$: tentative distance
- S: settled set
- Q: priority queue
- $w(u, v)$: weight of edge from u to v
- $N(u)$: neighbor set of u

Is the abstraction of "priority queue" useful?

- **A lot of greedy algorithms can essentially use priority queues**
  - Because they make "greedy" choices
- **Candy buying: find the cheapest candy**
  - construct, extract_min. No need to insert or update
  - Using sorted array, heap, or balanced binary tree will all work ($O(n \log n)$ cost)
- **Dijkstra's and Prim's algorithm**
  - extract_min, insert, update. No need to construct (start from a single element)
  - Using sorted array gives you $O(n^2)$ time.
  - Using binary heap or balanced binary tree gives you $O(m \log n)$ time
  - Using something called "Fibonacci heap" gives you $O(n \log n + m)$ time
- **Huffman Code: Construct, extract_min, insert**
- **Optimization with submodularity: CELF (lazy update)**

What other abstract data types do you know?

Example: ADTs in STL:

- **Ordered set/maps**
  - (std::map, std::set)
  - implemented by red-black trees
- **Unordered set/maps**
  - (std::unordered_map, std::unordered_set)
  - implemented by hash tables

Why they are called "std::map / std::unordered_map" instead of "std::red_black_tree / std::hash_table"?

What's the difference between sets and maps?

Use ADT and data structures for algorithm design

- Your algorithm may need to access and organize data:
    - How to store data?
    - What query to support? (lookup, findMin, findSum, …)
    - What update to support? (insertion, deletion, filter, multi_insert, delete_min, …)
- Based on the functions needed, you define an **abstract data type (ADT)**!
    - You don't care how they are supported (data organization/algorithms/cost bounds) for an ADT
- Then you find a **data structure** to support them
    - A concrete way to organize data and concrete algorithms to support the functions
    - They may have different costs for different functions
    - So based on your algorithm, you choose the best data structure

Examples of ADT

- FIFO Queue
- Deque (double-ended queue)
- Stack
- Priority queue
- Ordered set/map
- Unordered set/map

- (sometimes we say a queue or a stack is a data structure when we are talking about a concrete implementation)

In this course

- **Winning trees**
  - An implementation of priority queue
  - Easy to implement, same bound as binary heap

- **Augmented trees**
  - Range-related queries: 1D range max/min/sum
  - Rank/selection on trees

Winning Tree

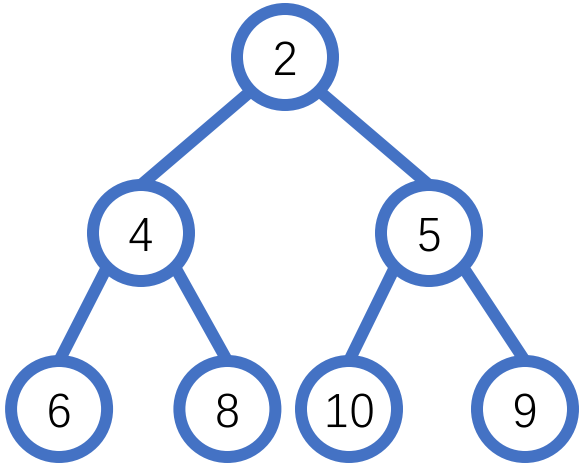
Priority queue

- **Store a set of keys**
  - find/delete smallest/largest key
  - update the keys
  - Insert/delete

Operation needed	Function name
Construct a priority queue (initialize) with n elements	construct (array A)
Find the smallest element and delete it	extract_min()
Insert an element	insert(x)

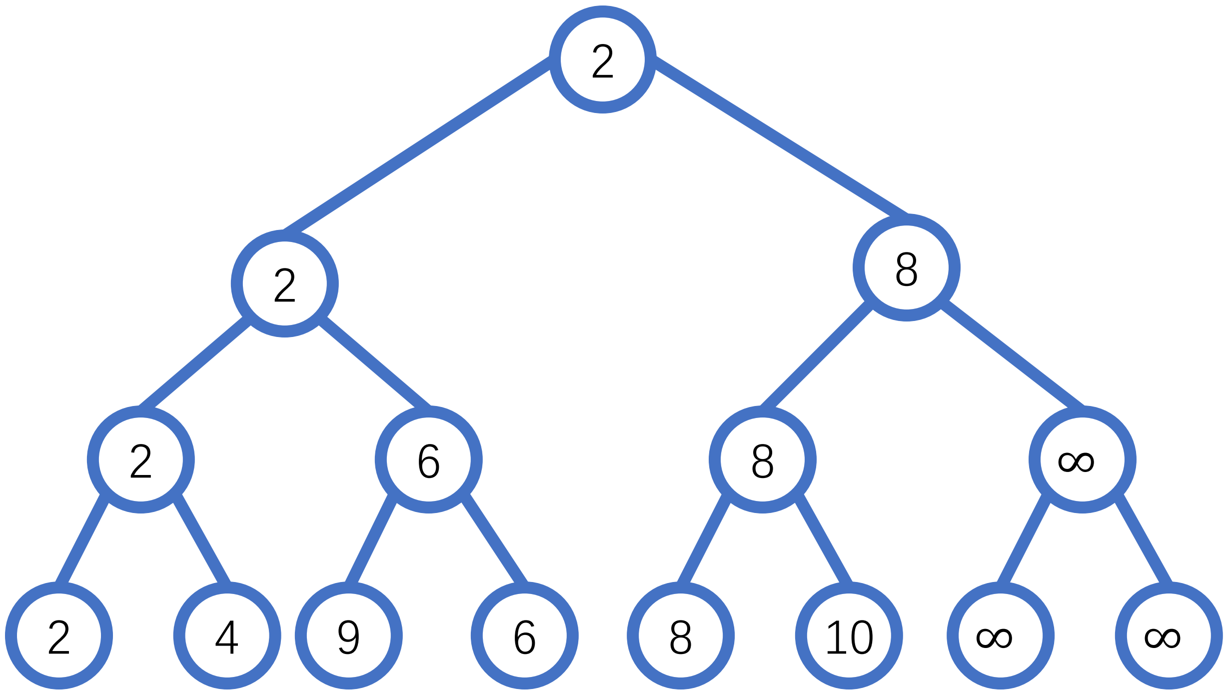
Binary Heap

- **Organize all keys in a complete binary tree**
- **The key at node x is smaller than its children**
- **Once updated, an element needs to move upwards or downwards (heapify)**
- **Not easy to implement**
- **Min/max query applicable only to the entire set**
    - Cannot support min/max of a range



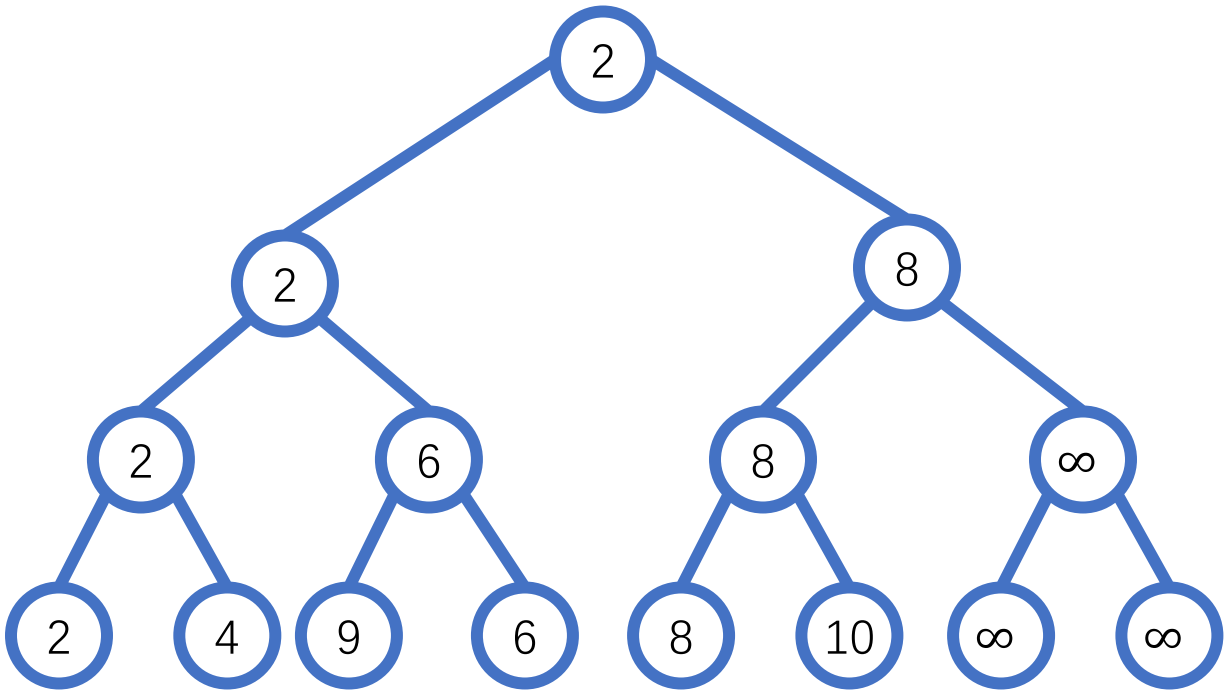
Winning tree

- **Store all elements at the leaves**
- **Each internal node is a competition**
  - the one wins (the smaller one) will be recorded at the node
- **To make it easier to be stored in an array, add some dummy nodes to make the size 2^k for integer k**



Winning tree

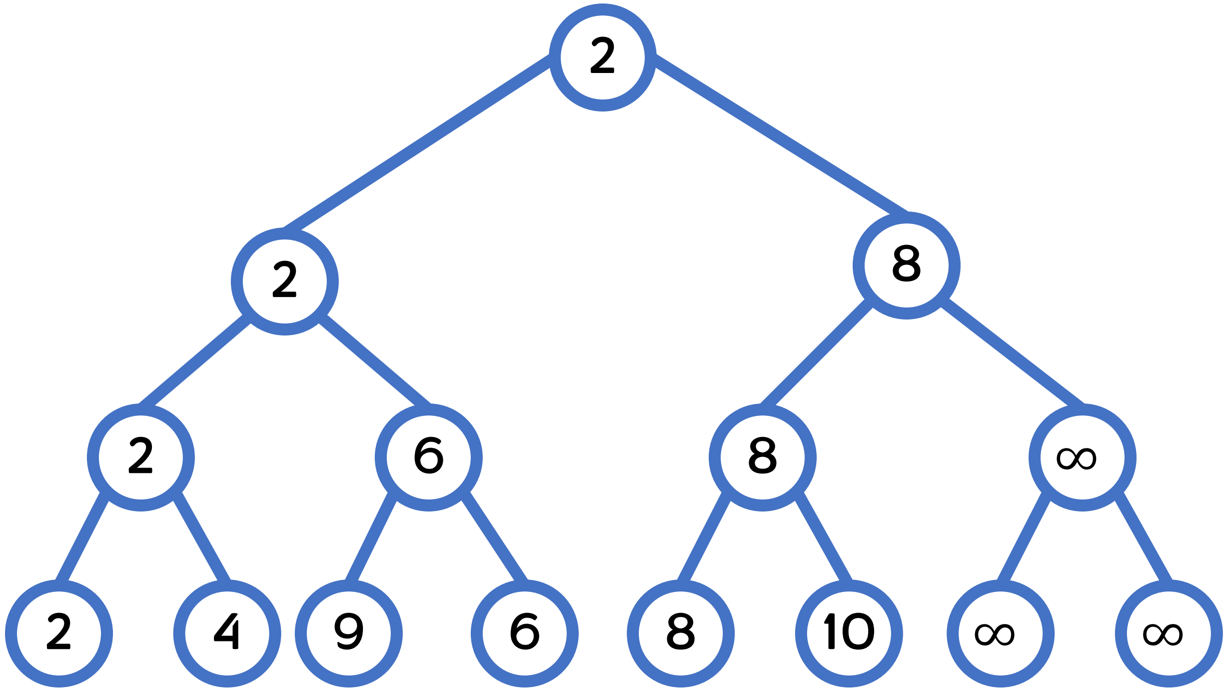
- **Store all elements at the leaves**
- **Each internal node is a competition**
  - the one wins (the smaller one) will be recorded at the node
- **To make it easier to be stored in an array, add some dummy nodes**



- **Insertion**
  - add at the end
  - re-compute all its ancestors
- **Deletion**
  - mark it as ∞
  - re-compute all its ancestors
- **Update**
  - update its key
  - re-compute all its ancestors

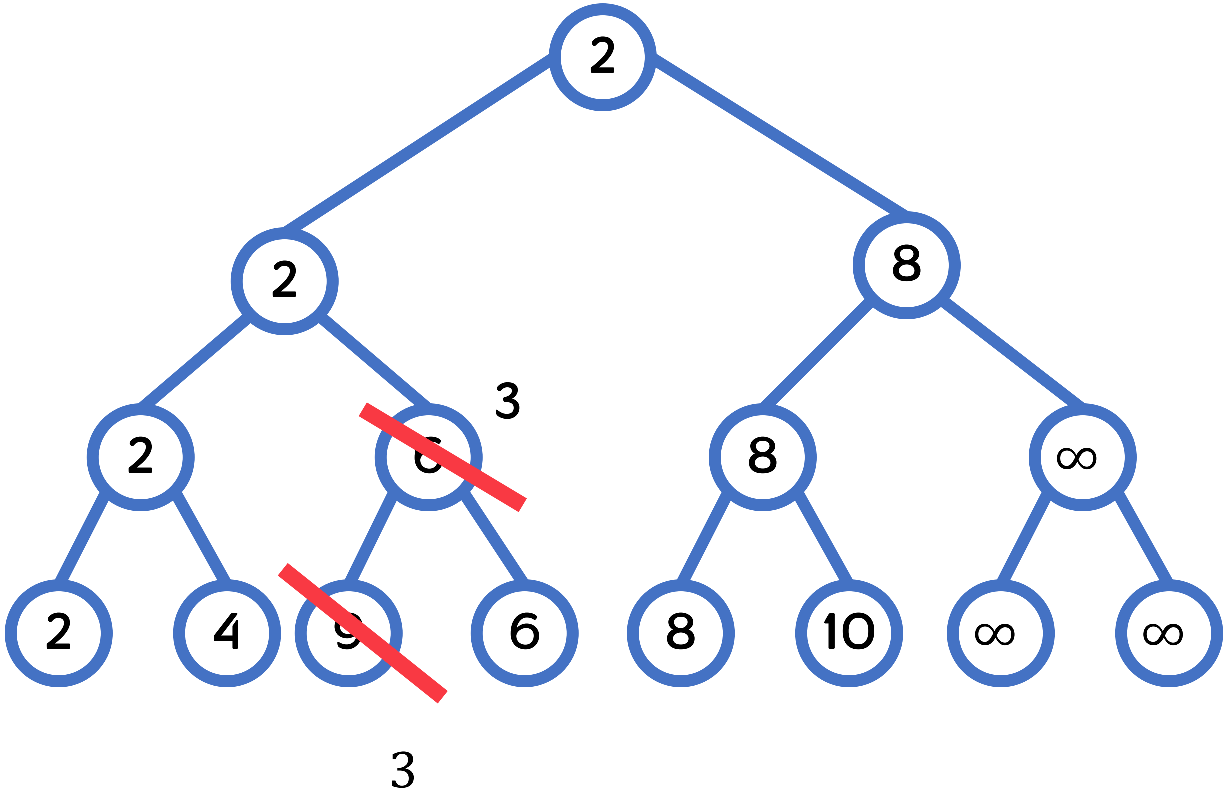
Winning tree – construct

- $O(n)$ time



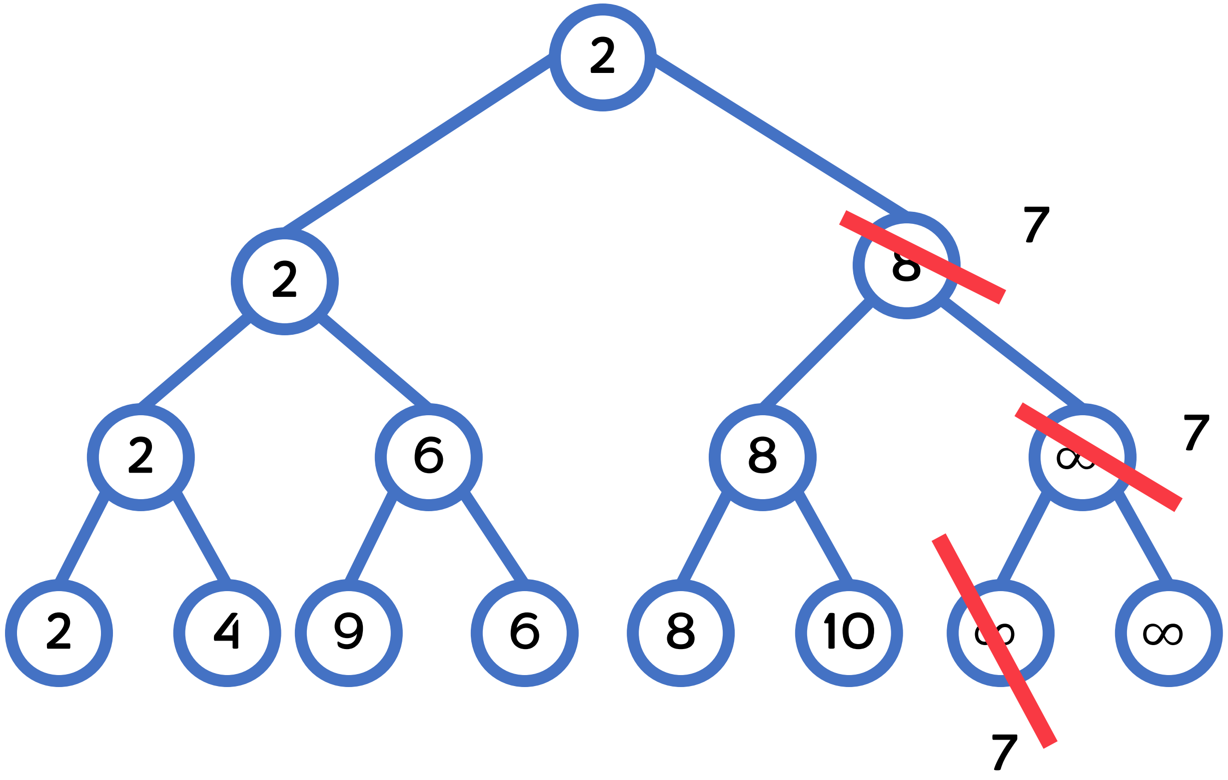
Winning tree – update

- $O(\log n)$ time



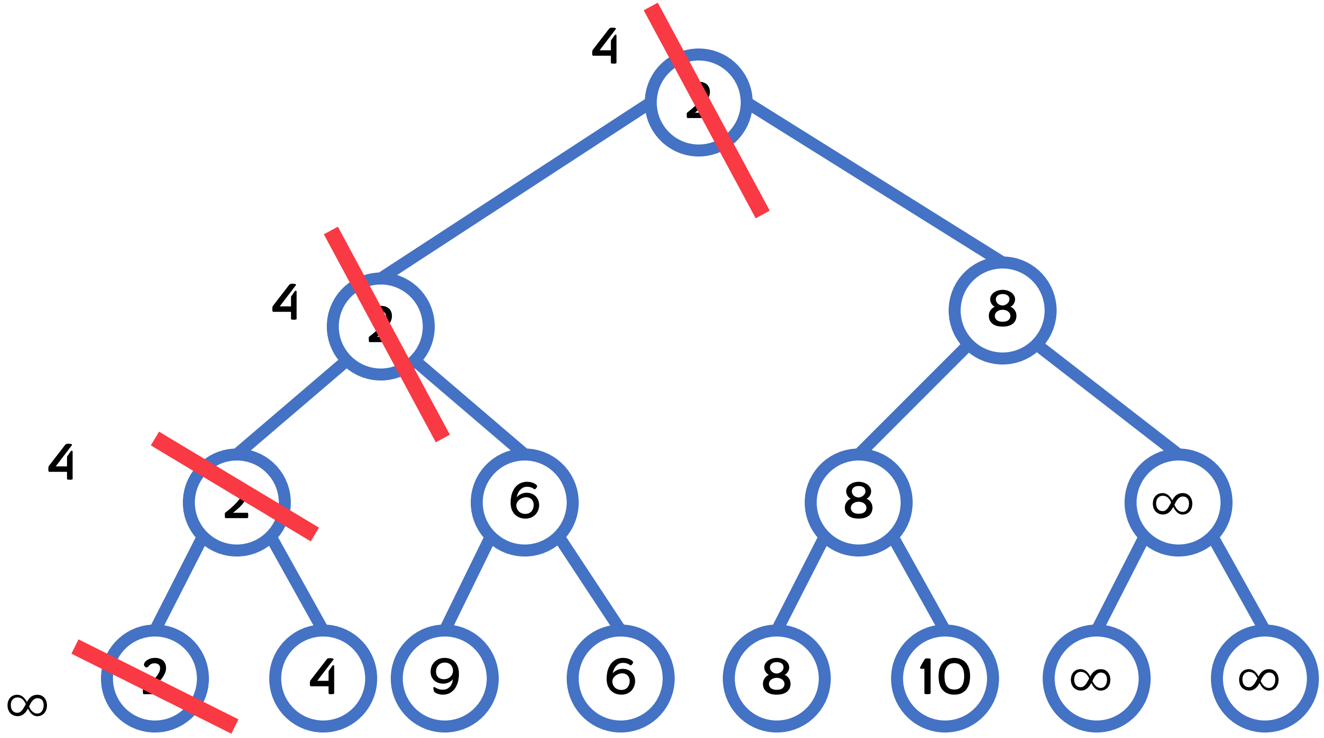
Winning tree: insertion

- $O(\log n)$ time

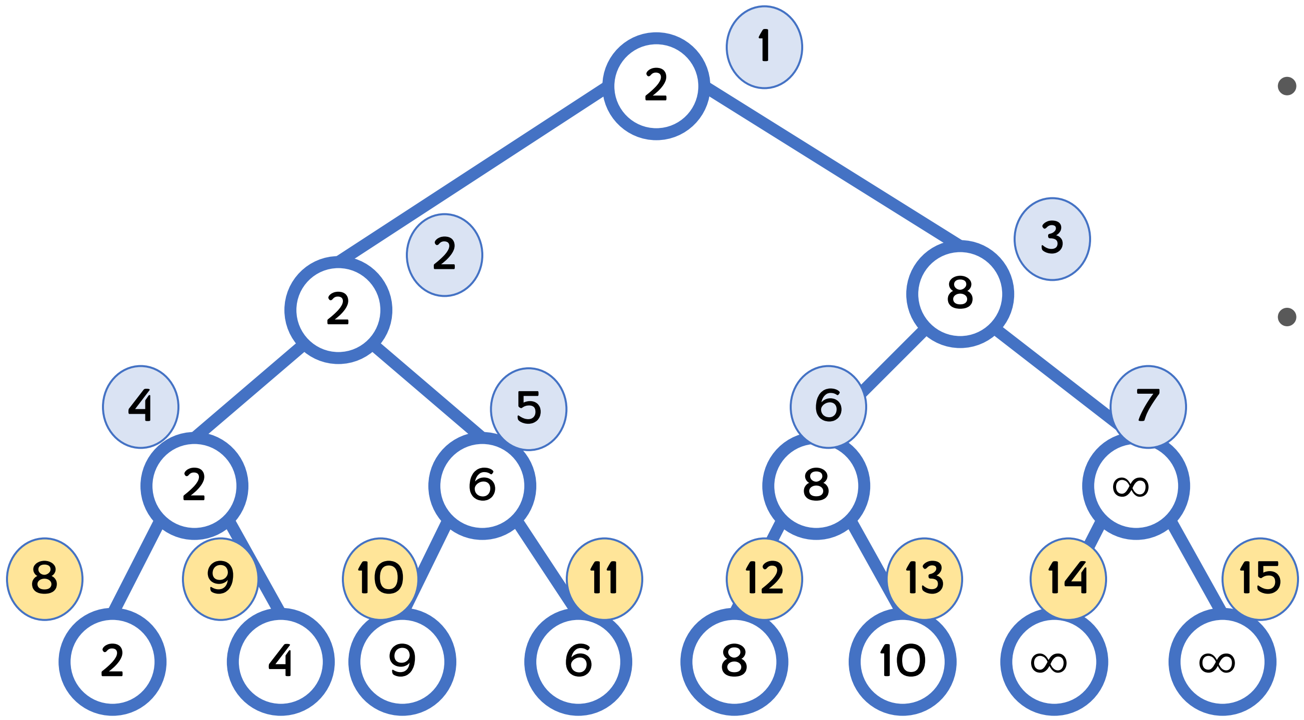


Winning tree - deletion

- $O(\log n)$ time
- If we want to make it more space-efficient, we can move the last element back to the slot of the deleted element



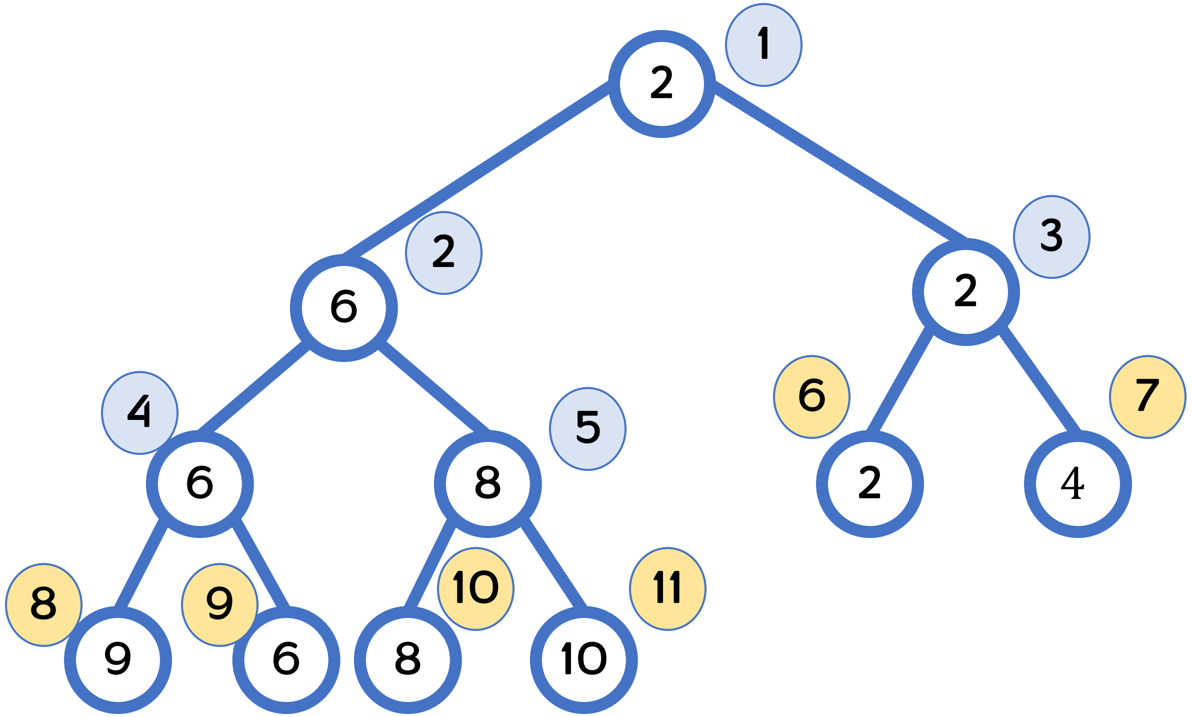
Winning tree



- Can also use an array to store it (similar to binary heap)
- Two children of A[i]: A[2i] and A[2i+1] (can also start from 0)

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	2	2	8	2	6	8	∞	2	4	9	6	8	10	∞	∞

Winning tree



- Can also use an array to store it
- Two children of A[i]: A[2i] and A[2i+1] (can also start from 0)

- We don't need dummy nodes!! Use a **complete binary tree**

Index	1	2	3	4	5	6	7	8	9	10	11
Value	2	6	2	6	8	2	4	9	6	8	10

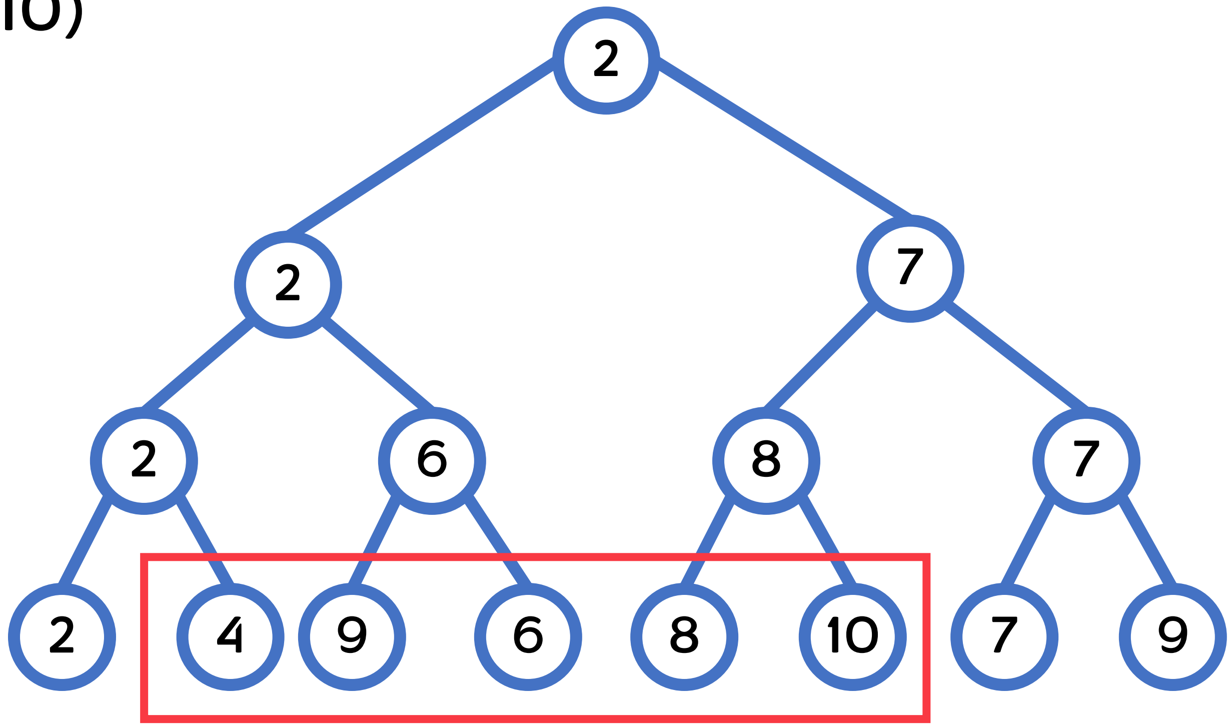
Winning tree vs. binary heap

- **Same asymptotical bound for insertion/deletion/update/extract_min**
- **Winning tree takes more space, and is slightly slower in practice**
  - $O(\log n)$ cost is tight for winning trees
- **Winning tree is much simpler to implement**
  - Essentially, it only needs one operation
- **However, winning tree is more general than binary heap**

Winning tree – range query

- Find the min of the 2nd – 6th elements?

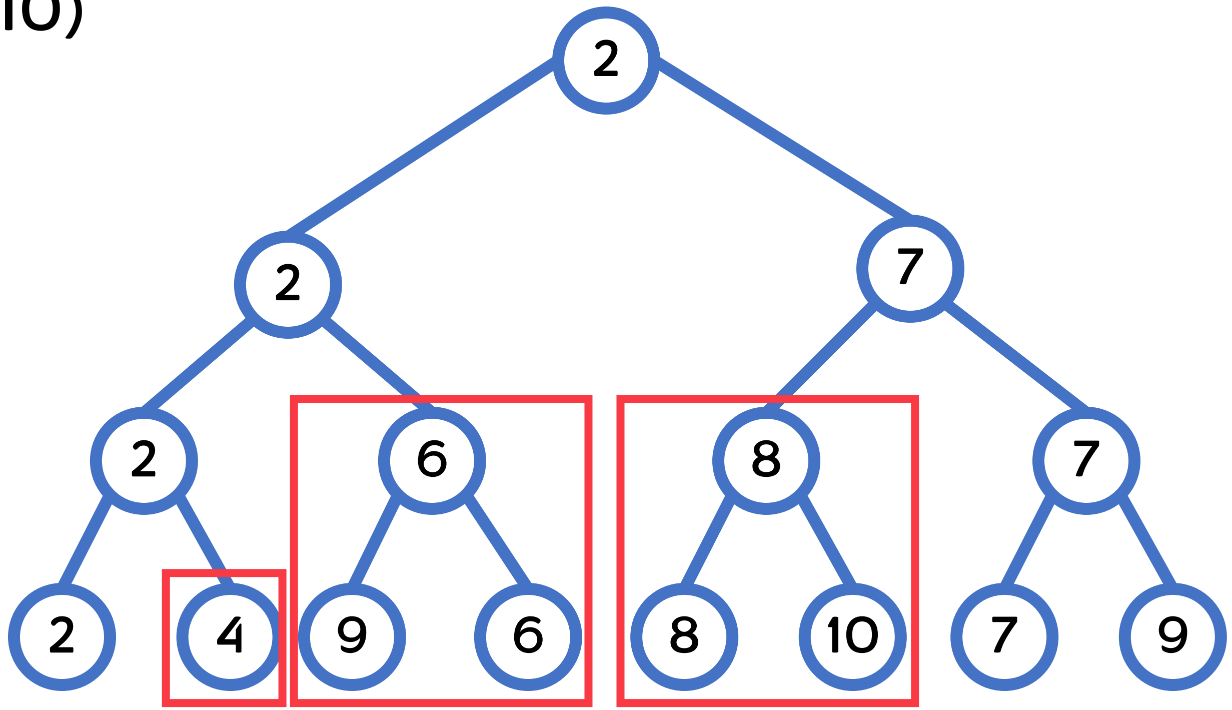
min(4, 9, 6, 8, 10)



Winning tree – range query

- Find the min of the 2nd – 6th elements?

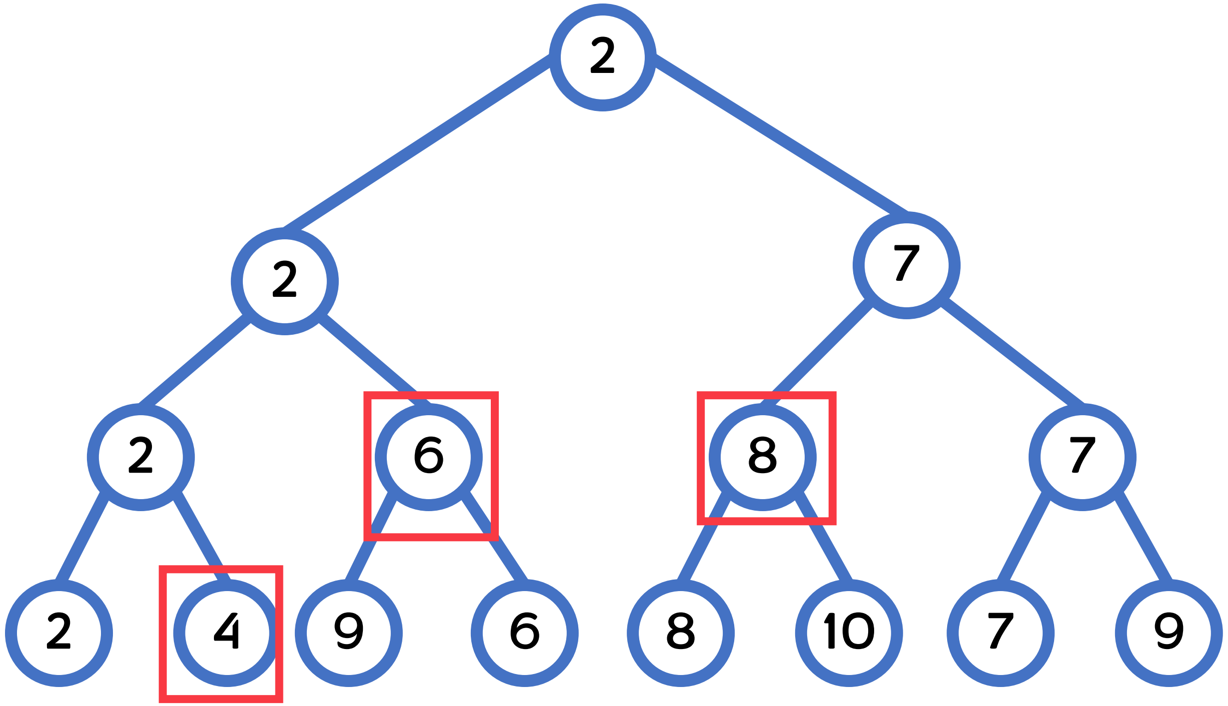
min(4, 9, 6, 8, 10)



Winning tree – range query

- Find the min of the 2nd – 6th elements?

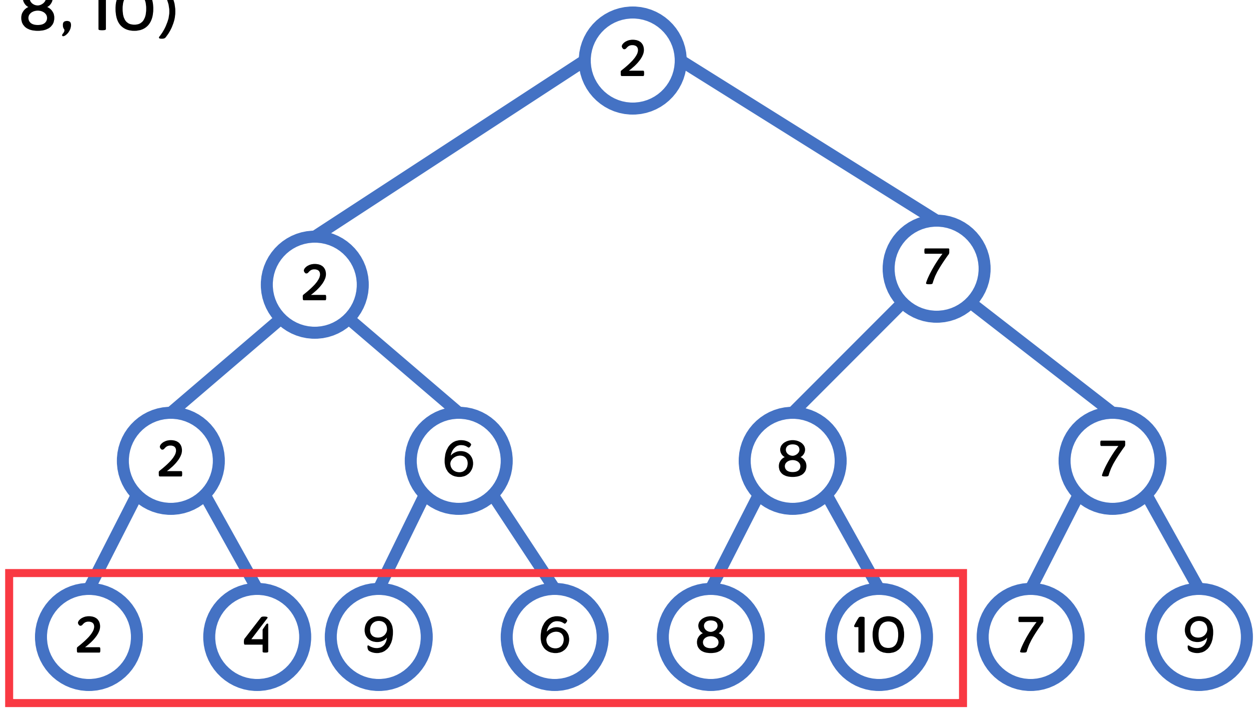
min(4, 6, 8)



Winning tree – range query

- Find the min of the first 6 elements?

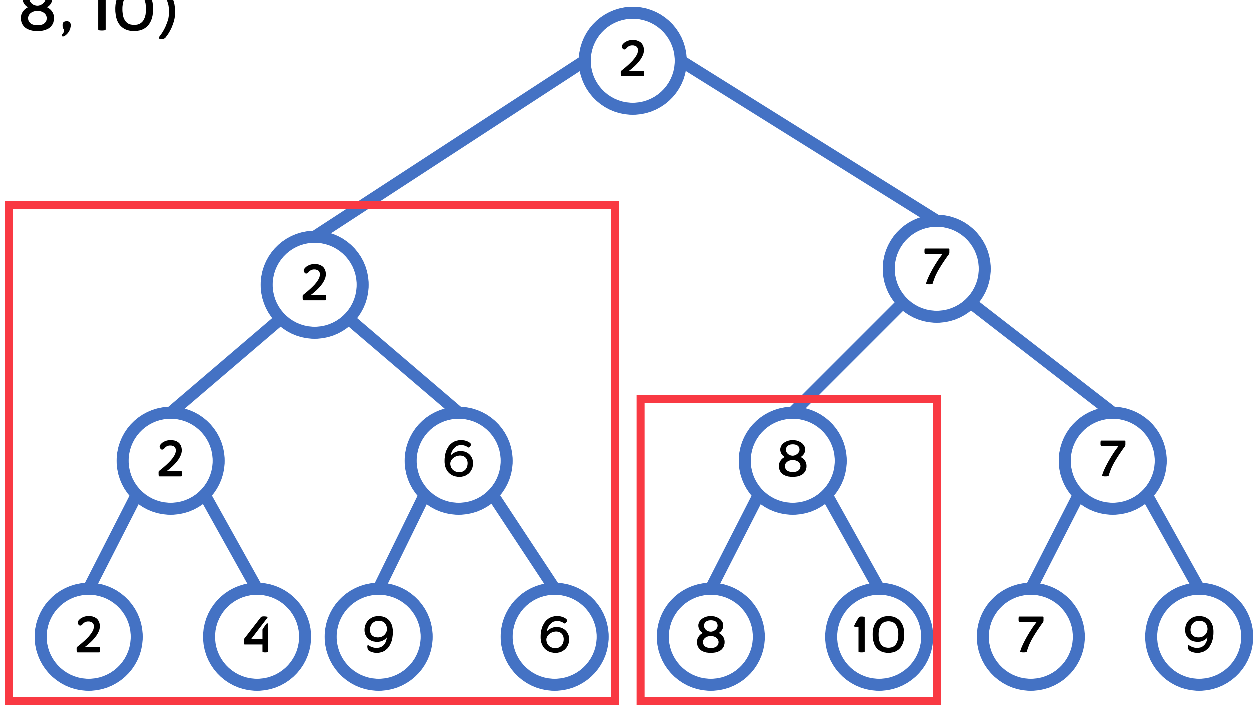
min(2, 4, 9, 6, 8, 10)



Winning tree – range query

- Find the min of the first 6 elements?

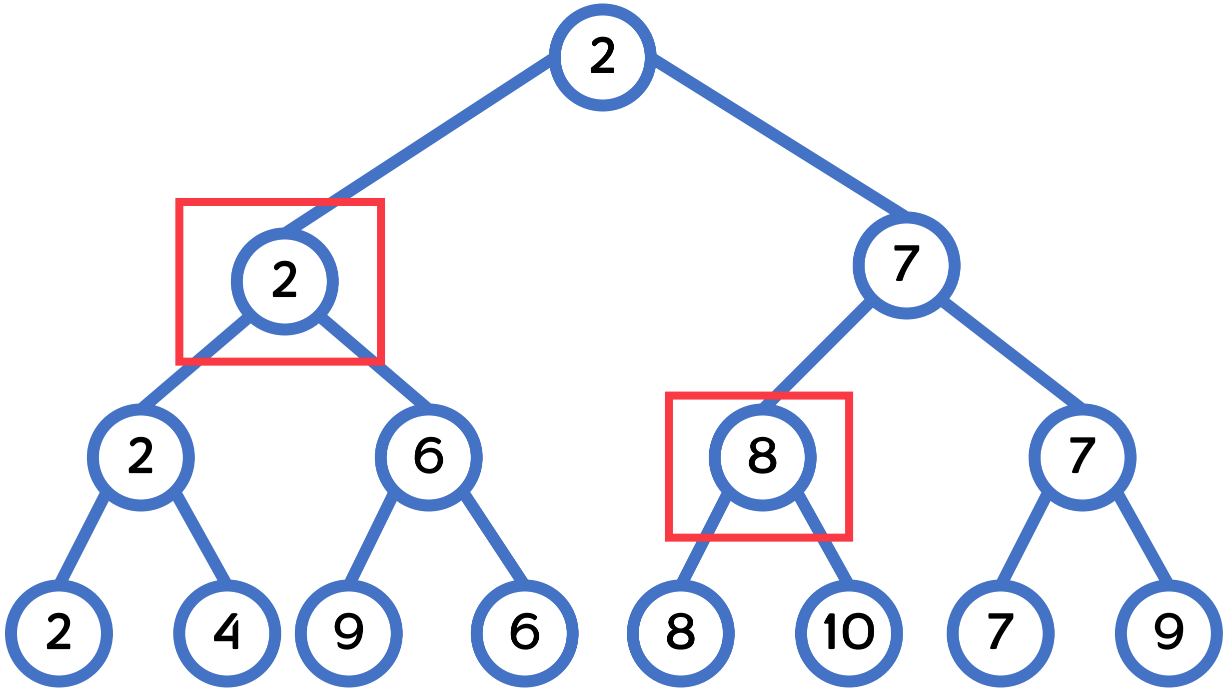
min(2, 4, 9, 6, 8, 10)



Winning tree – range query

- Find the min of the first 6 elements?

min(2, 8)



Winning tree vs. binary heap

- **Same asymptotical bound for insertion/deletion/update/extract_min**
- **Winning tree takes more space, and is slightly slower in practice**
  - $O(\log n)$ cost is tight for winning trees
- **Winning tree is much simpler to implement**
  - Essentially, it only needs one operation

Augmented search trees

Range sum query

- Given a set of key-value pairs, a query asks for the sum of values in between a key range

25,2 16,10 12,11 15,17 7,7 9,2

8,9 4,4 17,3 13,9 10,2

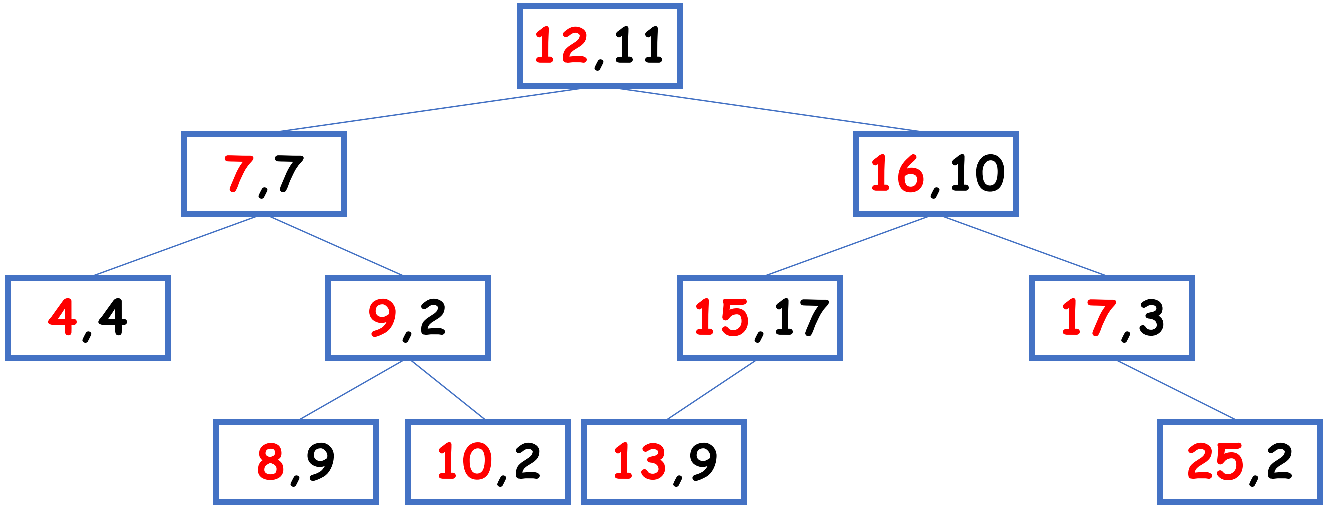
Key	4	7	8	9	10	12	13	15	16	17	25
value	4	7	9	2	2	11	9	17	10	3	2

Prefix sum	4	11	20	22	24	35	44	61	71	74	76

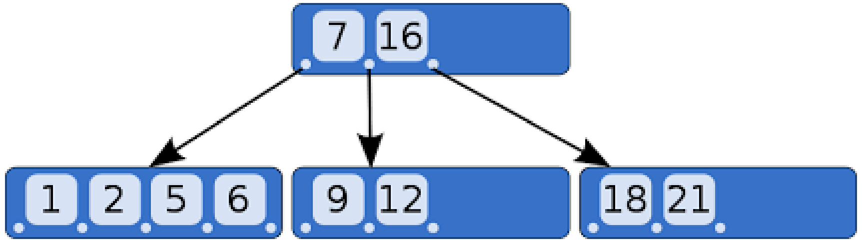
- **range_sum(7,16) = 71 – 4 = 67**
- $O(\log n)$ time

Range max query

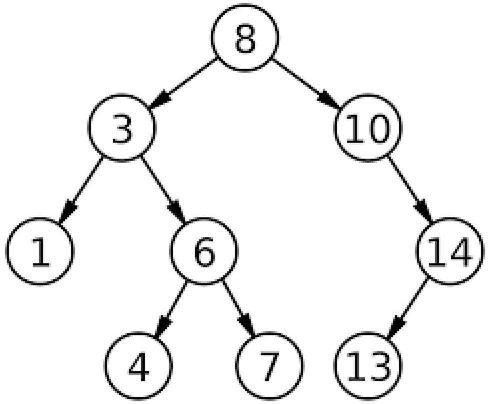
- **Given a set of key-value pairs, a query asks for the max of values in between a key range**
  - Cannot use subtraction!
- **The set of key-value pairs can be updated**
  - Insert/delete new key-values or update the values
- **Search tree**
  - Insertion/deletion/update in $O(\log n)$ time



Why we need search trees?



A B-tree structure maintaining ordering on keys.



A binary search tree structure maintaining ordering on keys.

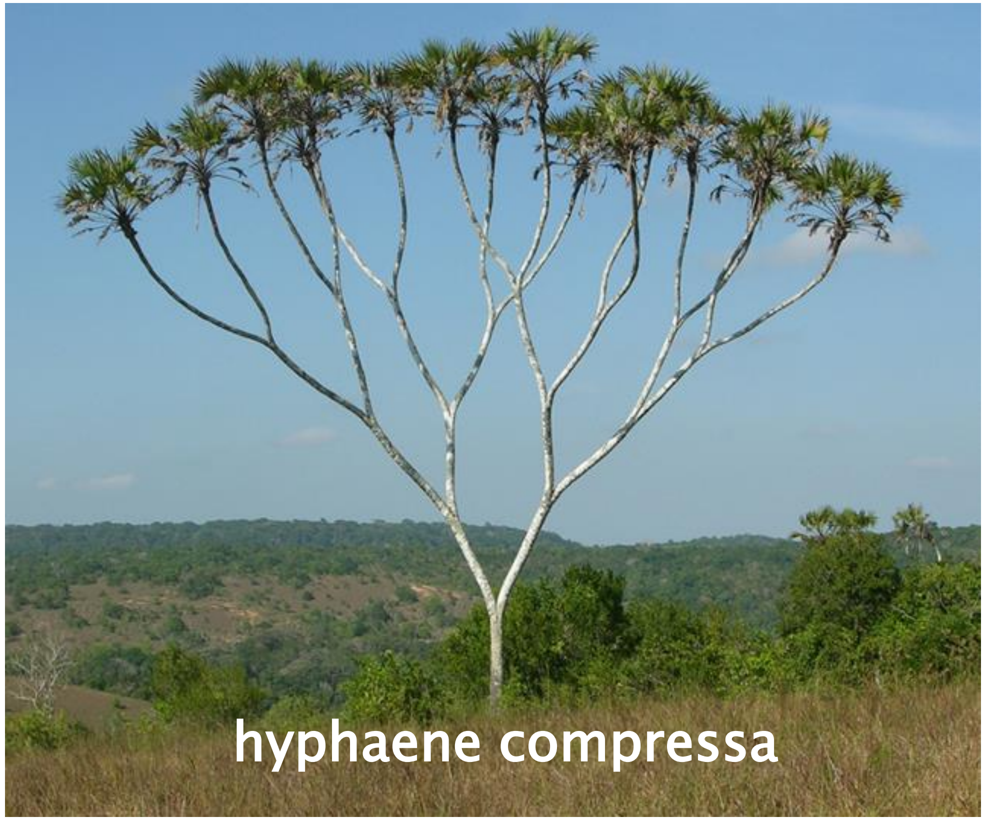
- **Organizing a set of data**
- **What is the benefit/disadvantage of using trees compared to arrays?**
- **What is the benefit/disadvantage of using trees compared to hash table?**

Search Trees

- **Ordered!**
  - The in-order traversal is sorted w.r.t. keys
  - (cannot be achieved if we use hash tables)
- **Dynamic!**
  - Insertion/deletion in O(log n) time
  - (cannot be achieved if we use ordered arrays)
- **Efficient!**
  - Insertion/deletion/lookup in O(log n) time
  - Traversal in O(n) time
  - Get some info from / operate on the root of subtrees (do not need to touch all tree nodes)
- **However?**
  - More space than flat arrays
  - Worse locality since tree nodes are scattered

Balanced Binary Trees

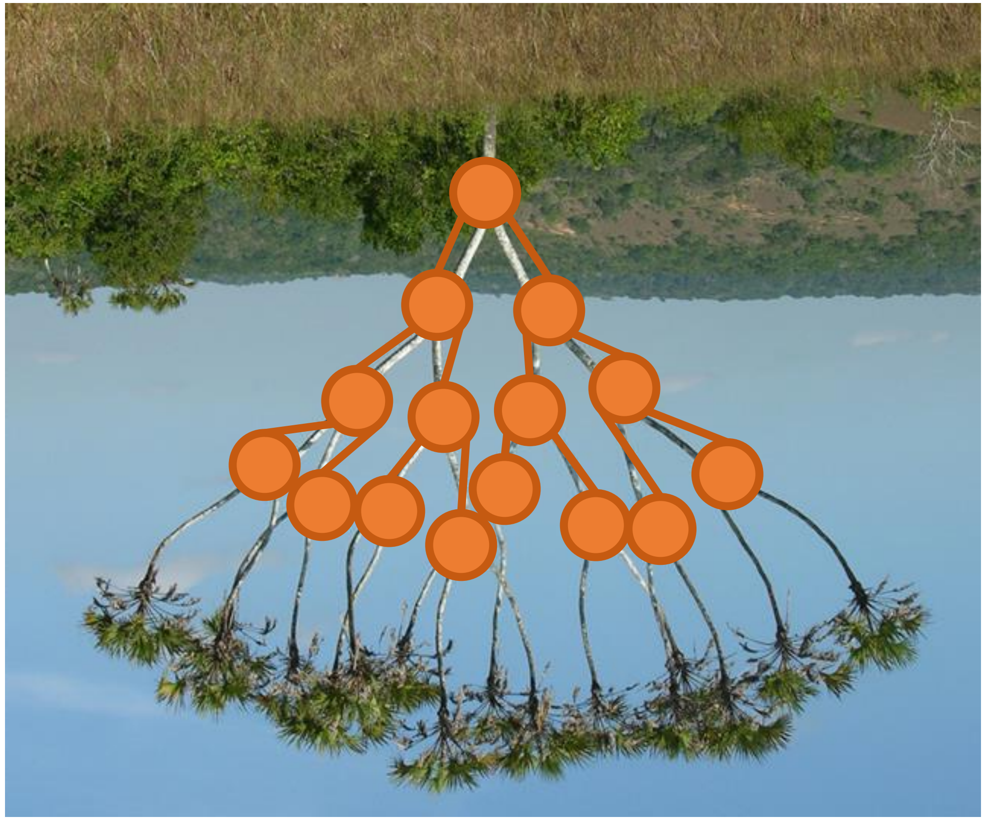
- **Binary: each tree node has at most two children**
- **Balanced: the tree has bounded height**
  - Usually $O(\log n)$ for size n



A wild balanced binary tree

Balanced Binary Trees

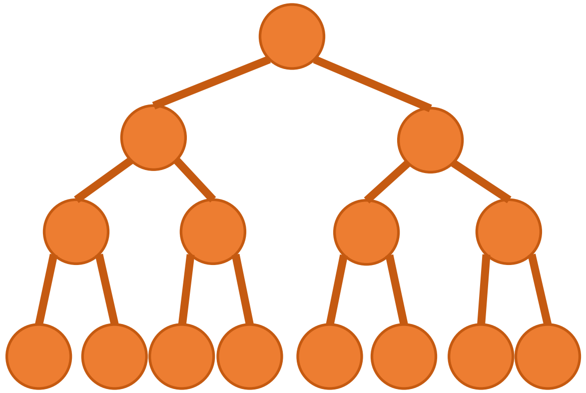
- Binary: each tree node has at most two children
- Balanced: the tree has bounded height
  - Usually $O(\log n)$ for size n



A wild balanced binary tree

Balanced Binary Trees

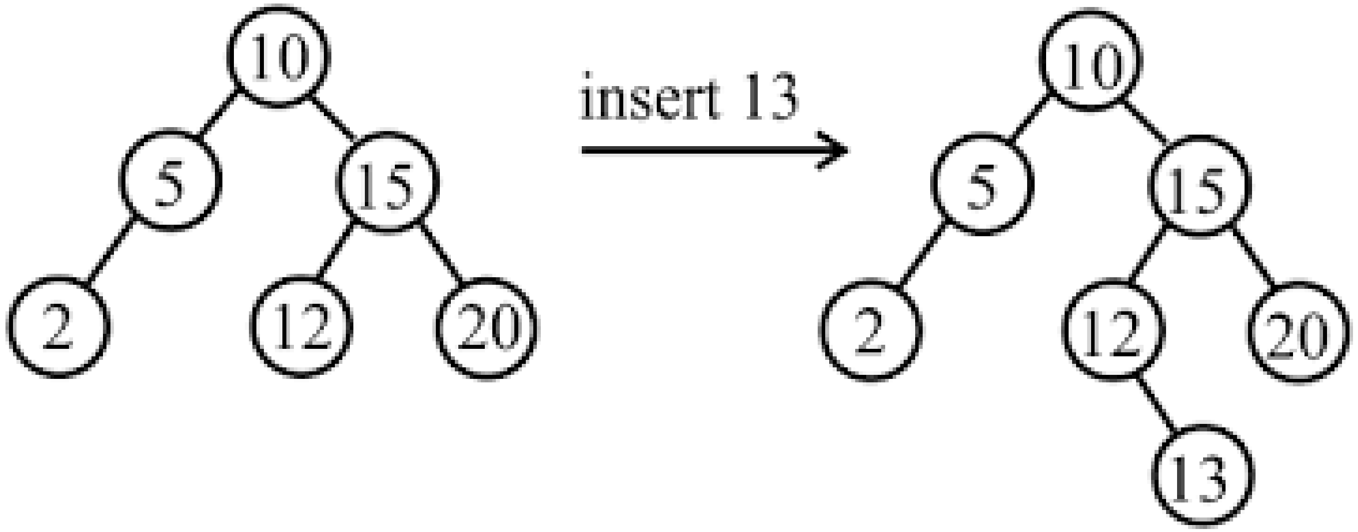
- **Binary: each tree node has at most two children**
- **Balanced: the tree has bounded height**
  - Usually $O(\log n)$ for size n



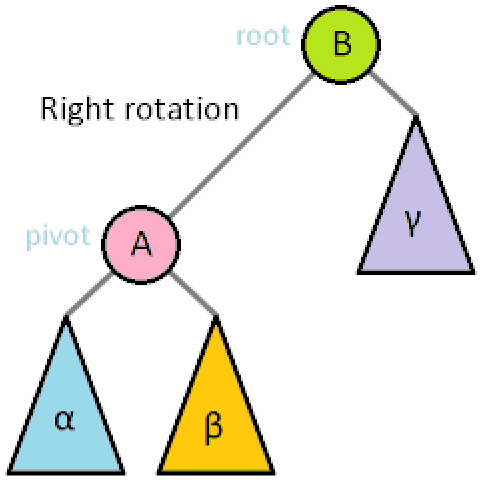
An abstract balanced binary tree

Binary Search Trees

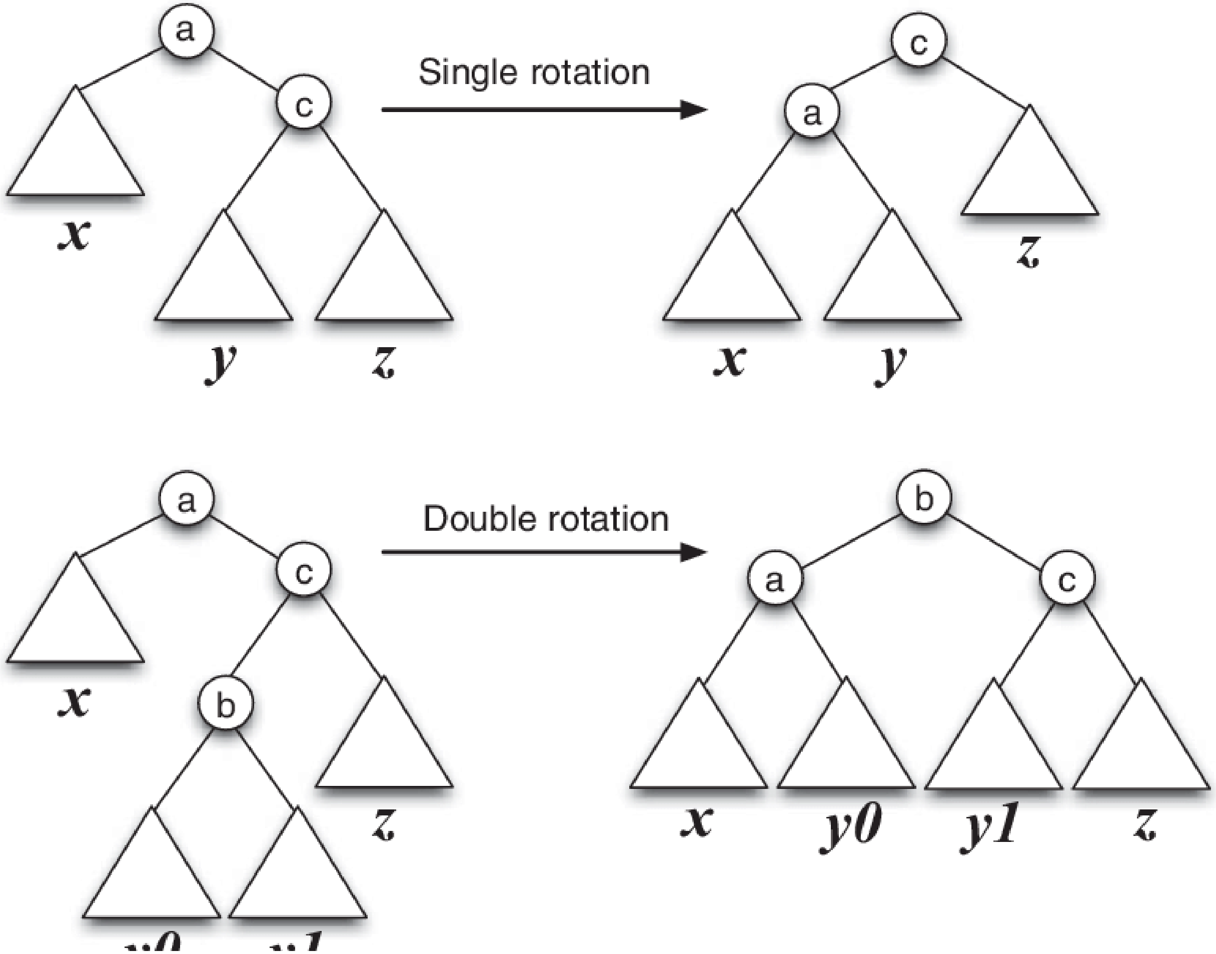
- Lookup/insertion just follows the path
- Deletion may be more involved
- Need to rebalance the tree using rotation!



Tree rotation



Source: wikipedia



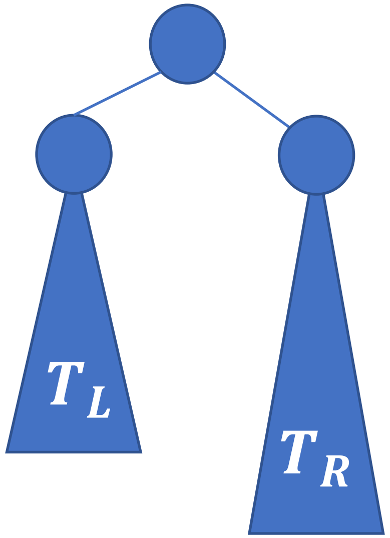
Source: https://www.researchgate.net/figure/A-single-left-rotation-and-a-double-left-rotation-a-b-and-c-are-elements-x-y-y0_fig4_220676747

AVL trees

- **Invariant: for any node in the tree, the heights of its two subtrees differ by at most 1**

- **Height is bounded by O(log n)**

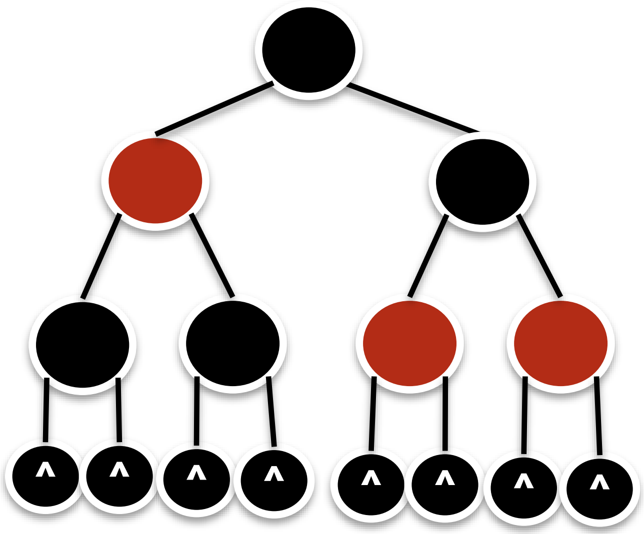
- $|\boldsymbol{h}(\boldsymbol{T_l}) - \boldsymbol{h}(\boldsymbol{T_R})| \leq \mathbf{1}$
  - $h(\cdot)$ denotes the height of the tree



Red-black trees

- **Invariants:**
  - Red node only have black children;
  - External nodes (leaves) are black;
  - The black height from the root to any leaf is the same.

- **Tree height at most $O(\log n)$**



Augmented tree

Augmented data structures

- **Very rarely we need to design a brand-new data structure**

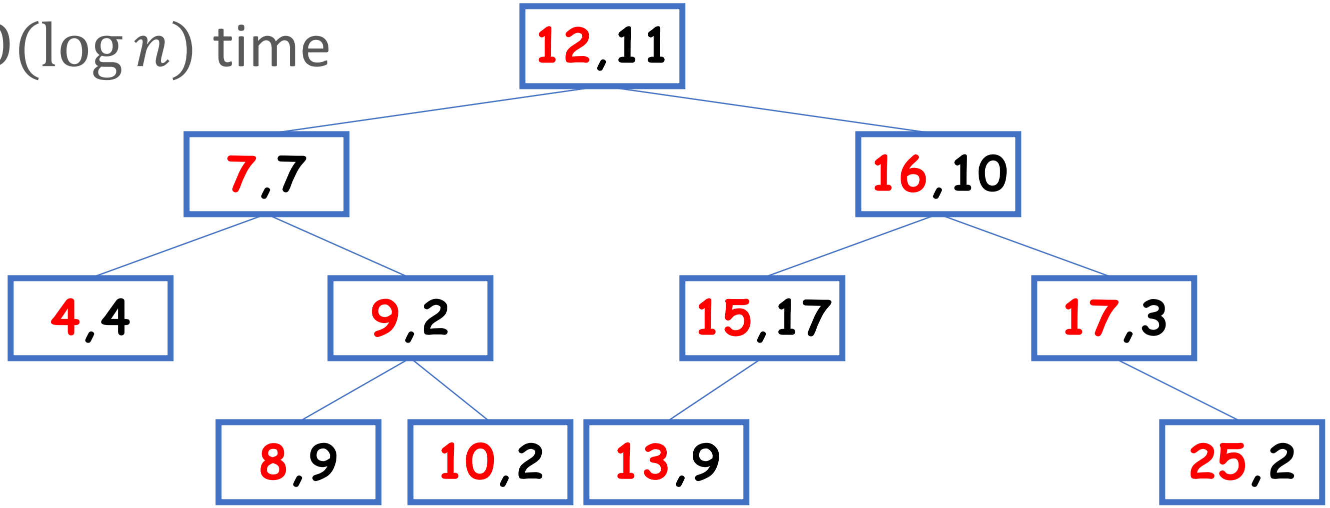
- **We could augment some of the existing ones!**
  - Directly use existing operations/analysis
  - With minimal additional information to be maintained

Range sum query

- **Given a set of key-value pairs, a query asks for the sum of values in between a key range**

Key	4	7	8	9	10	12	13	15	16	17	25
value	4	7	9	2	2	11	9	17	10	3	2

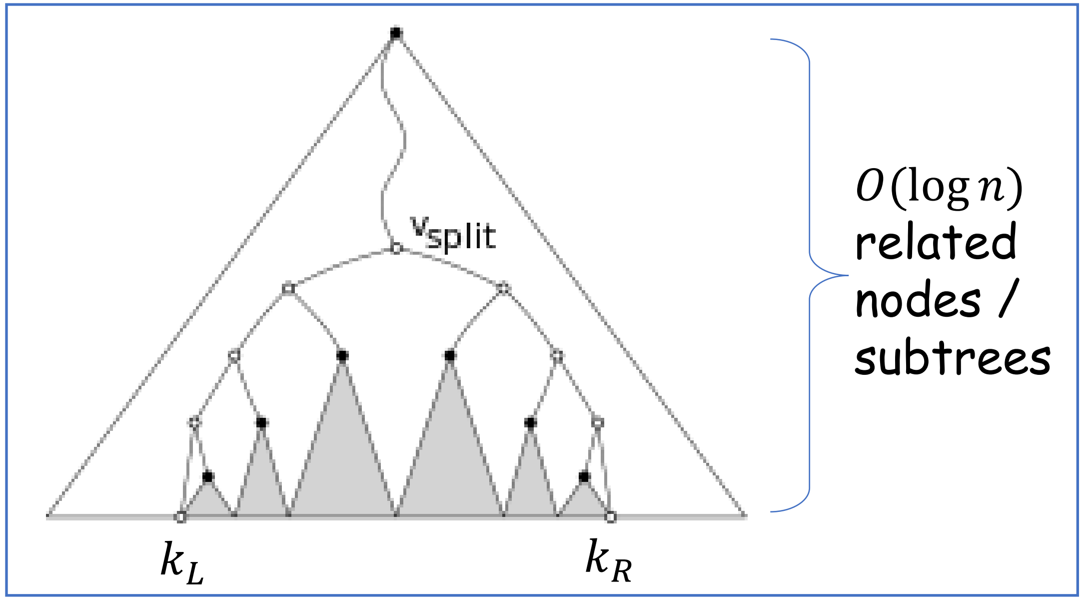
- **Search tree**
  - Insertion/deletion/update in $O(\log n)$ time



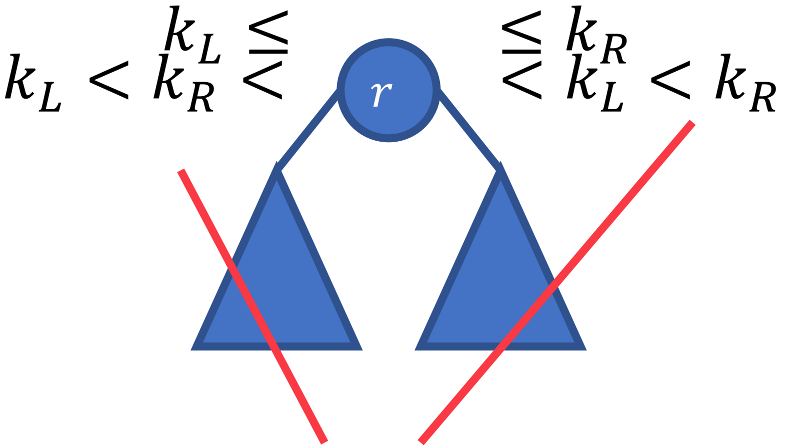
First of all, how to find all records in a given key range on a binary search tree?

Range query algorithm

- **Report all entries in key range $[k_L, k_R]$.**

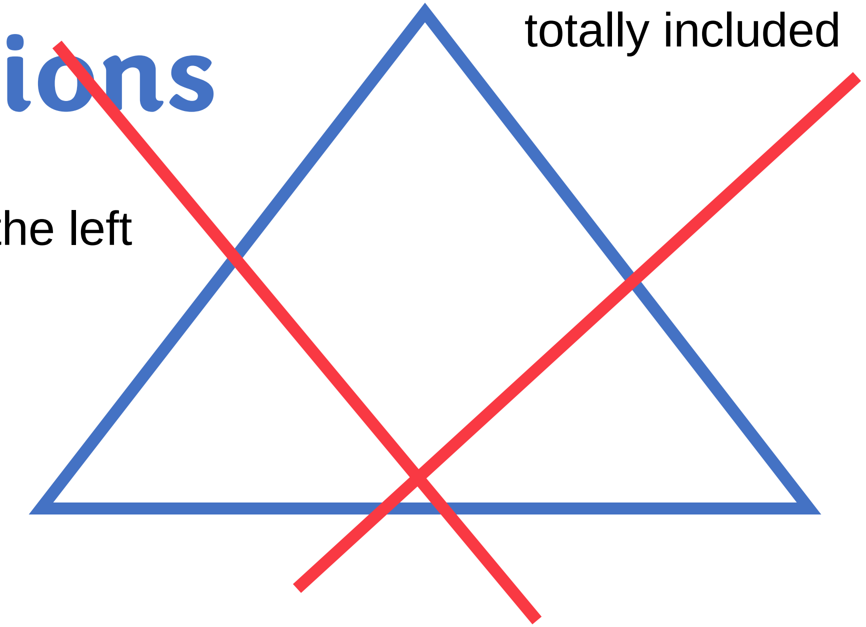


range(t, k_L, k_R) {
  r = t.root; if (!r) return;
  if ($k_R < r$) return range(t.left, k_L, k_R); // total range in the left branch
  if ($k_L > r$) return range(t.right, k_L, k_R); // total range in the right branch
   add rangeR(t.left, k_L) to result
   add t.root to result
   add rangeL(t.right, k_R) to result
}

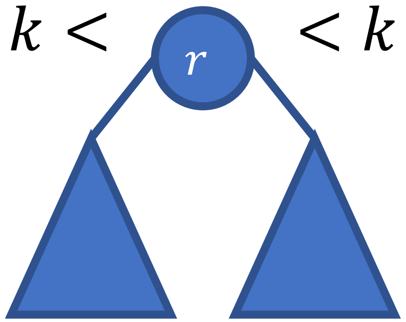


rangeL and rangeR functions

- **rangeL: everything $\leq k$**



rangeL(t, k) {
  r = t.root; if (!r) return 0;
  if ($k < r$.key) return rangeL(t.left, k); // total range in the left branch
  if ($k \geq r$.key) {
    add t.left to result //the whole left subtree should be included
    add t.root to result
    if ($k > r$.key) add rangeL(t.right, k) to result
  }
}



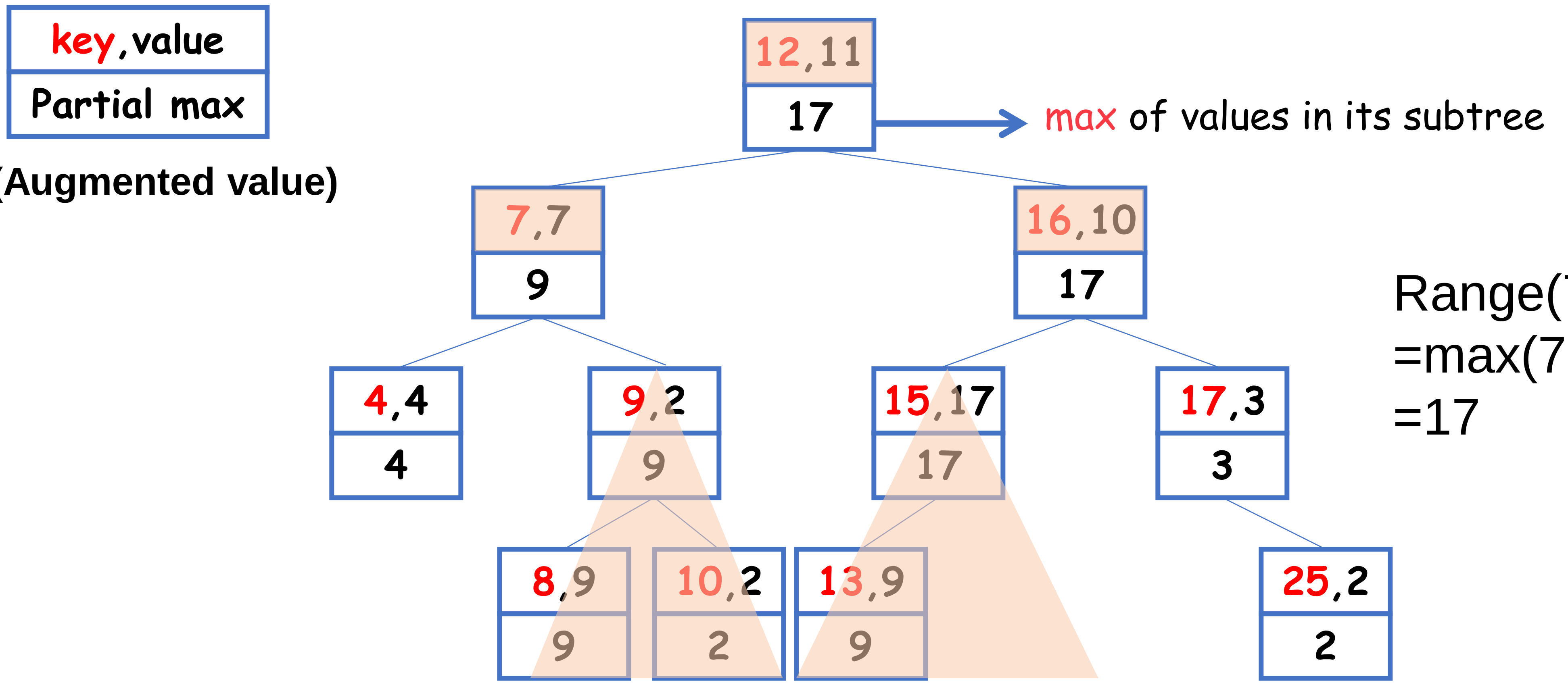
Range query: time complexity

- $O(\log n)$ time to find the relevant subtrees
- $O(k)$ time to traverse and output (k is the output size)
- Total cost: $\mathrm{O}(\log n + k)$

What if we only need the sum/max/count of the range?

Augmented Trees for Range Max

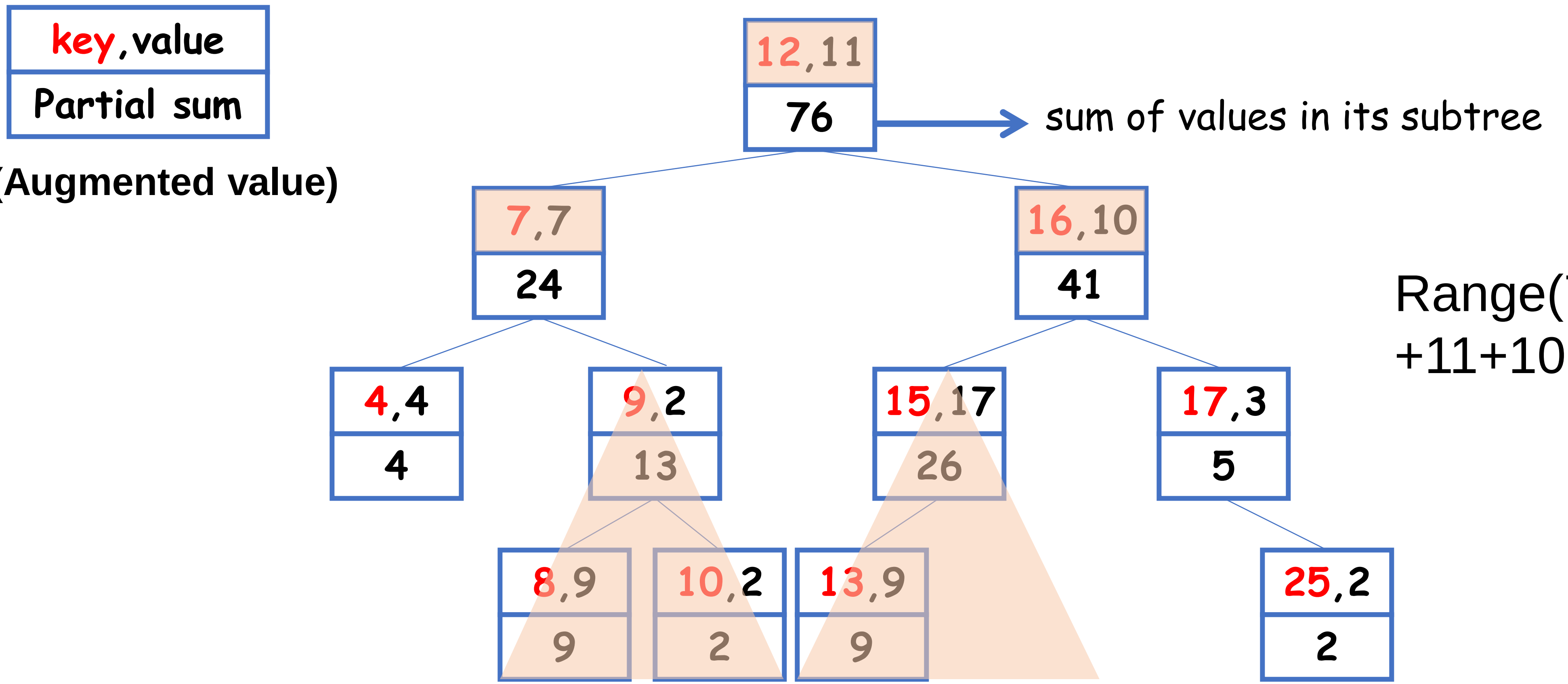
- Storing the max in each tree node for answering range sum queries



Range(7,16)
=max(7,9,11,10,17)
=17

Augmented Trees for Range Sum

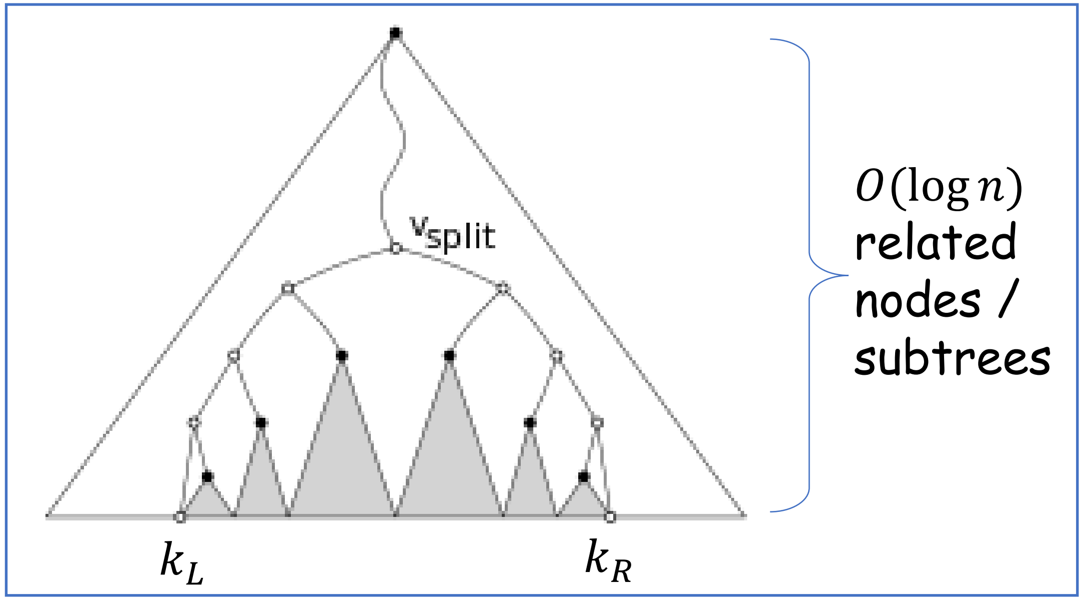
- Storing the sum in each tree node for answering range sum queries



Range(7,16)=7+13
+11+10+26=67

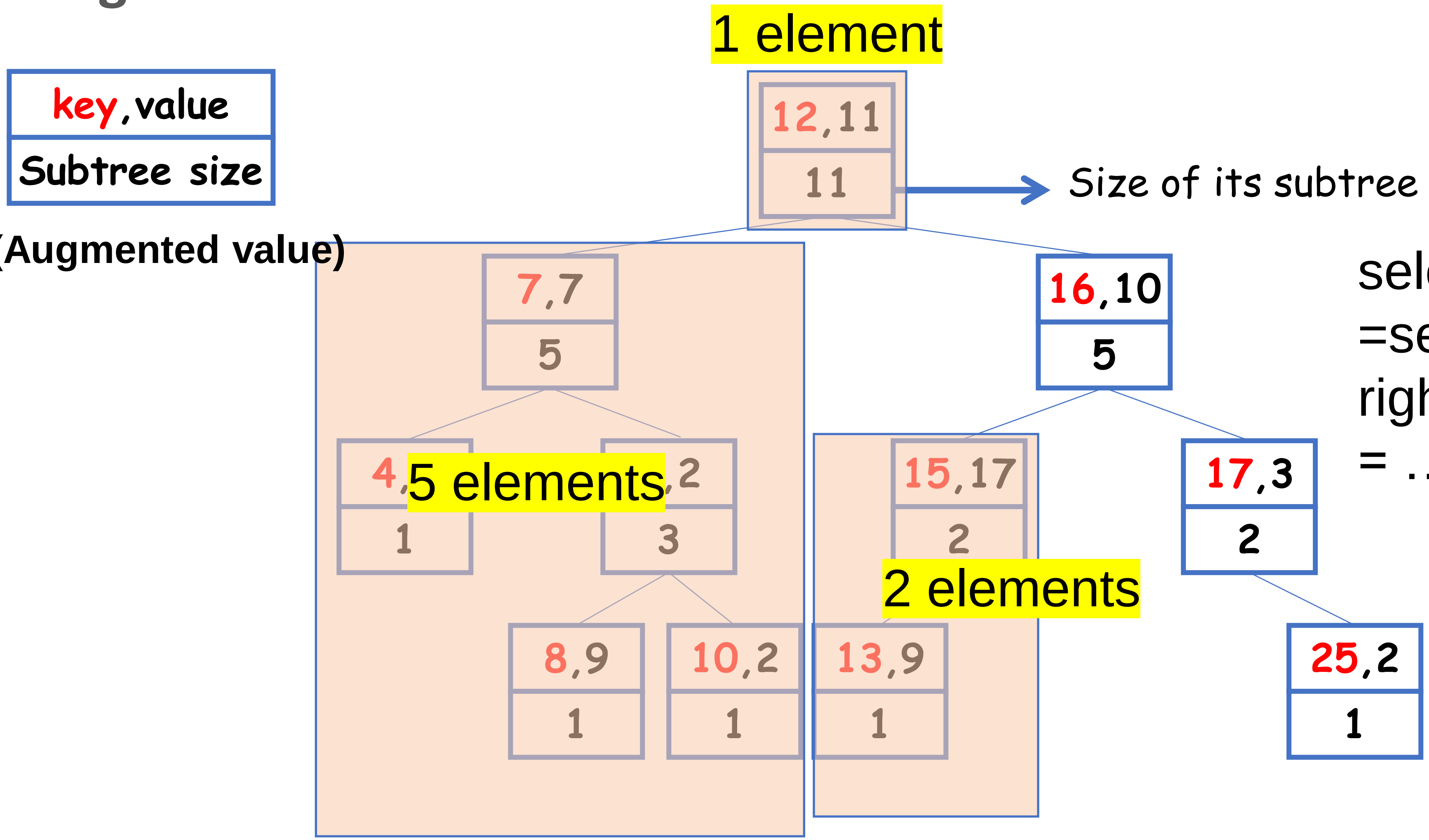
Range max query algorithm

- **When the whole subtree is included in the result, sometimes we don't need to traverse the tree – use the augmented value!**
- **E.g., if we want the max of values**
  - Store at each node the max value
  - When we say "add the subtree to the result", directly read the max value (augmented value) from the subtree root, and add the value to the result
  - Takes only $O(\log n)$ time!



Augmented Trees for rank report / select k-th element

- Storing the subtree size



select(7)
=select(7-5-1) from right subtree
= …=(13,9)

rank(16)
=5+1+2+1=9

Maintain augmented values

- **In any construction/insertion/deletion/update … algorithms**
- **Any modification will update all related tree nodes on the path**
- **Asymptotically the same time bound**
- **aug_val of a node can be computed by its two children and the node:**
  - min = min(leftmin, rightmin, rootvalue)
  - sum = leftsum+rightsum+rootvalue
  - size = leftsize+rightsize+1

Augmented tree for range queries

- **Range sum/min/max/… queries**
- **Can be combined with any searched tree data structure**
  - Base data structure can be AVL, red-black tree, etc.

- **Using different augmentations we can get different functionalities**

Revisit of the range sum/max/rank query

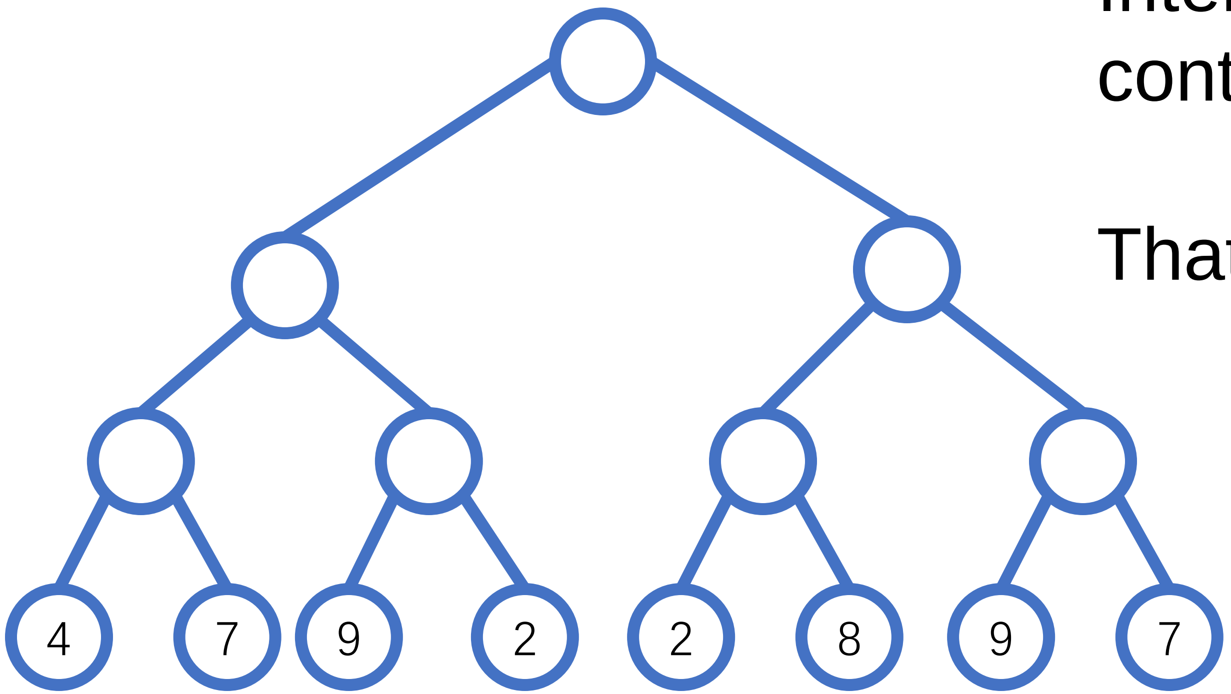
- Given a set of key-value pairs, a query asks for the sum of values in between a key range

Key	4	7	8	9	10	12	13	15	16	17	25
value	4	7	9	2	2	8	9	7	10	3	2

Prefix sum	4	11	20	22	24	32	41	48	58	61	63

- **range_sum(7,13) = 41 – 4 = 37**
- $O(\log n)$ time

Build a winning tree for the query!

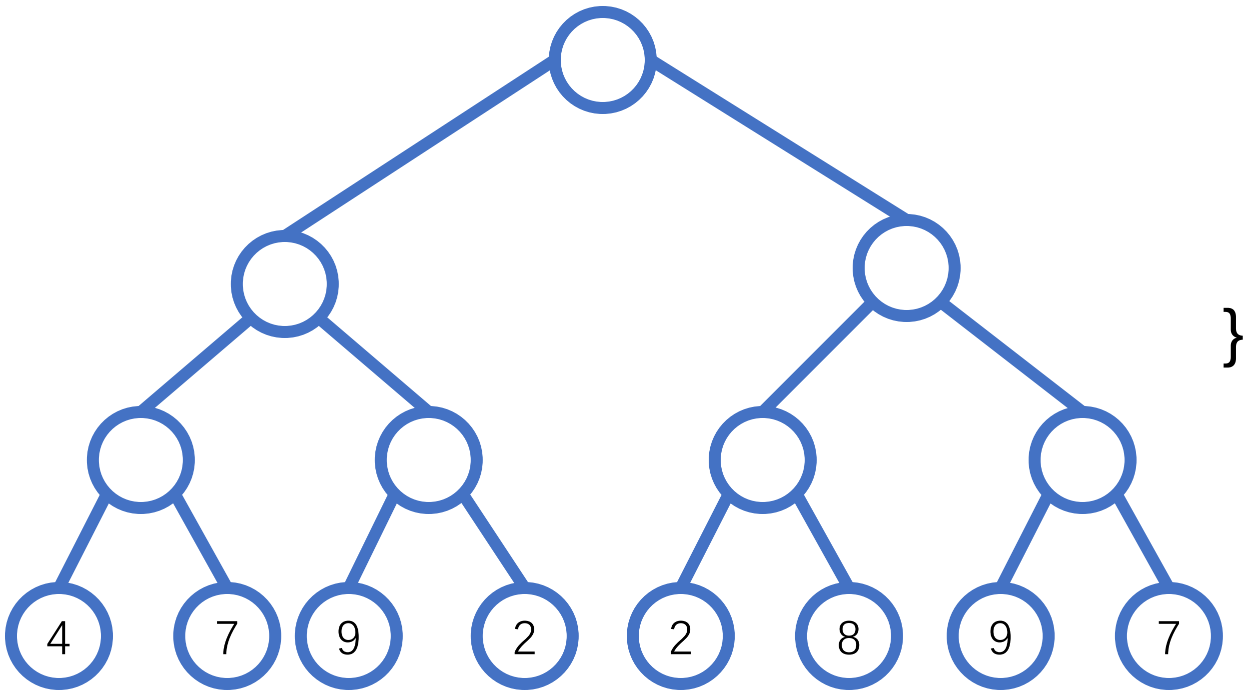


Key	4	7	8	9	10	12	13	15	16	17	25
value	4	7	9	2	2	8	9	7	10	3	2

Interior node does not contain an element!

That make things easier!

Range query on a winning tree



rangeMax(t, k_L, k_R) {
  r = t.root; if (r is null) return;
  if ($t.\min < k_R$ and $t.\max < k_L$): return;
  if ($k_L \leq t.\min$ and $t.\max \leq k_R$):
    ans = max(ans, $t.\mathrm{augval}$) and return;
  ~~if ($k_L \leq r \leq k_R$): ans = max(ans, r);~~
  rangeMax($t.\mathrm{left}, k_L, k_R$);
  rangeMax($t.\mathrm{right}, k_L, k_R$);
}

Winning tree itself is a special augmented tree

- It can be used directly as a priority queue because the extract-Min for priority queue is a special range query
- We can augment the winning tree differently for different queries; we can augment multiple fields simultaneously
- The left-to-right order of the leaves in a winning tree can be used to represent a list, and augmenting the winning tree can solve all list query problems
- If we pre-sort all elements or the key range is fixed (e.g., $[1, \dots, n]$), winning tree can be used as a static search tree **(discretization)**

Summary

Winning tree + augmentation is almost sufficient for any data structure to maintain 1D data

- **Winning tree stores the elements in all leaves in a complete binary tree**
  - Unordered: binary heap
  - A list: list queries (static)
  - A sorted list: static binary search trees

- **Augmentation supports range and rank queries (no matter what tree that is)**
  - For the whole list: stored in the root
  - For a range (k_L, k_R): use the algorithm that visit $O(\log n)$ subtrees
  - Supports min/max/sum/rank/…
  - For list-all, the cost is $O(\log n + k)$ where k is the output size

The next lectures …

- Dynamic programming!
- (For the next four lectures)

- Training part of Homework 2 due tomorrow