Theory exam 3

Review session

Mariam Vardishvili

This slide is intentionally left blank

Updated slide numbers: -

Number of updates: 0;

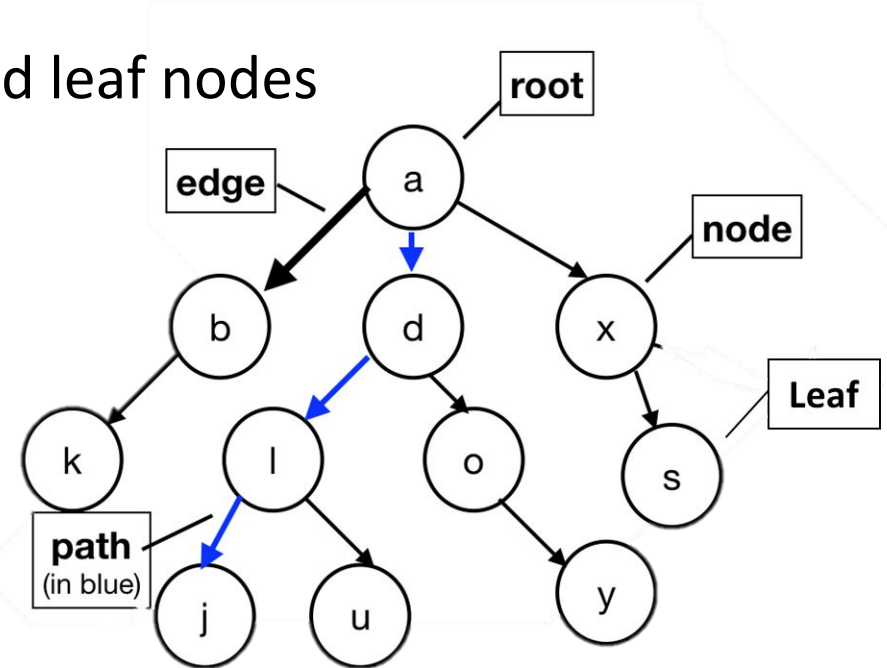
(make sure to keep up with updates);

AVL Tree

1. Difference between a “Tree”, “BST”, and an “AVL”
2. Operations:
   - find, including running times in terms of h &n
   - insert, including running times in terms of h & n
   - delete, including running times in terms of h & n
3. Strategy for a 2-child delete
4. AVL Tree Rotations
5. Minimum/maximum nodes in an AVL tree

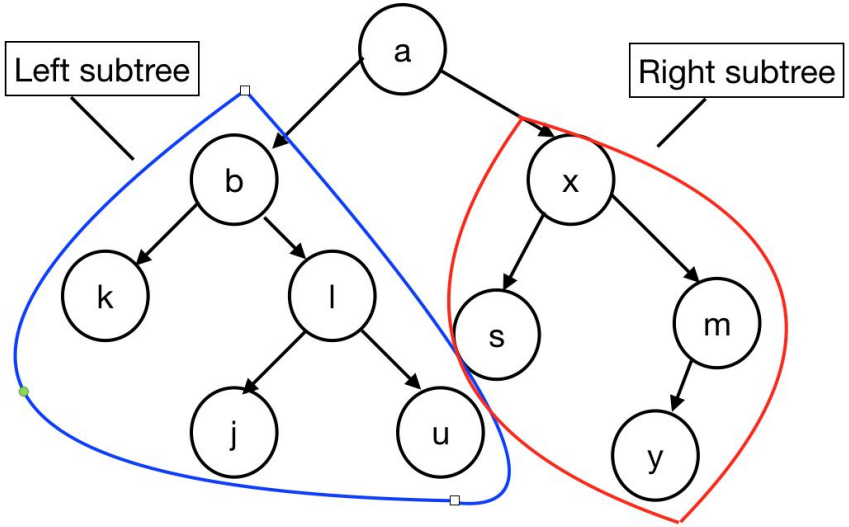
Properties of trees:

- Acyclic data structure
- Edges are always directional (downwards)
- Root is the starting point in the data structure
- Nodes without children are called leaf nodes



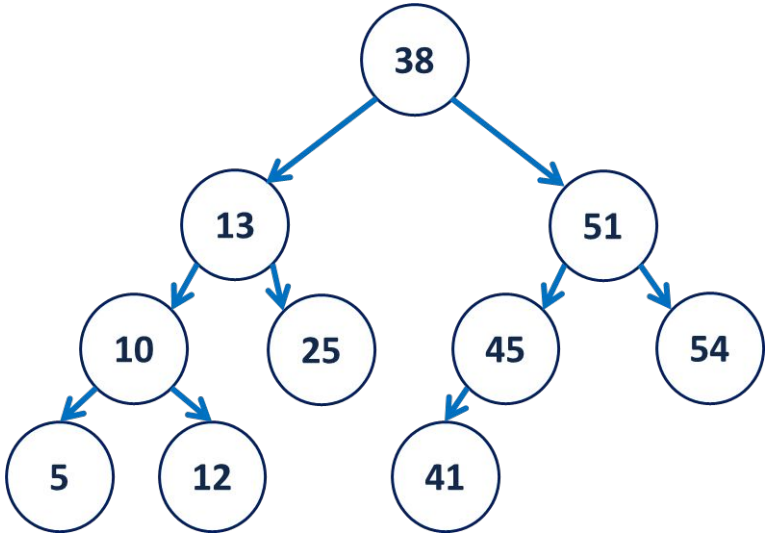
Binary Tree:

- Each node has at most two children (left and right subtree).



Properties of BST:

- Elements in the:
  - left subtree of the root is less than the root.
  - right subtree of the root is greater than the root.
- The properties are recursive → true for every node.



Formal Definition of BST:

- $T = \{\}$
- $T = \{d, T_L, T_R\}$, where

$$\forall x, x \in T_L, x < d$$

$$x \in T_R, x > d$$

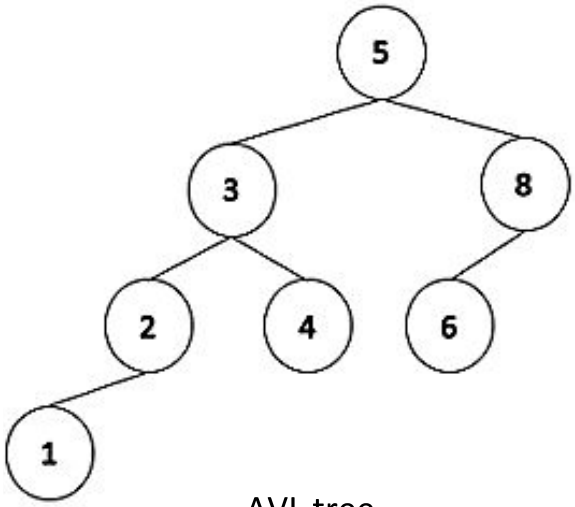
Translation:

A BST is a binary tree T such that:

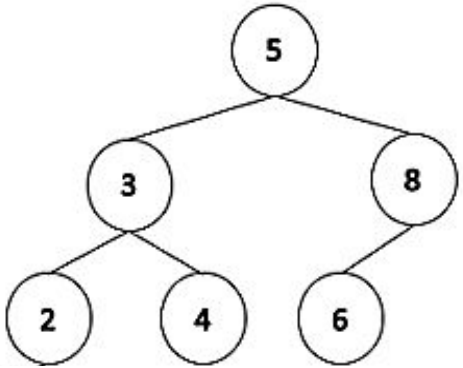
- T is either an empty tree or
- T = {d, T_L, T_R} and every element in the left subtree is less than d (value of the node), and every element in right subtree is greater than d.

AVL Tree is a Balanced Binary Search Tree:

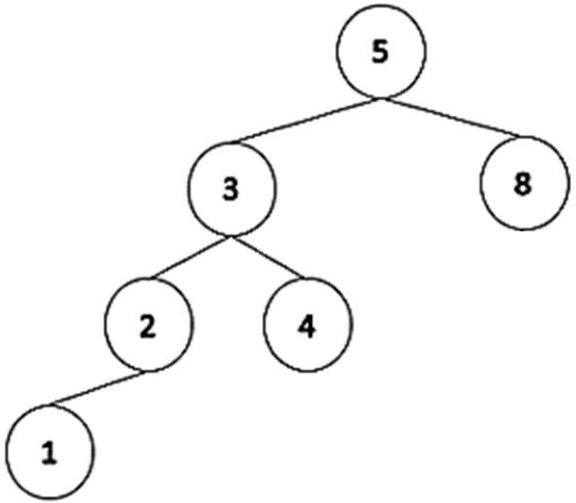
- The sub-trees of every node differ in height by at most one.
- Every sub-tree is an AVL tree



AVL tree



AVL tree



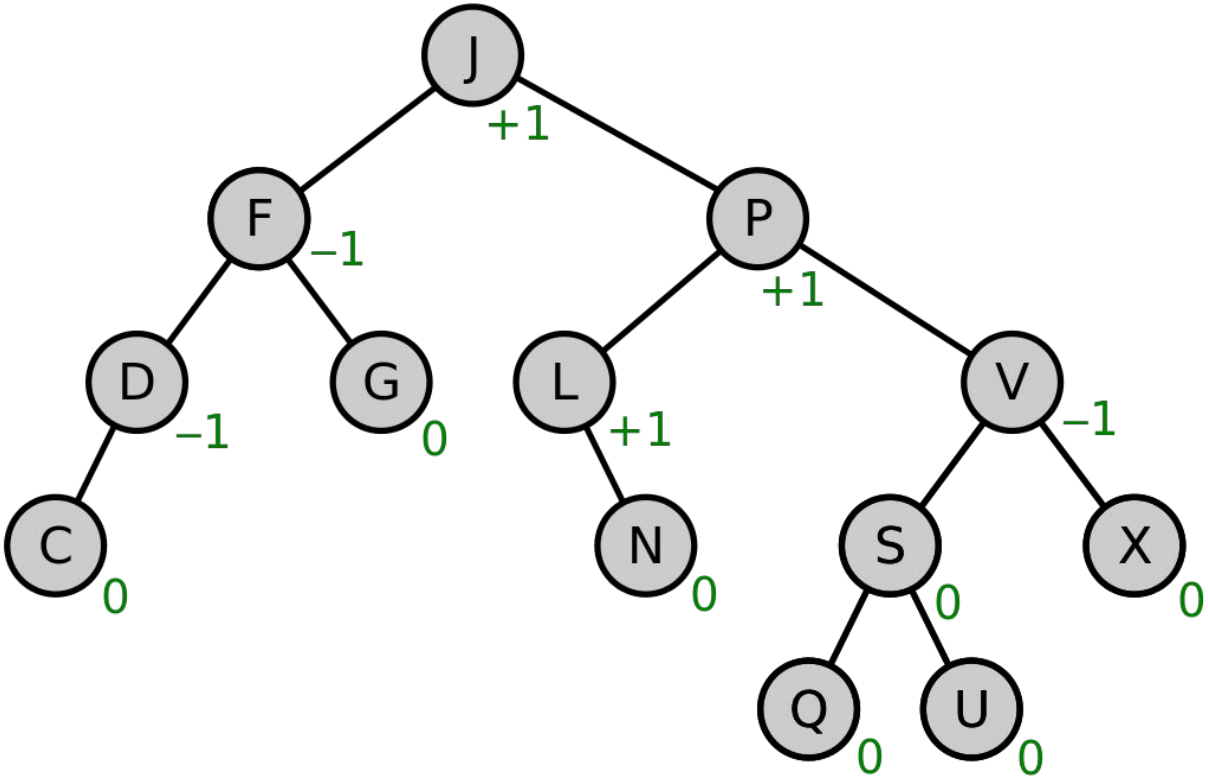
Not an AVL tree

Balance Factor:

$$b = height(T_R) - height(T_L)$$

- By definition, a tree is balanced if the absolute value of the balance factor is less than or equal to 1 $\rightarrow$ $\mathbf{|b| \leq 1}$
- Balance is determine locally (for each node).

AVL tree with balance factors



Rebalancing

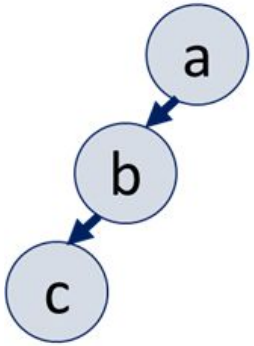
Simple rotation:

1. Left
2. Right

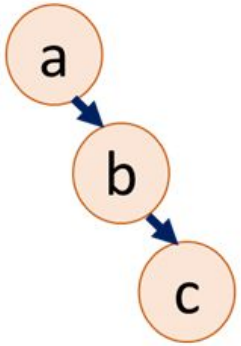
Double rotation:

3. Left Right
4. Right Left

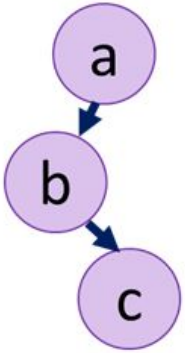
To fix imbalance, we have to do next rotations:



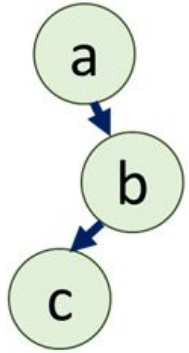
Right rotation



Left rotation



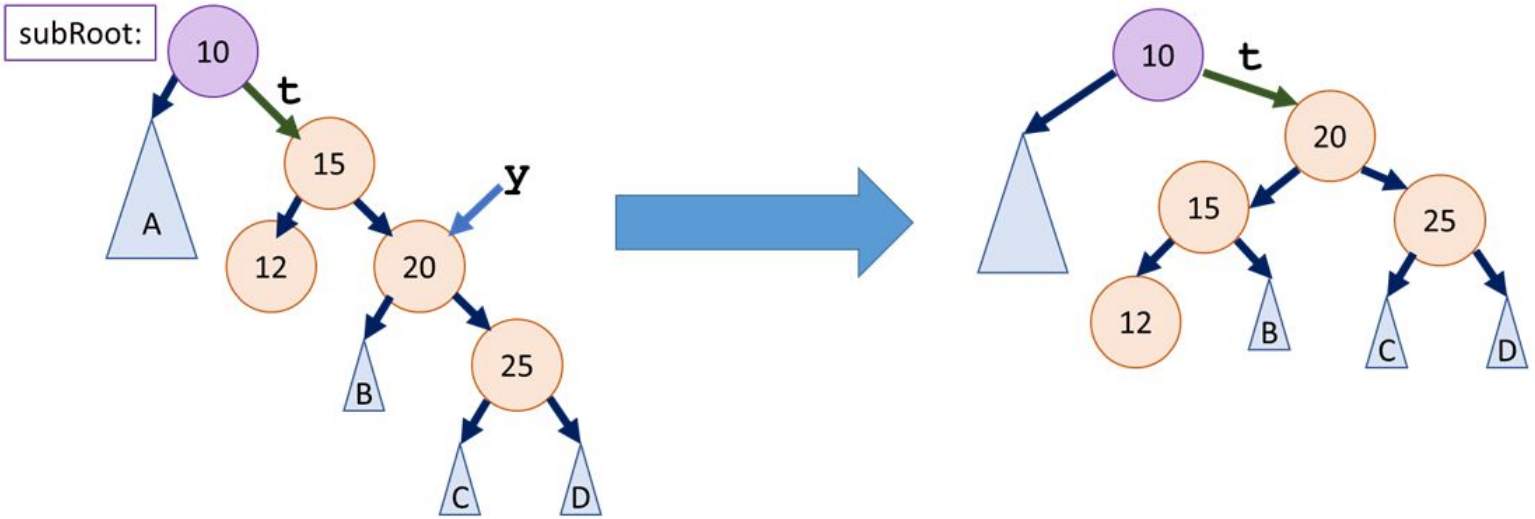
Left-Right rotation



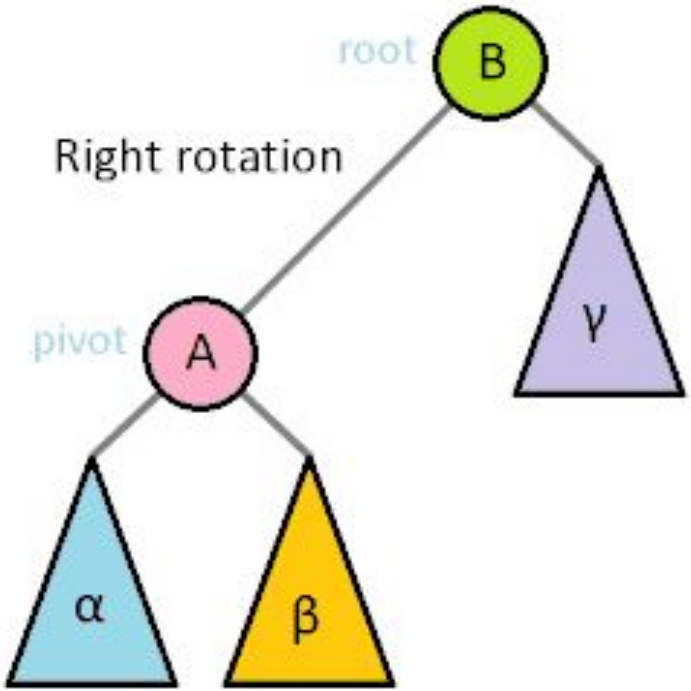
Right-left rotation

Left rotation

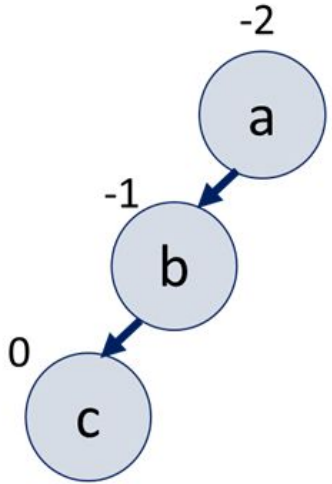
```
3  TreeNode *& t = subRoot->right;
4  TreeNode * y = t->right;
5  t->right = y->left;
6  y->left = t;
7  t = y;
8  update the heights!
```



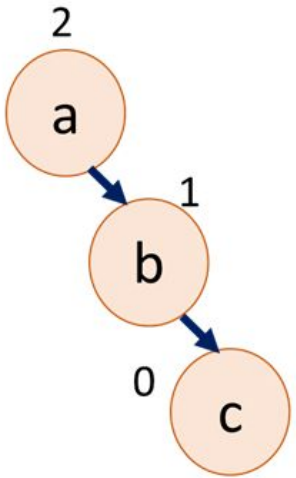
Single Rotations:



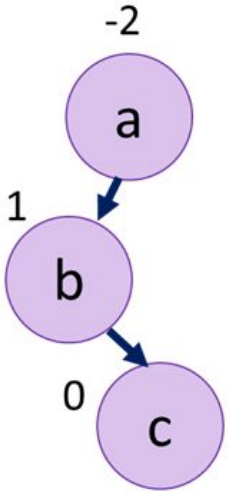
Connection between balance factor and rotation type



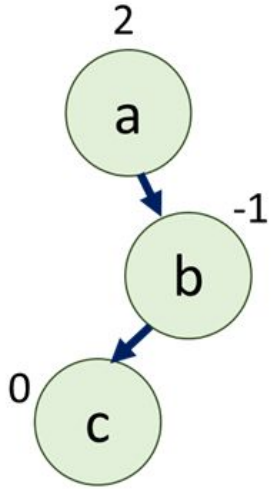
Right rotation



Left rotation

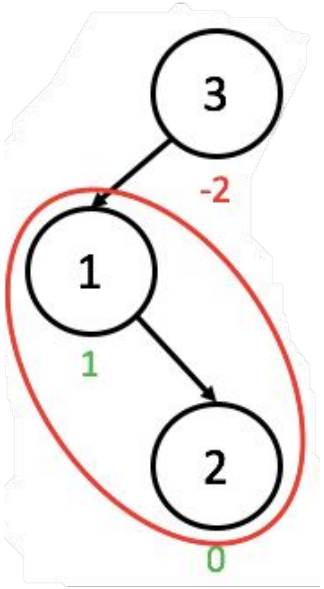


Left-Right rotation



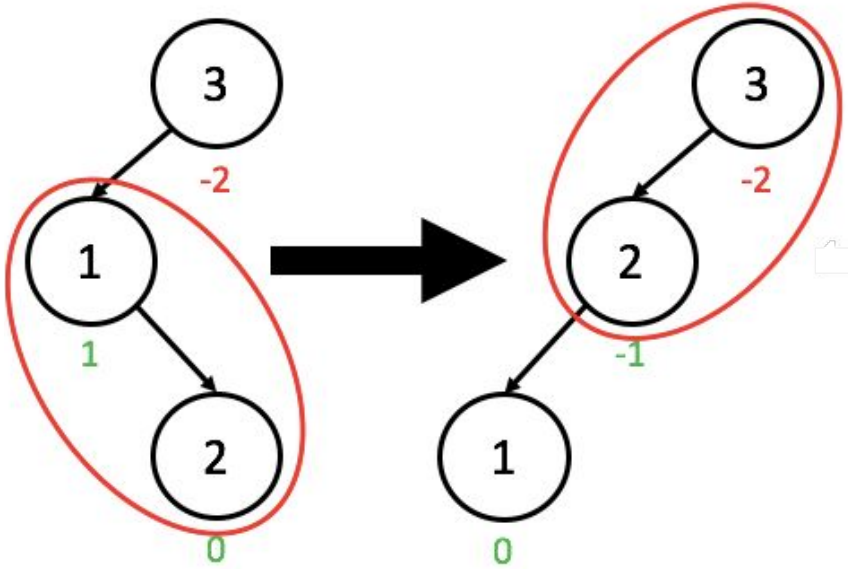
Right-left rotation

Double Rotation: Left-Right rotation



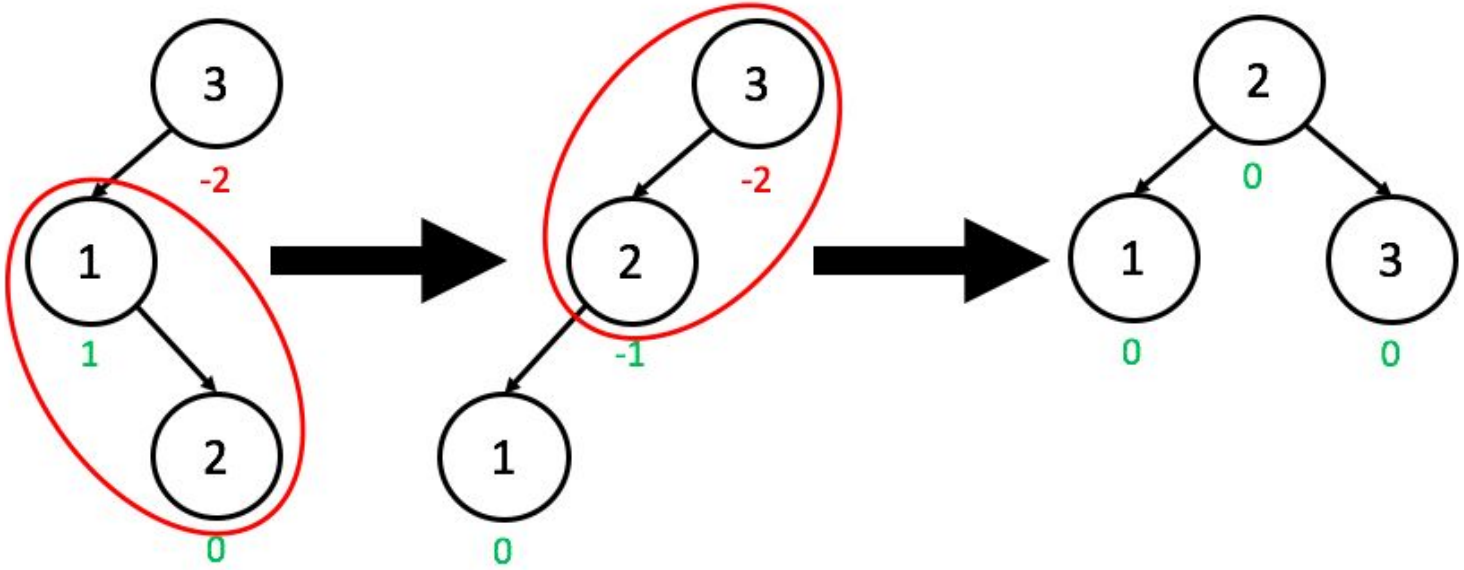
Right-left is combination of two rotations: left and right;
Double rotation is used to fix the ‘elbow’.

Double Rotation: Left-Right rotation



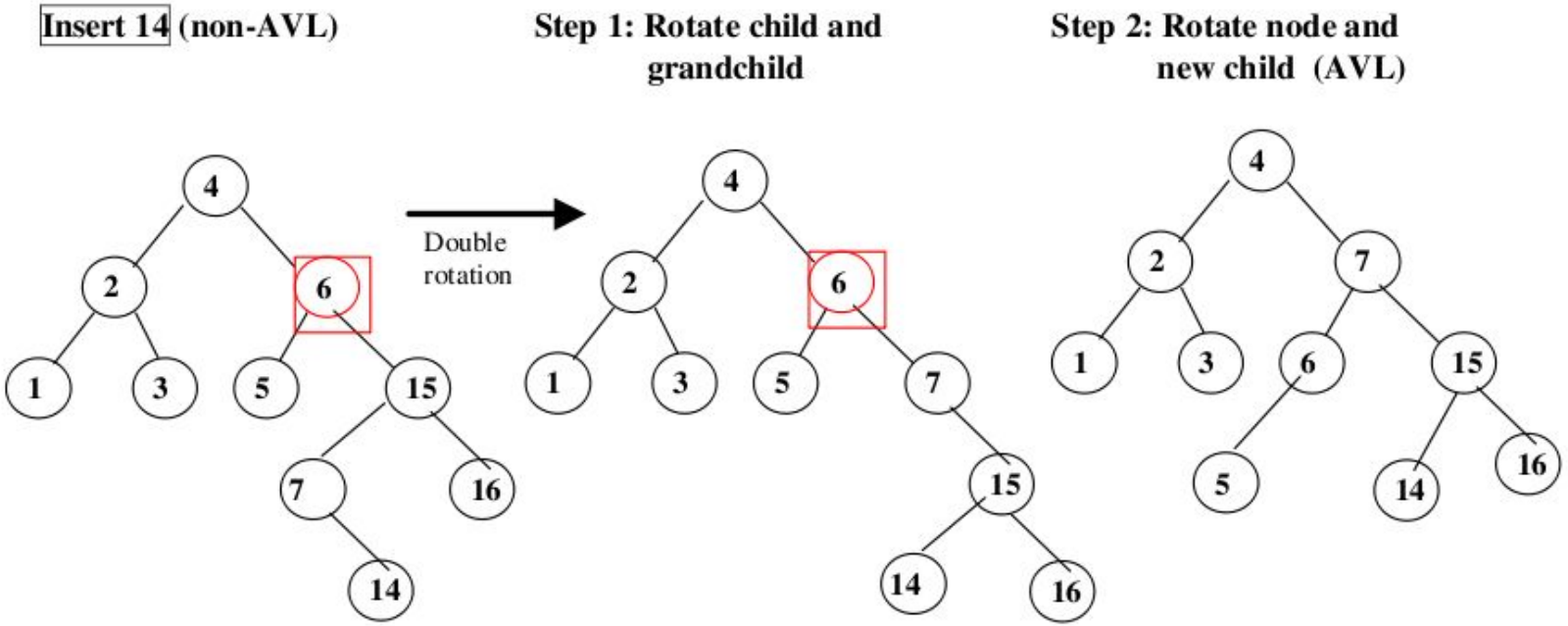
After first rotation ‘elbow’ is turned into ‘stick’

Double Rotation: Left-Right rotation



After second rotation ‘stick’ is turned into ‘mountain’

Double rotation: Right-Left rotation



Insertion into an AVL Tree

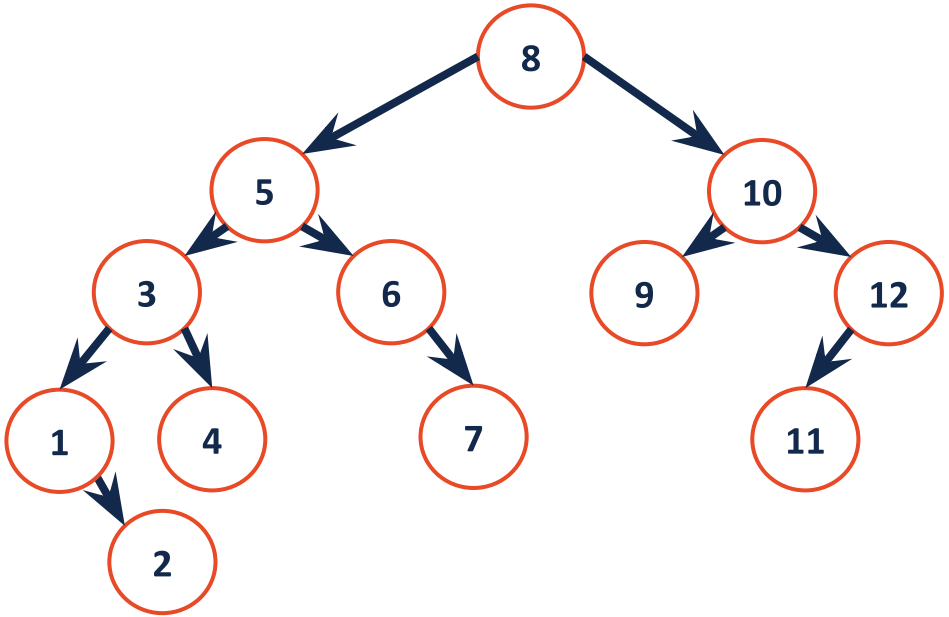
Insert (pseudo code):

1: Insert at proper place (use recursive calls to walk down the path)

2: Check for imbalance (when you unwind the recursion)

3: Rotate, if necessary

4: Update height



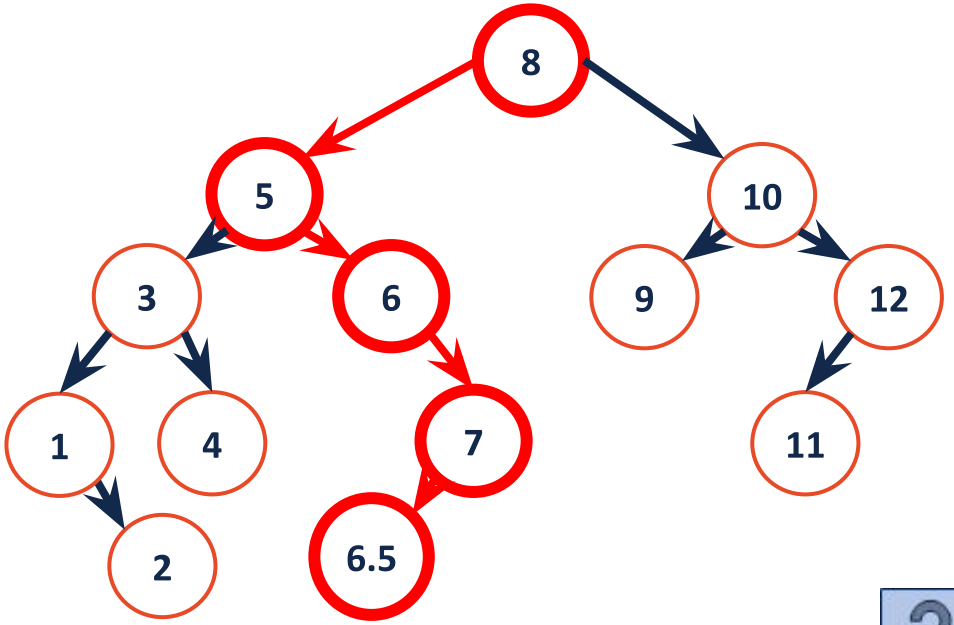

```
1  struct TreeNode {
2    T key;
3    unsigned height;
4    TreeNode *left;
5    TreeNode *right;
6  };
```

Insertion into an AVL Tree

_insert(6.5)

Insert (pseudo code):

1: Insert at proper place (use recursive calls to walk down the path)
2: Check for imbalance (when you unwind the recursion)
3: Rotate, if necessary
4: Update height




```
1  struct TreeNode {
2    T key;
3    unsigned height;
4    TreeNode *left;
5    TreeNode *right;
6  };
```

Running time

```
151 template <typename K, typename V>
152 void AVL<K, D>::_insert(const K & key, const V & data, TreeNode
    *& cur) {
153   if (cur == NULL)          { cur = new TreeNode(key, data);   }
157   else if (key < cur->key) { _insert( key, data, cur->left ); }
160   else if (key > cur->key) { _insert( key, data, cur->right );}
166   _ensureBalance(cur);
167 }
```

- Insertion calls itself $\boldsymbol{h}$ time (*for every node started from root to the leaf node*); Each insert takes O(1) time;
- After insertion we need maximum one rotation to balance the tree: $O(1)$

✓ **Running time of insert is $\boldsymbol{O(h)}$**

insert(…) -> ‘M’, ‘N’, ‘O’, ‘L’, ‘K’, ‘Q’, ‘P’, ‘H’, ‘I’, ‘A’

Remove from an AVL Tree

AVL tree remove works same as removing an element from a BST, except we have to rebalance tree if necessary.

- Perform standard BST delete for the given node n;
- Check the nodes for the imbalance once you unwind the recursion.

Fixing the first lowest point of imbalance does not balance the whole tree!
You need to check every node, going up from deleted node to the root node;

Remove from an AVL Tree

Find the element n and if:

- **has no child:**
  remove the element and rebalance the tree

- **has one child:**
  swap n with the child node, remove n and rebalance the tree

- **has two children:**
  Swap n with IOP, call remove for n and rebalance the tree

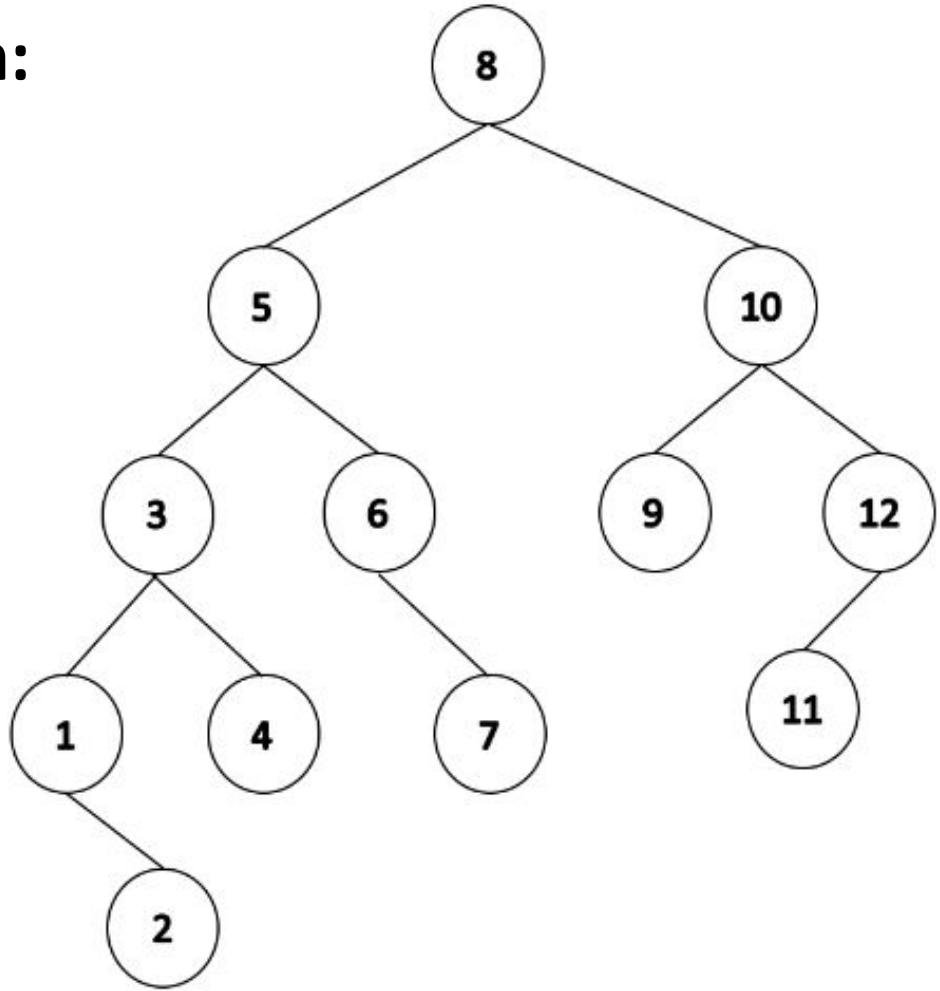
Remove with two children:

Find the element n:

- ✓ Replace it with IOP
- ✓ Call remove again for n
- ✓ Rebalance the tree

Identify IOP for next nodes:

5	?
10	?
8	?



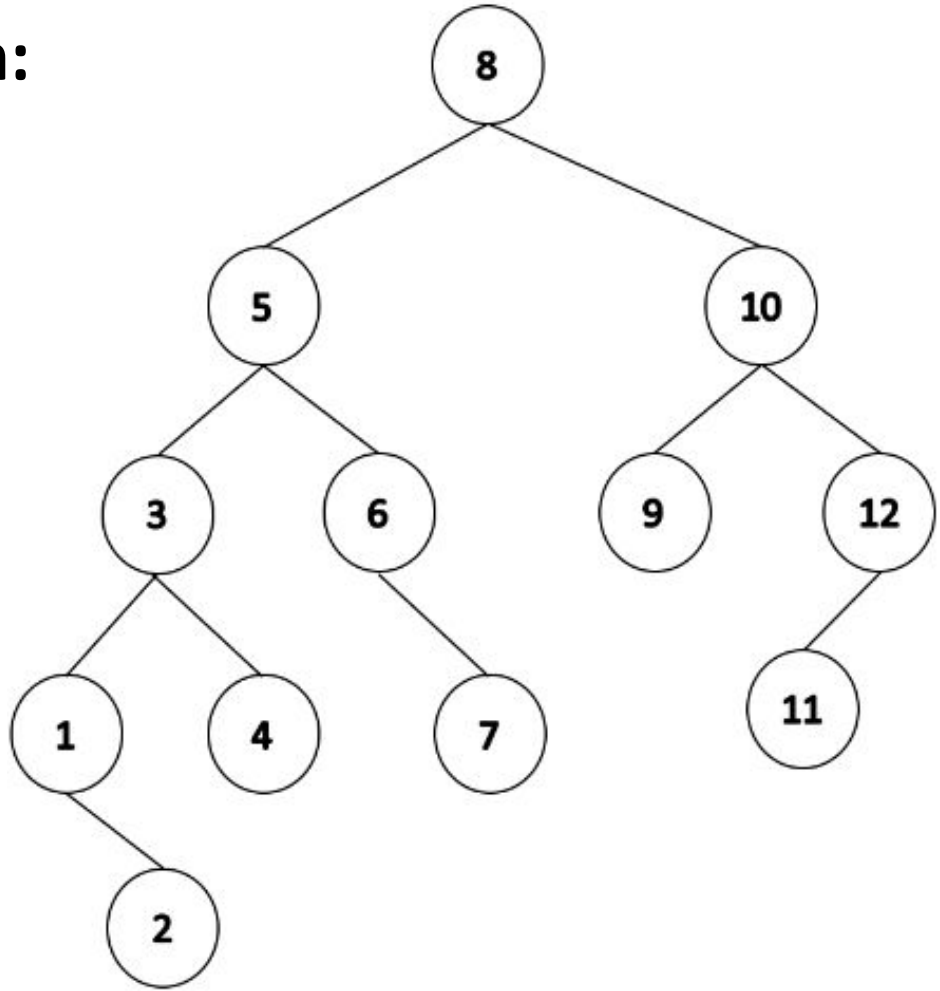
Remove with two children:

Find the element n:

- ✓ Replace it with IOP
- ✓ Call remove again for n
- ✓ Rebalance the tree

Identify IOP for next nodes:

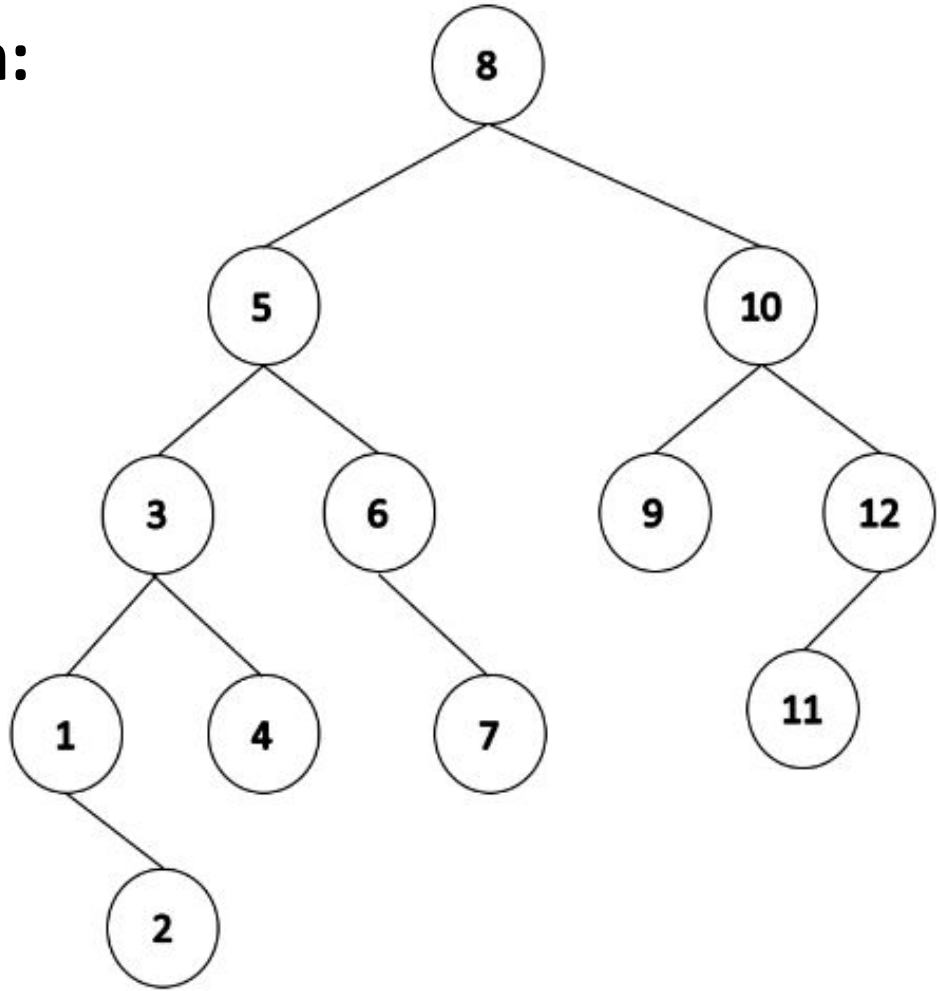
5	4
10	?
8	?



Remove with two children:

Find the element n:

- ✓ Replace it with IOP
- ✓ Call remove again for n
- ✓ Rebalance the tree



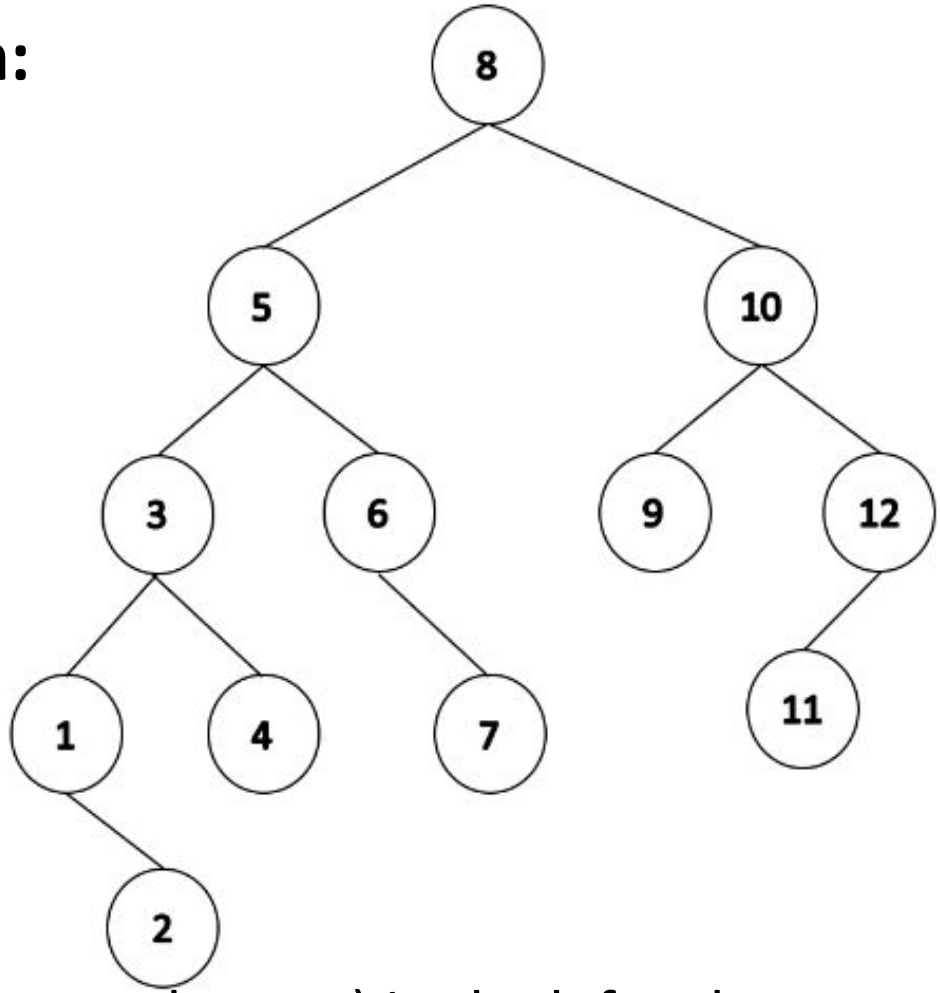
Identify IOP for next nodes:

5	4
10	9
8	?

Remove with two children:

Find the element n:

- ✓ Replace it with IOP
- ✓ Call remove again for n
- ✓ Rebalance the tree



Identify IOP for next nodes:

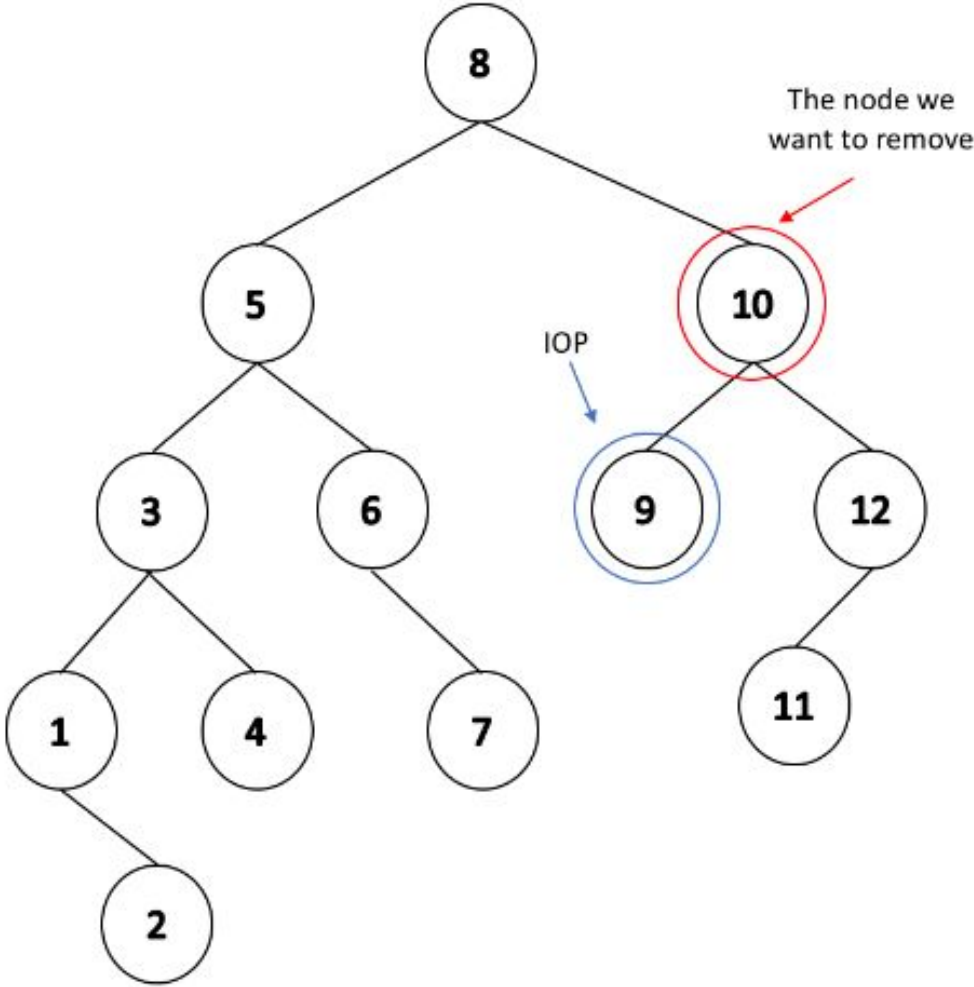
5	4
10	9
8	7

IOP: Right most element (maximum element) in the left subtree;

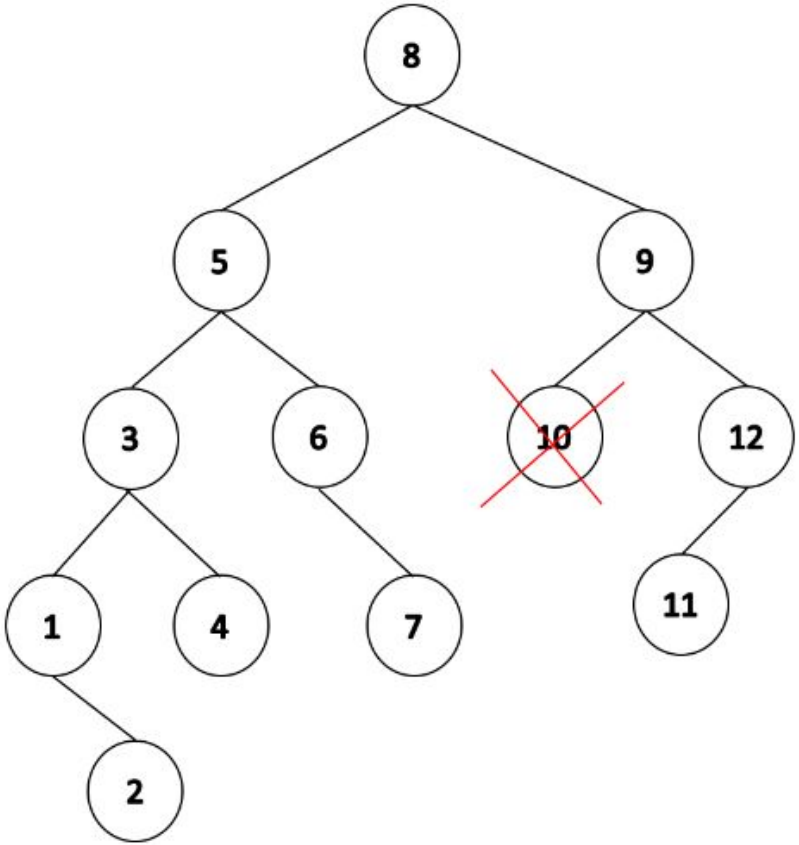
Remove with two children:

Find the element n $(= 10)$:

1. Replace it with IOP
2. Call remove again for n
3. Rebalance the tree

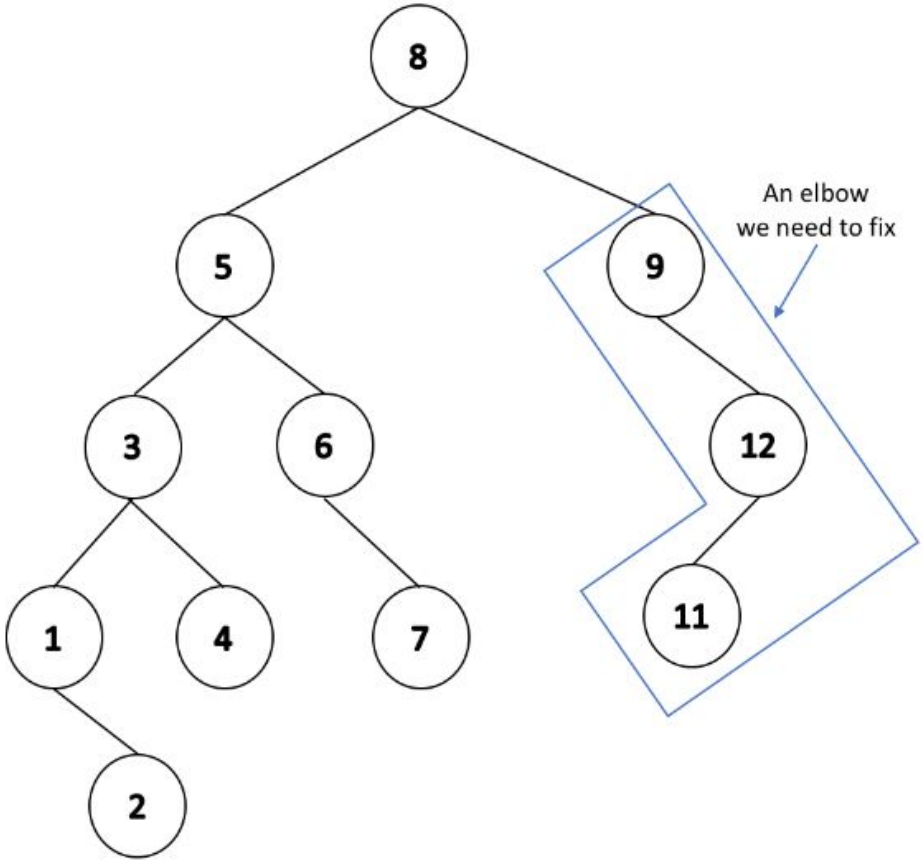


2. Call remove again for n



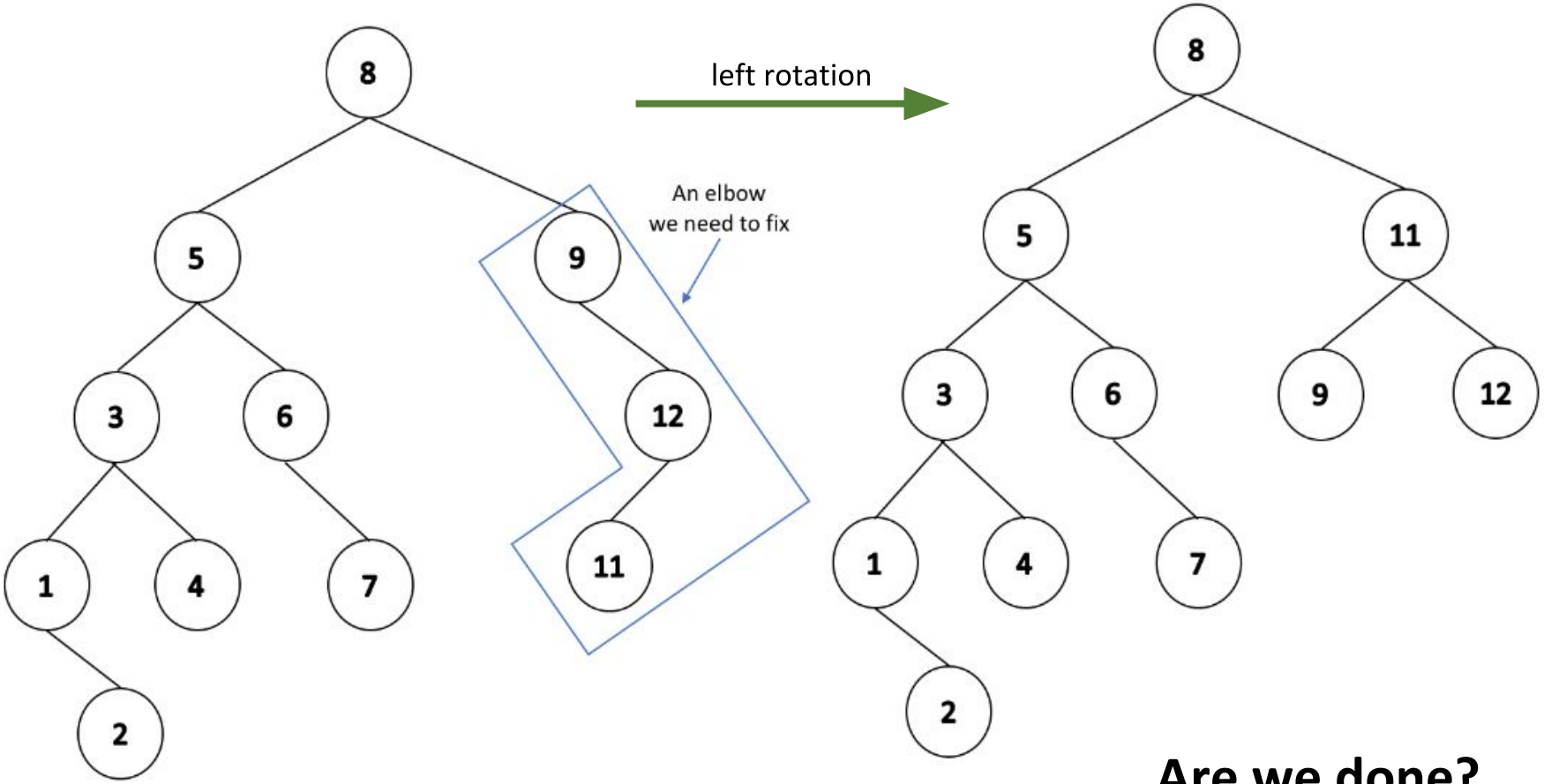
3. Rebalance the tree:

Find lowest point of imbalance



3. Rebalance the tree:

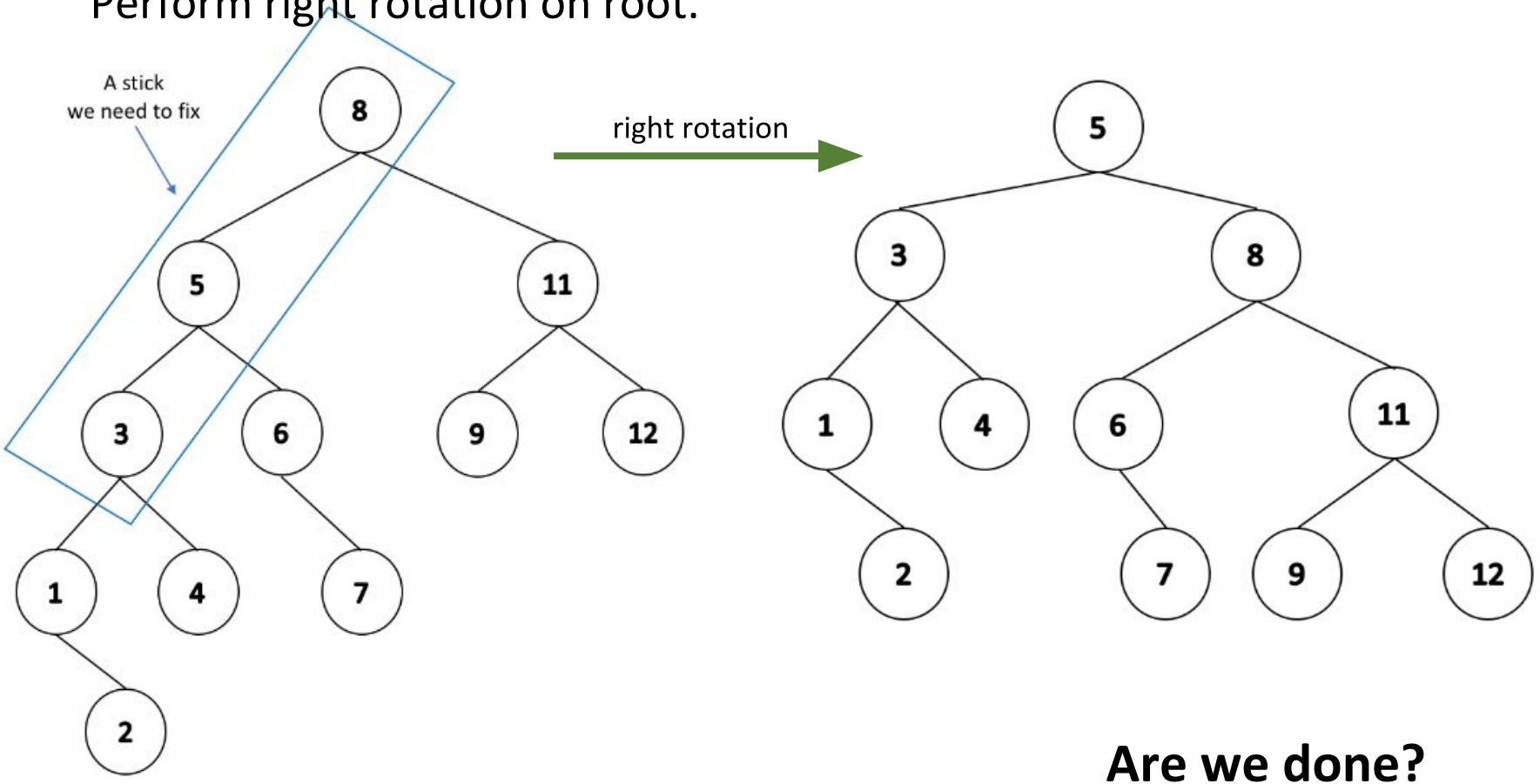
Perform left rotation



Are we done?

3. Rebalance the tree:

Perform right rotation on root.



Are we done?

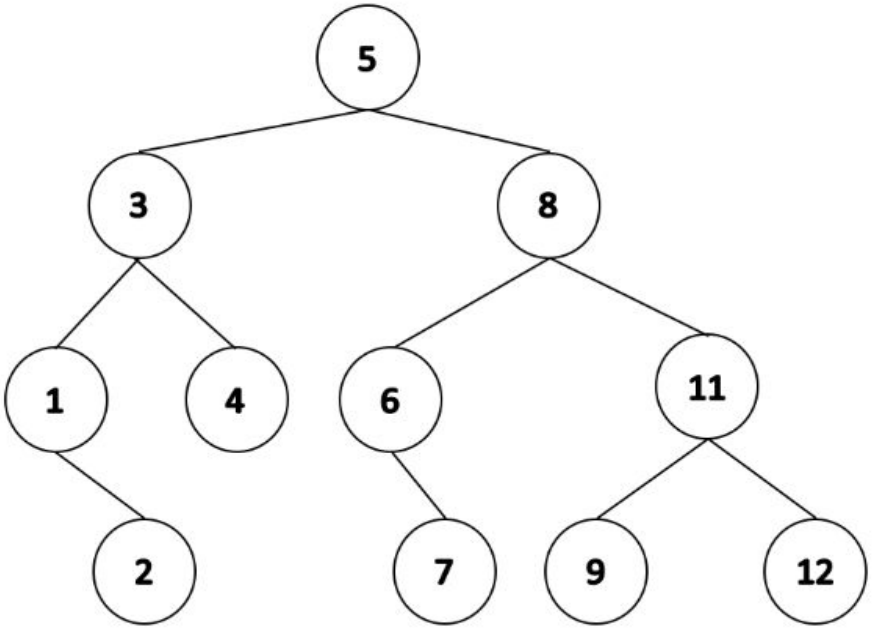
Rebalance the tree.

After removing element,

check imbalance for every node on the path until the root node *(up to h node)*

Rebalance if necessary

(up to h rotation).



- find(...) $\rightarrow$ $O(h) + 0$ rotation
- insert(...) $\rightarrow$ $O(h) + up\ to\ 1$ rotation
- remove(...) $\rightarrow$ $O(h) + up\ to\ h$ rotations
  - Each rotation is $O(1)$.
  - Doing h rotations is $h * O(1) = O(h)$
  - $O(h) + O(h) = 2 * O(h) = O(h)$

All operations take O(h) and O(h) = O(log n)!

Running times of different operations

	Unbalanced BST	AVL tree
Printing the value of the root node	$O(1)$	$O(1)$
Searching for an element	$O(h)^1$	$O(h) = O(\log n)$
Printing the value of each node in the tree	$O(n)$	$O(n)$
Finding the node with largest value	$O(h)^1$	$O(h) = O(\log n)$

1. Since BST is not balanced, $O(h) \leq O(n)$, so worst worst case for find will be O(n) in BST

Bounds on n given h

- **Lower bound** on the number of nodes given height **h**
  - **N(h) = N(h-1) + N(h-2) + 1**
  - A loose analysis gives
    - **$n >= 2^{h/2}$**
    - **$h <= 2 \times \lg(n)$** or **$h=O(\lg(n))$**
  - This gives an **upper bound** for **h** in terms of **n**
  - To be exact: h <= 1.44 * lg(n)
- **Upper bound** on the number of nodes given height **h**
  - A perfect tree has the most number of nodes
  - **$1+2+4+...+2^h = 2^{h+1} - 1$**

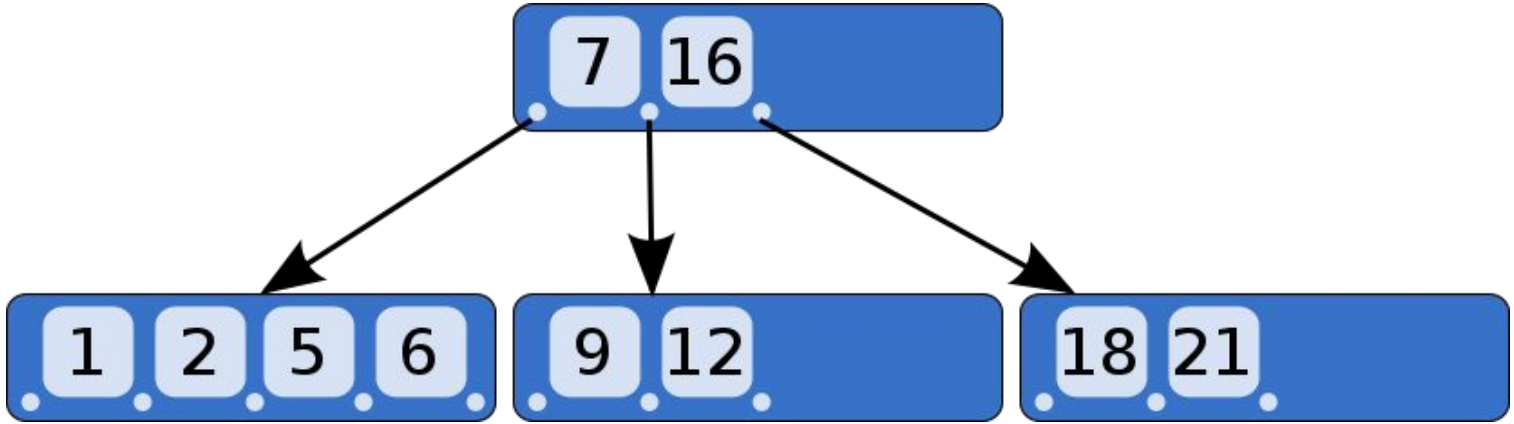
Questions?

B-tree

- Difference between a key and a node in a BTree
- The order of a BTree
- Structural properties of a BTree
- The min/max keys/children in each node in a BTree
- The min/max total number of keys in a BTree
- Advantages of a BTree
- Running time of a Btree

B-tree

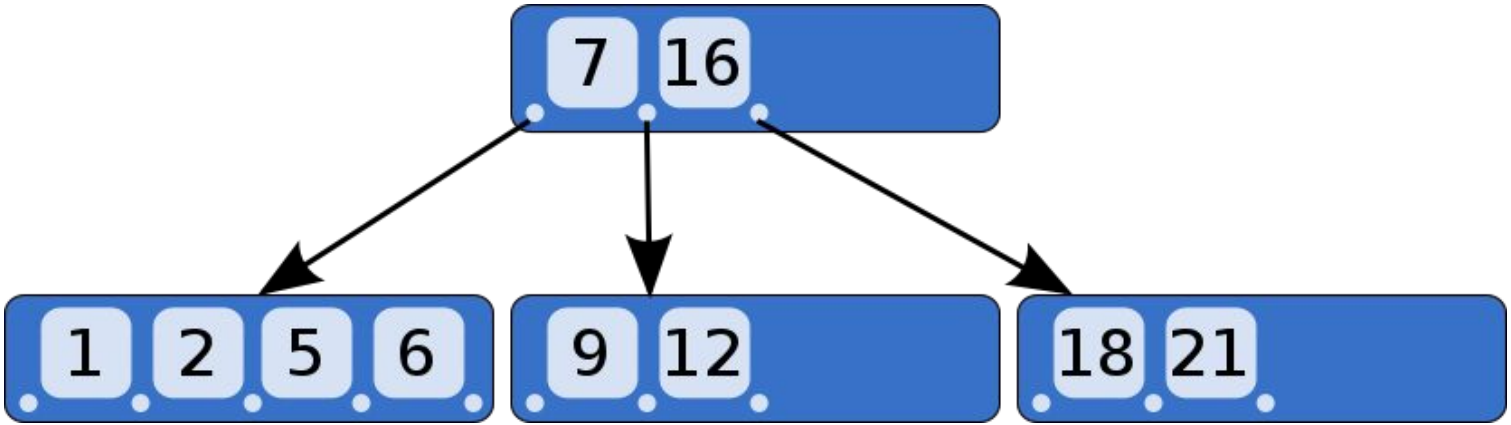
- B-Tree is a self-balancing search tree.
- The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node.
- B-Tree is defined by the term order `m`: maximum number of children node can have.
- Each B-tree node has up to `m-1 <key, data> pairs`



B-tree Properties

A **B-Trees** of order **m** is an m-way tree:

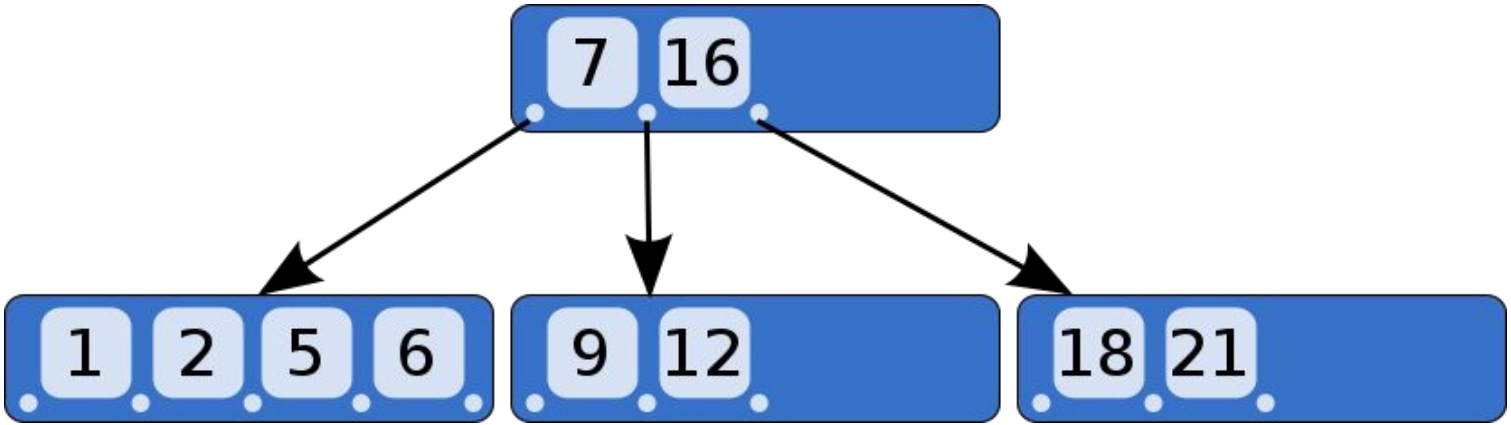
1. All keys within a node are ordered
2. All internal nodes have exactly **one more child than keys**
3. All leaves are on the same level



B-tree Properties

A **B-Trees** of order **m** is an m-way tree:

4. Root nodes can have: [1, m-1] keys and [2, m] children
5. All non-root, nodes have [ceil(m/2)-1, m-1] keys.
6. All non-root internal nodes have [ceil(m/2), m] children

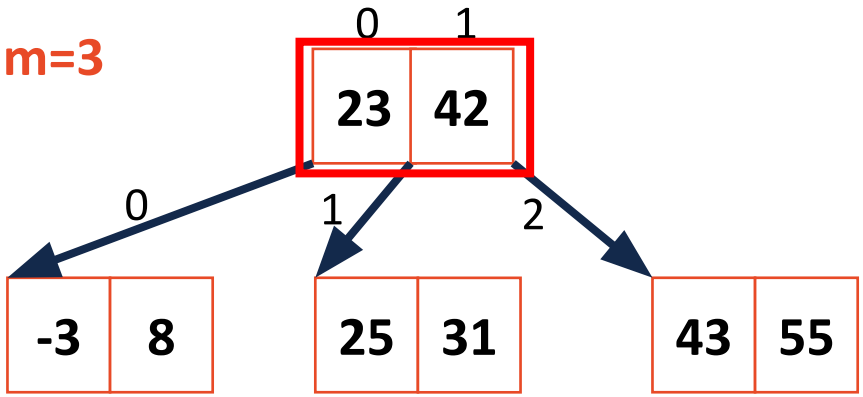


Insertion

1. Find the proper leaf node (recursive calls) and insert the new element into that leaf node (keeping sorted property).

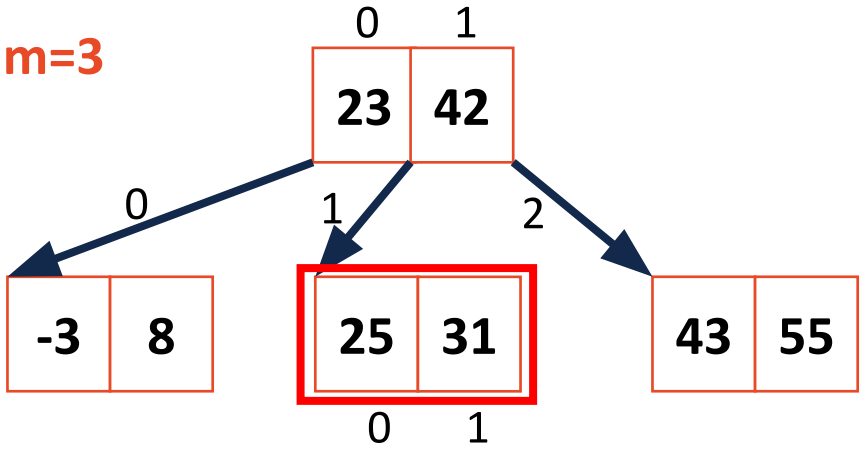
2. If the leaf node has more than m-1 elements: we must ***split*** the node by ***throwing up*** the middle element to the parent node.
3. Check the parent nodes for the overflow when you unwind the recursion;

Insert(28)



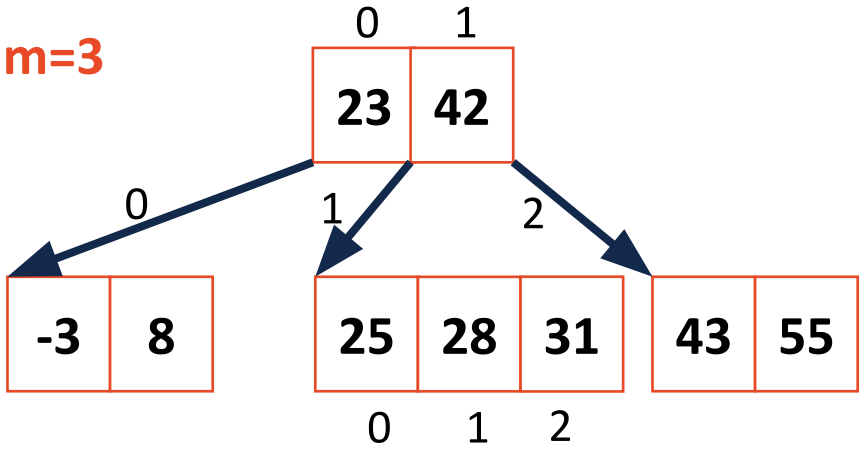
1. Find the corresponding leaf node, start from the root:
   - 23 < **28** <42 $\Longrightarrow$ follow pointer 1;

Insert(28)



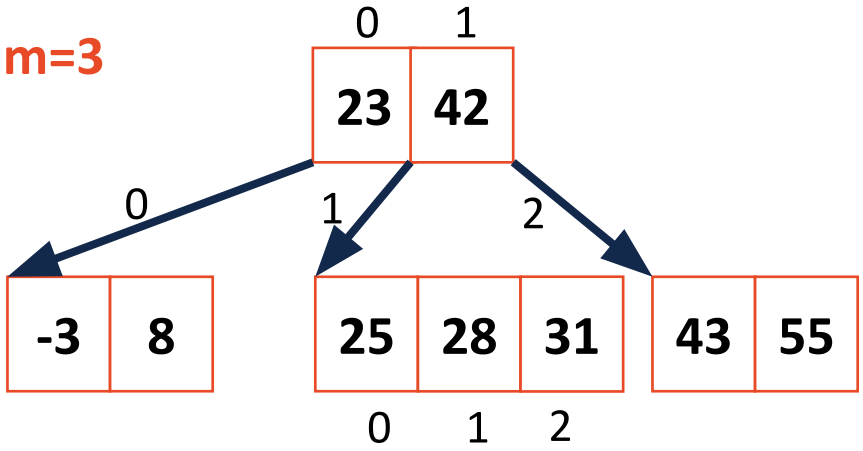
3. Current node is the leaf
4. Insert 28 on the right index: 25 < **28** <31

Insert(28)



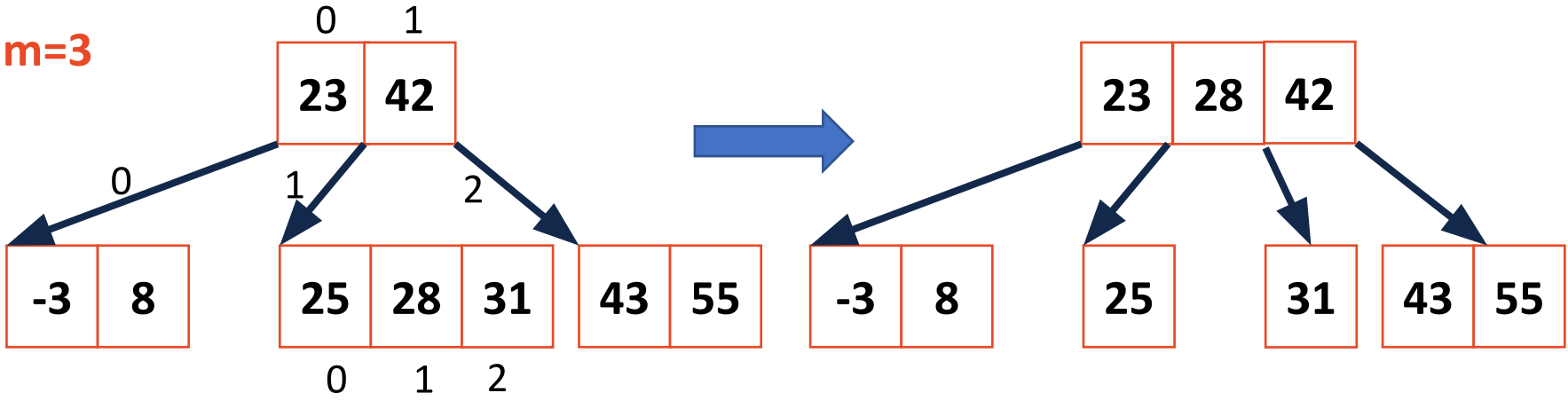
3. Current node is the leaf
4. Insert 28 on the right index: 25 < **28** <31
5. Check node for overflow: since m=3, we have to split the node

Insert(28)



3. Current node is the leaf
4. Insert 28 on the right index: 25 < **28** <31
5. Check node for overflow: since m=3, we have to ‘throw up’ middle element and split the node.

Insert(28)

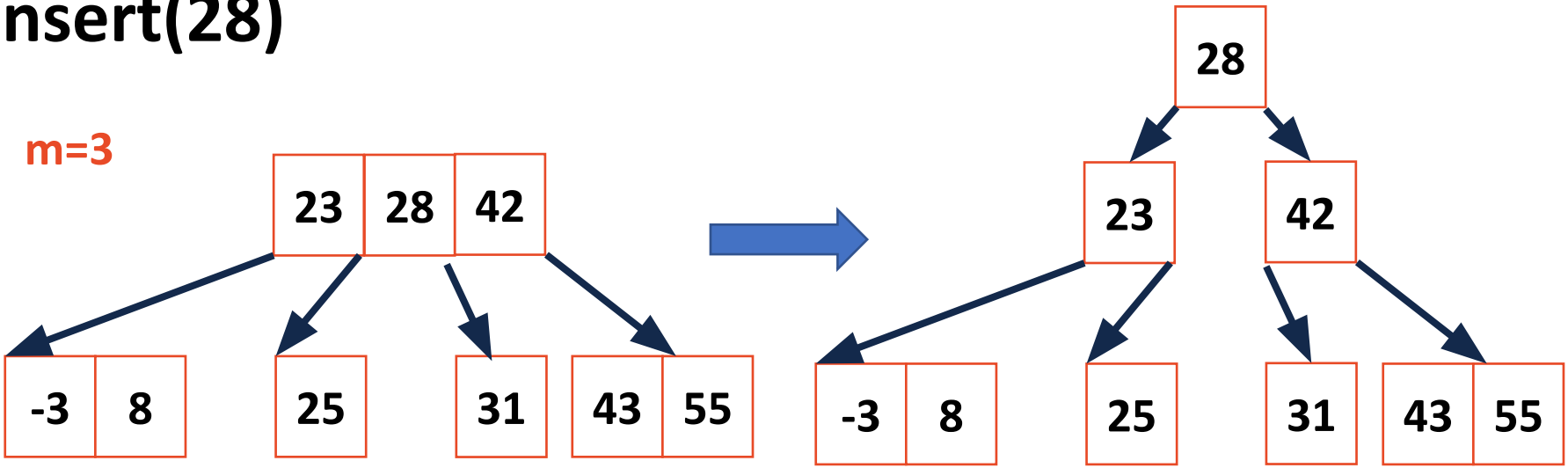


Split node:

Split the node and insert middle element into the parent node;

Check the parent node for overflow => split parent;

Insert(28)



Split node (parent):
Split the node and ‘throw up’ middle element (create new node with 28);
Check new node for overflow => we are done!

B-Tree with height h: minimum and maximum number of elements

- $min = 2 * \left\lceil \frac{m}{2} \right\rceil^{h} - 1$
- $max = m^{h+1} - 1$

We showed that $h \leq \log_{\lceil \frac{m}{2} \rceil} \frac{n+1}{2} \approx \log_m n$

h determines maximum number of **disk/network seeks** possible when searching for data.

https://drive.google.com/open?id=1dNTPQO3Eg-GMhwrILWf3wUFaR_bhydoL
https://docs.google.com/document/d/1WY0PqkN2ceaPN_0tl4OFSRpZKcXWEDGe46Oq44pLPt0/ed

Advantages of B-Tree

To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory.

Generally, a B-Tree node size is kept equal to the disk block size or network packet.

Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

https://drive.google.com/open?id=1dNTPQO3Eg-GMhwrILWf3wUFaR_bhydoL Exercise 1.3

Space complexity

	Average	Worst case
Space	O(n)	O(n)

Running time

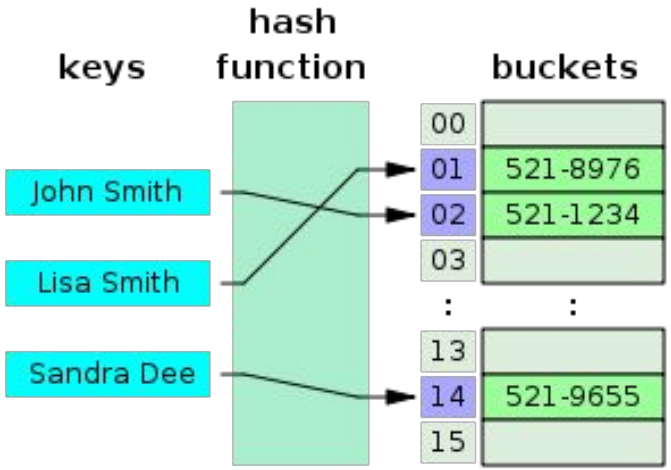
Algorithm	Average	Worst case
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Hash Table

- Hash functions
- Properties of a good hash function
- SUHA
- Hash Table Array
- Load Factor
- Hash Table Collisions
- Open Hashing vs. Closed Hashing
- Separate Chaining
- Linear Probing
- Double Hashing
- Re-Hashing
- Runtime of a hash table, in terms of n
- Load factor’s impact on running times
- Properties of a good hash table

Hash function:

- Distribute entries across an array in ideally O(1) time
- Hash function computes an index that suggests where the entry will be found inserted/found
- **A perfect hash function** is one that gives a unique output for each unique input, thus resulting in no collisions



A small phone book as a hash table

Properties of a good hash function:

1. Computation Time: running time must be O(1)
2. Deterministic:

$$If\ k1 == k2 => h(k1) == h(k2)$$

3. Satisfy the SUHA – Simple Uniform Hashing Assumption:

$$\forall\ keys\ i, j\text{: } if\ i\ != j,\ p\big(h(i) = h(j)\big) = 1/N$$

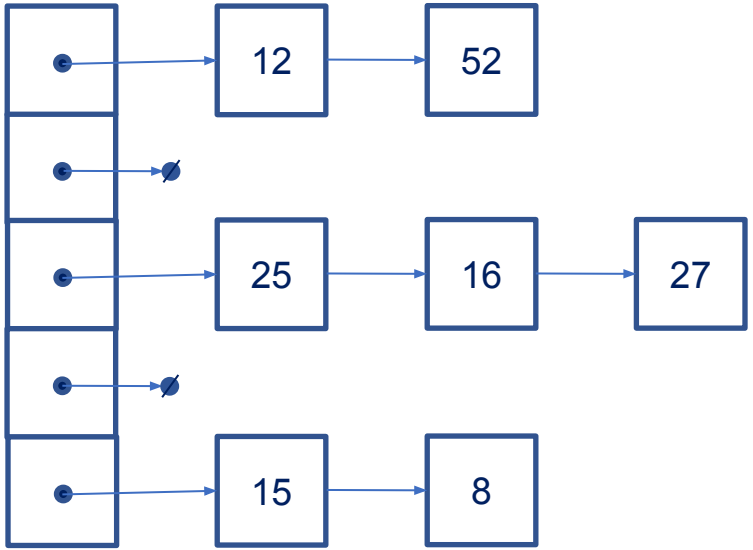
SUHA – Simple Uniform Hashing Assumption:

$$\forall\, keys\ i, j\ if\ i! = j,\ p\big(h(i) = h(j)\big) = 1/N$$

- Evenly distribute items in the hash table
- Each item has equal probability to be placed into a slot

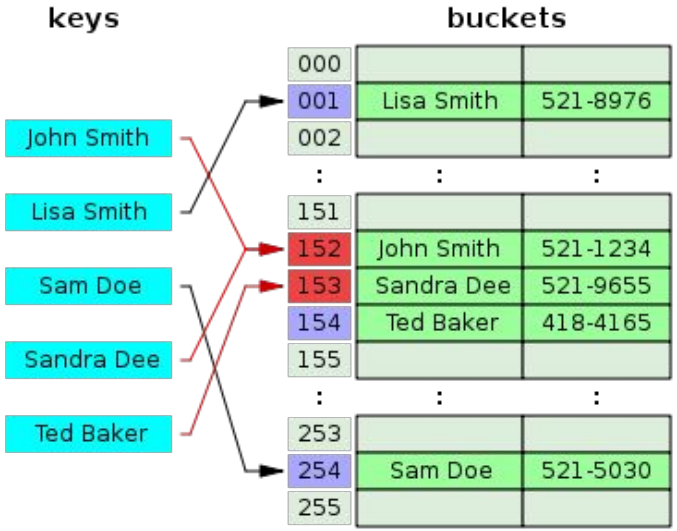
Open Hashing (separate chaining) vs. Closed Hashing

- **Open Hashing**: Values are stored outside of the table - *Each cell of hash table points to a linked list of records that have same hash value.*
- **Useful** if data consists of large records, they can be stored outside of the table.



Open Hashing (separate chaining) vs. Closed Hashing

- **Closed Hashing (Open addresses)** - All entry records are stored in the bucket array itself.
- **Useful** when we don't care about structure speed and data is easy to copy.



Hash Table Collision handling:

- *Open hashing* - Separate Chaining
- *Closed hashing* - Linear Probing
- *Closed hashing* - Double hashing

A collision in a hash table happens when the hash function $hash(\ldots)$ gives the same index i for multiple data values as they are being inserted into the table.

Load Factor

$$\alpha = \frac{\text{The number of elements inside } hashtable\ (\boldsymbol{n})}{\text{the total size of the } hashtable\ \ (\boldsymbol{N})}$$

Load factor is used to detect when to resize the table;

After resizing the table we have to rehash every element!
(Since size of the hashtable changes, hashvalue for keys also change)

New size must be **prime.**

We are rehashing once in n times, we are hashing n elements, each takes O(1), so rehashing takes O(1)*

After resizing the table we have to rehash every element!

(Since size of the hashtable changes, hashvalue for keys also change)

We are rehashing elements once in n times

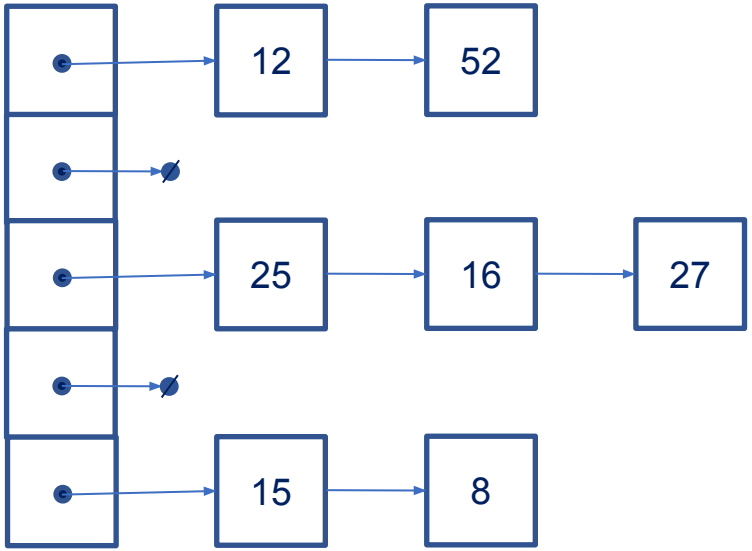
Number of elements $= n$.

Each insert takes $O(1) \Rightarrow$ so rehashing takes $O(1)^*$

Resizing the hash table takes $O(1)^*$

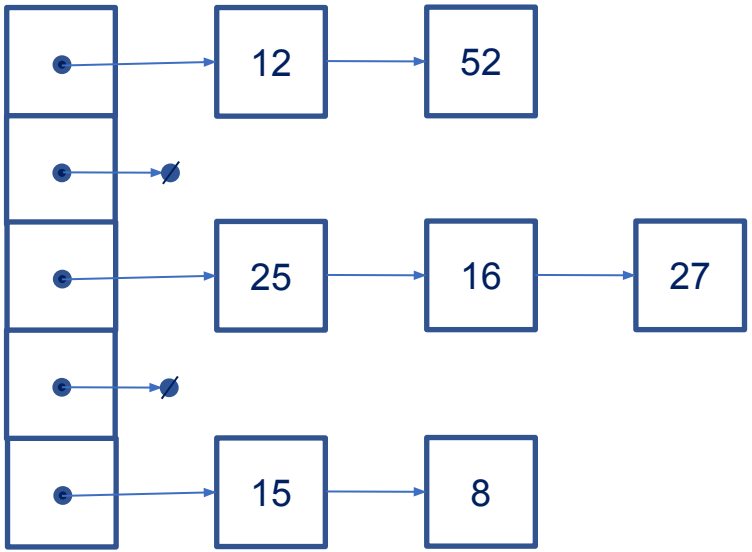
Separate Chaining

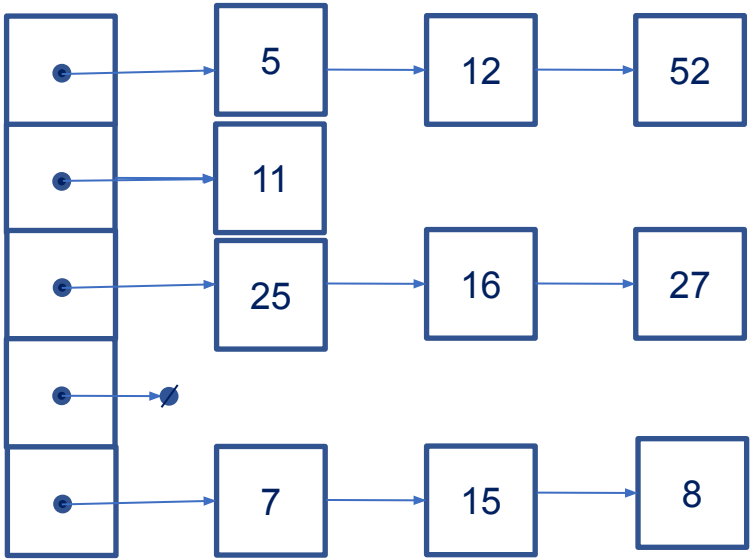
Each slot in the table holds a pointer to the head of a linked list where the actual data values are stored;



If a collision occurs, then the new data is simply inserted **at the head** of the linked list for that slot.

Insert 5, 11, and 7 into the hash table below: $h(5) = 0, h(11) = 1$, and $h(7) = 4$





	Worst Case	SUHA
Insert	O(1)	O(1)
Remove/Find	O(n)	O(n/N) = O(α)=O(1)

Collision resolution, closed hashing: (a) Linear Probing:

if `h(v) = i` causes a collision (index `i` is already occupied), then we increment (`i++`) until we find an empty spot in the table.

(modulus function is used 'wrap around' and continue incrementing from the start of the table).

Hash function: h(key, size) = key % size

14			31		12	27

Size =7

Insert: 33;

`i = h(key = 33, size = 7) = 33 % 7 = 5`

Calculate `i=(i+1) mod size,` until `array[i]` becomes an empty slot:

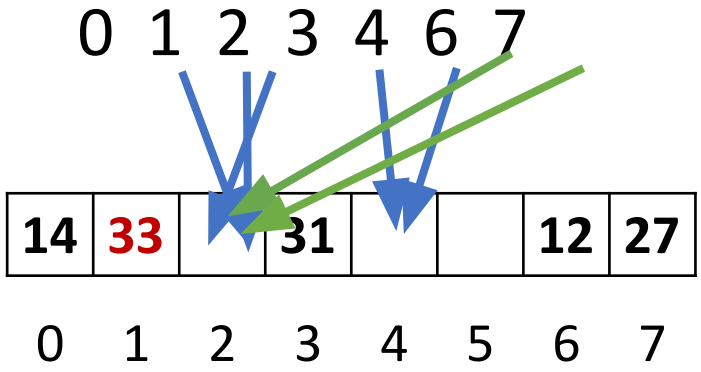
```
i = (5+1) % 7 = 6
i = (6+1) % 7 = 0
i = (0+1) % 7 = 1
```

14	33		31		12	27

Collision resolution, closed hashing: (a) Linear Probing:

if `h(v) = i` causes a collision (index `i` is already occupied), then we increment (`i++`) until we find an empty spot in the table (We are using modulus function to 'wrap around' and continue incrementing from the start of the table).

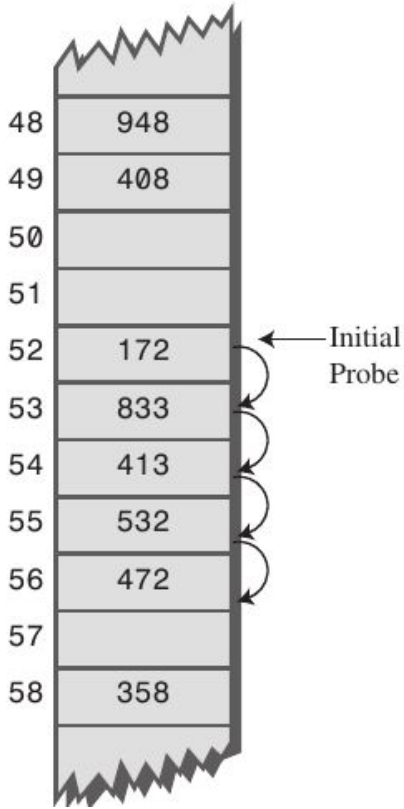
Which keys will map to the same index?:



As you can see, most of the keys map to the same index. 😭

Linear Probing: clusters

- Primary Clustering is the tendency for a collision resolution scheme such as linear probing to create long runs of filled slots near the hash position of keys.
- If the primary hash index is x, subsequent probes go to x+1, x+2, x+3 and so on, this results in Primary Clustering.
- Once the primary cluster forms, the bigger the cluster gets, the faster it grows. And it reduces the performance.



Collision resolution, closed hashing: (b) double hashing:

Given two hash functions h_1 and h_2, the i_{th} location in the bucket sequence for value k in the hash table is:

$$h(i,k) = \left(h_1(k) + i * h_2(k)\right) mod\ size$$

The idea of double hashing is to add a second hash function that will be used as a step function when looking for 'next' available index to avoid clusters

! h_2 should never return 0, for any key.

! Size of the array should be relative prime to the range of the second hash function (if size of array is prime, this will hold).

Collision resolution, closed hashing: (b) double hashing:

$$h(i,k) = \left(h_1(k) + i * h_2(k)\right) mod\ size$$

S = { 16, 8, 4, 13, 29, 11, 22 }

$h_1(k) = k \% 7$

$h_2(k) = 5 - (k \% 5)$

	8	16		4		13
	8	16	29	4		13
	8	16	29	4	11	13
22	8	16	29	4	11	13

Running time and Space complexity

	SUHA	Worst Case
Find	$O(1)$, with $\alpha < \sim 0.7$	$O(n)$
Insert	$O(1) *$	$O(n)$
Storage Space	$n/\alpha * \ data \rightarrow \ O(n)$	

Questions?

Heaps

- Construction of a Heap
- Heap runtime
- HeapifyUp
- HeapifyDown
- Priority queue ADT

Priority queue

- In computer science, a priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it.

- In a priority queue, an element with high priority is served before an element with low priority.

Priority queue

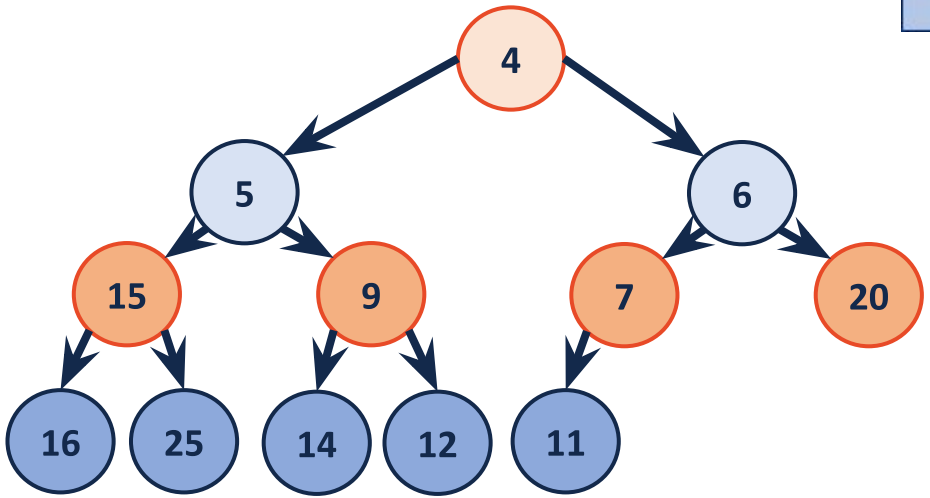
- `is_empty`: check whether the queue has no elements.
- `insert:` add an element to the queue with an associated priority.
- `pop`: remove the element from the queue that has the highest priority, and return it.
- `peek` – return the highest-priority element but does not modify the queue (O(1) performance is crucial to many applications of priority queues)

a priority queue can be implemented with a heap or a variety of other methods such as an unordered array.

(min)Heap

A **complete** binary tree T is a min-heap if:

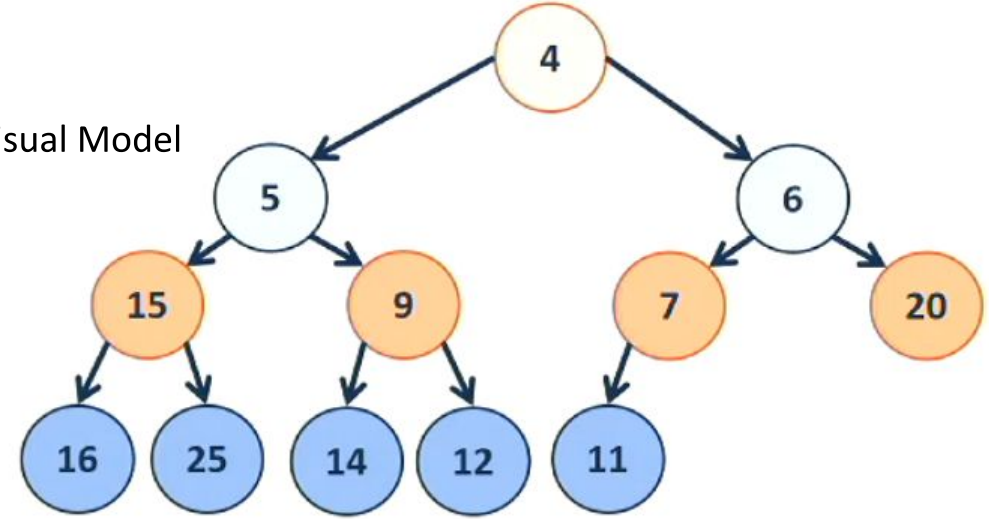
- **T = {}** or
- **T = {r, T_L, T_R}**, where **r** is less than the roots of {T_L, T_R} and {T_L, T_R} are min-heaps.



Heap:

We can use array to represent Heap (do level order traversal):

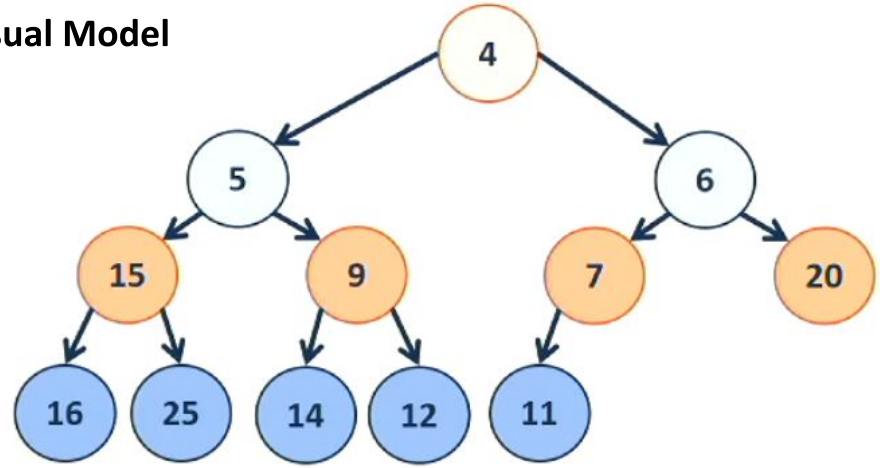
Visual Model



Implementation

X	4	5	6	15	9	7	20	16	25	14	12	11			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Visual Model



	With sentinel node	Without sentinel node
Root	1	0
Left child of i	$2 * i$	$2 * i + 1$
Right child of i	$2 * i + 1$	$2 * i + 2$
Parent of i	$\left\lfloor \frac{i}{2} \right\rfloor$	$\left\lfloor \frac{i-1}{2} \right\rfloor$

X	4	5	6	15	9	7	20	16	25	14	12	11			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

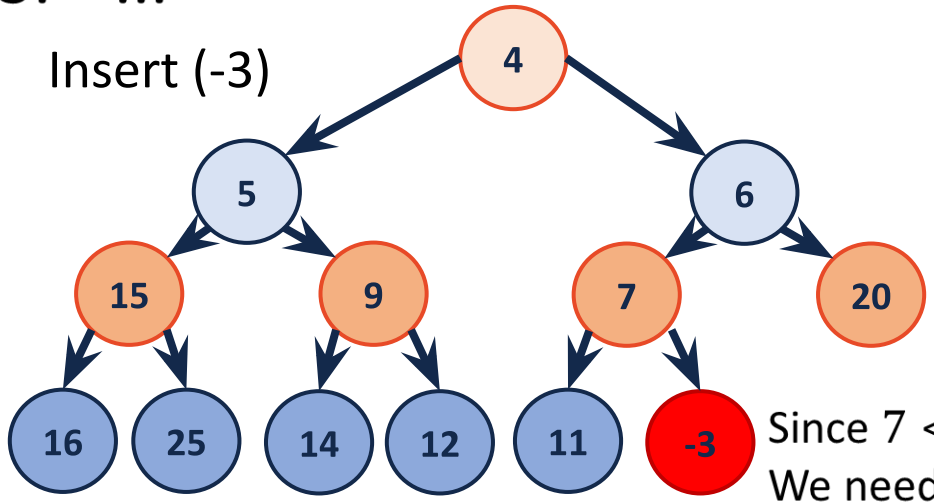
With sentinel node

Insert (key)

1. If array is full, double the size of array;
2. Insert key at the end of the array (index: i)
3. ...

Insert (key)

1. If array is full, double the size of array;
2. Insert key at the end of the array (index: i)
3. ...

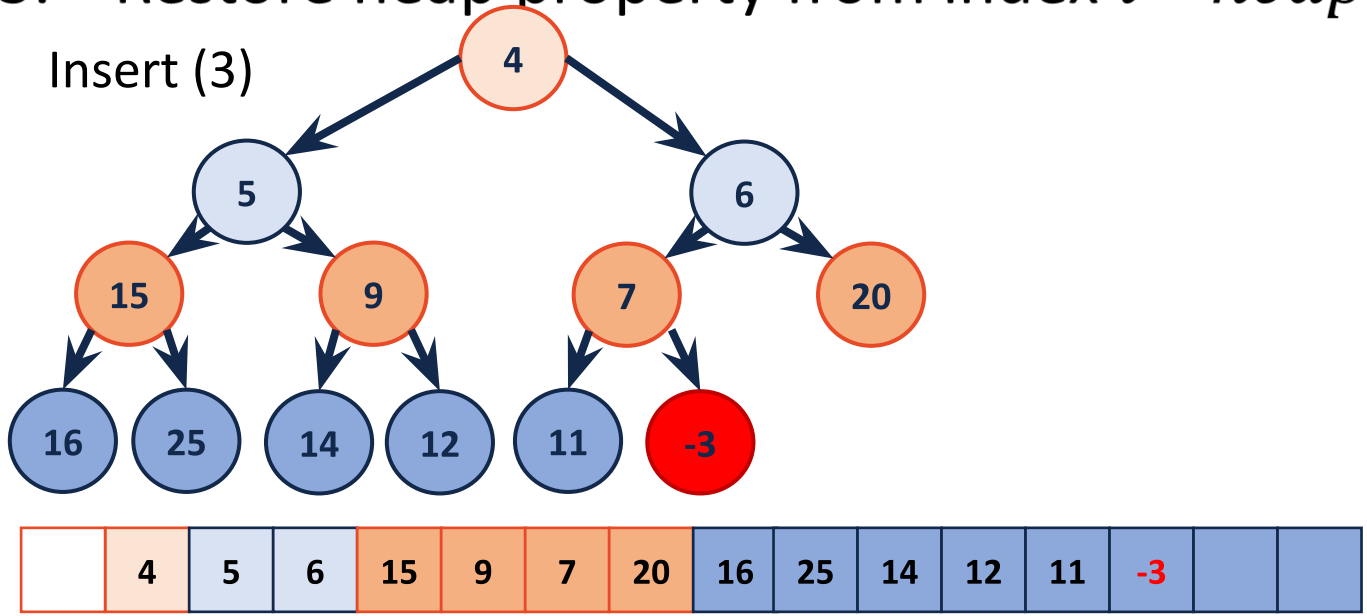


Since $7 < -3$, given tree is not a min Heap anymore.
We need to restore heap property

	4	5	6	15	9	7	20	16	25	14	12	11	-3		

Insert (key)

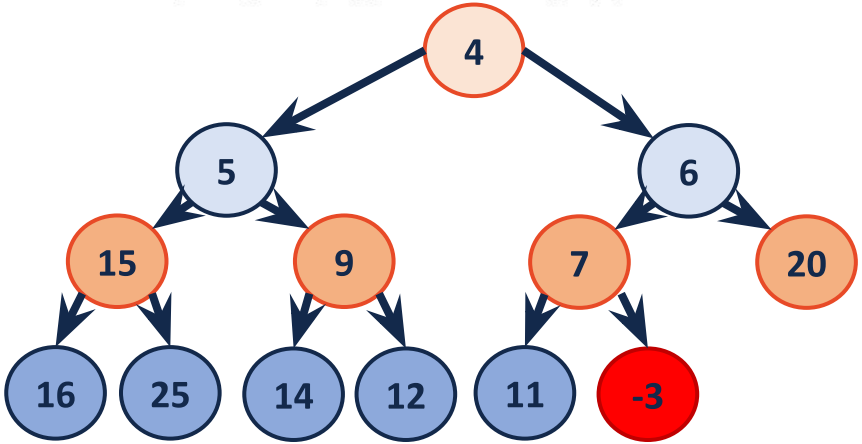
1. If array is full, double the size of array;
2. Insert key at the end of the array (index: i)
3. Restore heap property from index i – $heapifyUp(i)$



While the new element on given index has a smaller value than its parent, swap the element and its parent.

heapifyUp(i)

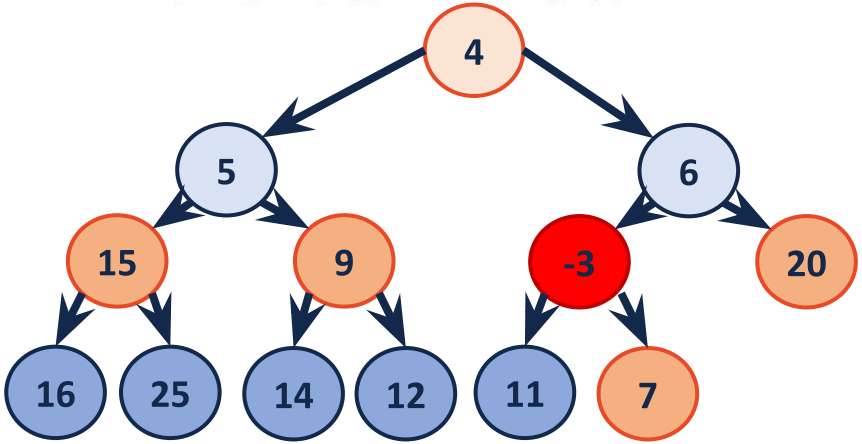
```
if i != rootIndex && A[i] < A[parent(i)]
    swap(i, parent(i))
    heapifyUp(parent(i))
```



While the new element on given index has a smaller value than its parent, swap the element and its parent.

heapifyUp(i)

```
if i != rootIndex && A[i] < A[parent(i)]
    swap(i, parent(i))
    heapifyUp(parent(i))
```

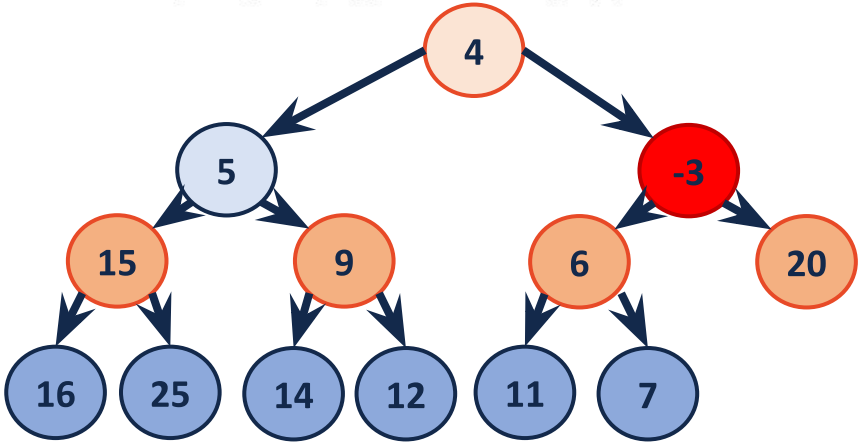


	4	5	6	15	9	-3	20	16	25	14	12	11	7		

While the new element on given index has a smaller value than its parent, swap the element and its parent.

heapifyUp(i)

```
if i != rootIndex && A[i] < A[parent(i)]
    swap(i, parent(i))
    heapifyUp(parent(i))
```

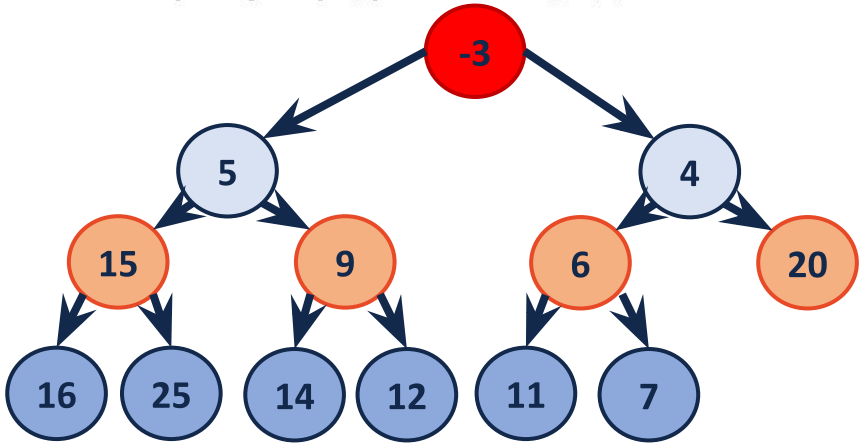


	4	5	-3	15	9	6	20	16	25	14	12	11	7		

While the new element on given index has a smaller value than its parent, swap the element and its parent.

heapifyUp(i)

```
if i != rootIndex && A[i] < A[parent(i)]
    swap(i, parent(i))
    heapifyUp(parent(i))
```



	-3	5	4	15	9	6	20	16	25	14	12	11	7		

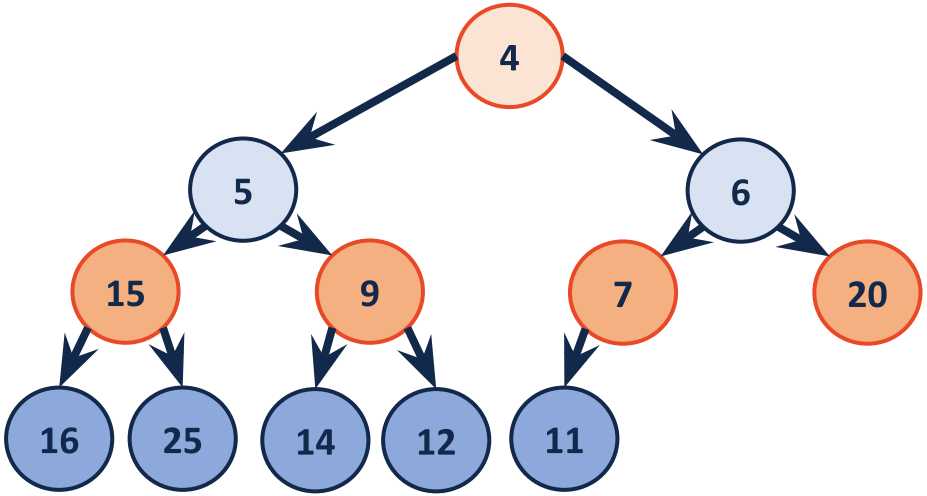
Running time of insert:

1. Resizing array: $O^*(1)$ (we are resizing once in n times)
2. Inserting element after the last element: $O(1)$
3. HeapifyUp: $O(h) = O(\log n)$

Running time of insert: $O(h) = O(\log n)$

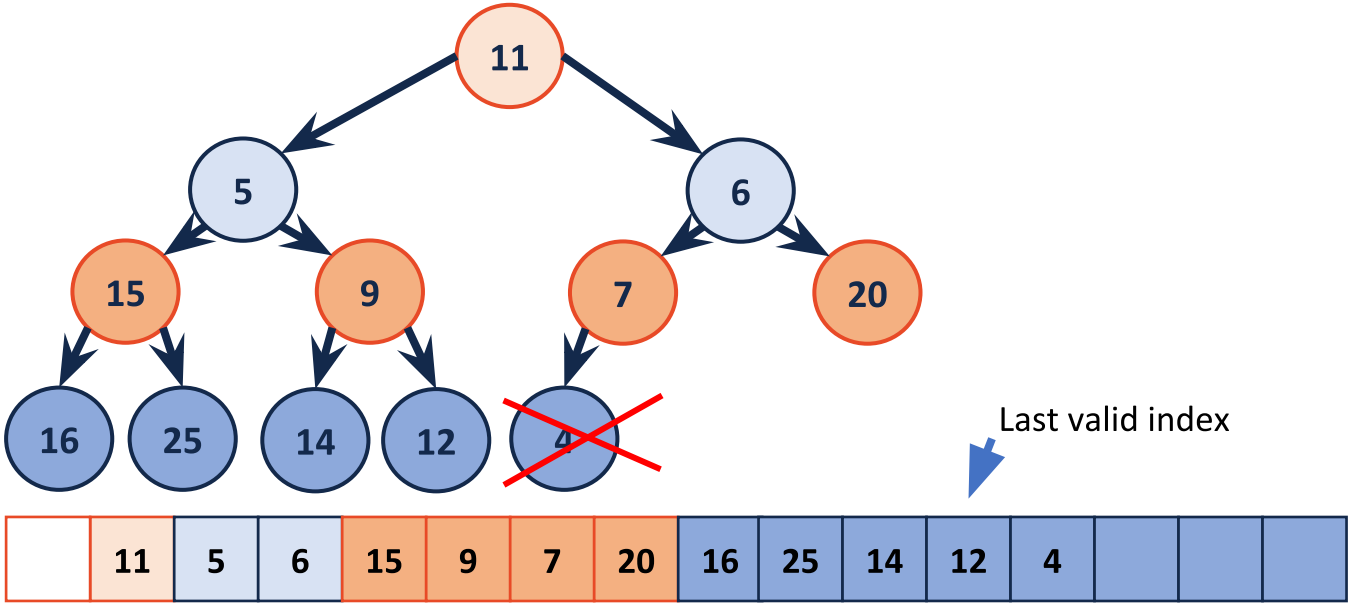
RemoveMin ()

1. Swap the last element for the root
2. return last element (which was the root)
3. HeapifyDown(root)



RemoveMin ()

1. Swap the last element for the root
2. return last element (which was the root)
3. HeapifyDown(root)



HeapifyDown(current)

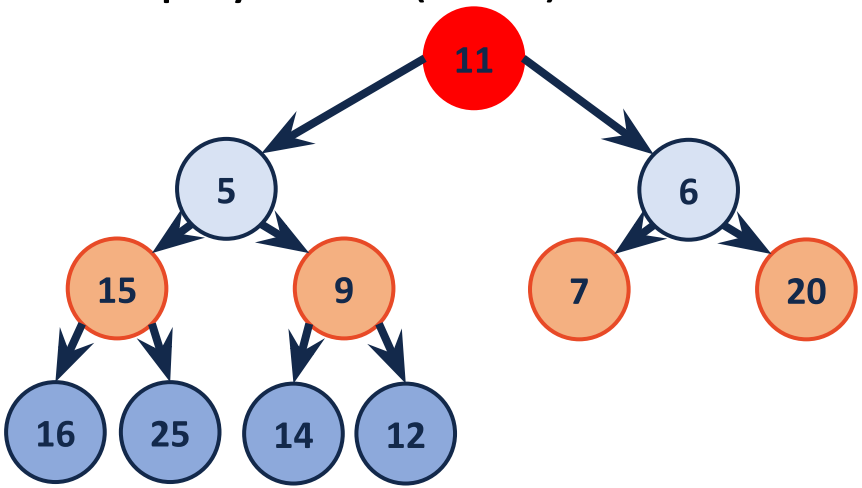
```
If ! isLeaf(current)
```

Find the `min` = index of min child of `current`

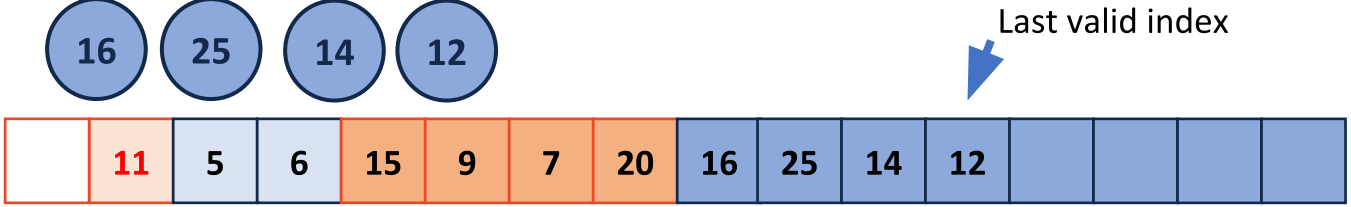
If A[`current`] > A[`min`] child

    swap A[`current`] and A[`min`]

    HeapifyDown(`min`)



//heapifyDown restores heap property only when left subtree and right subtree are already heaps!



HeapifyDown(current)

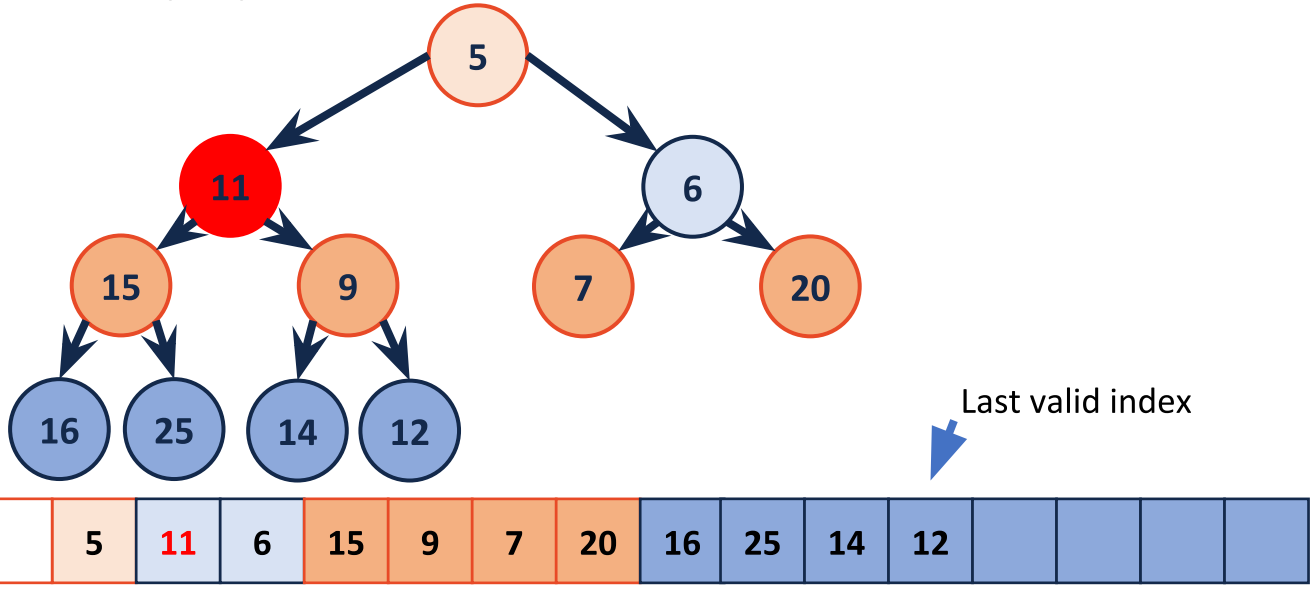
```
If ! isLeaf(current)
```

Find the `min` = index of min child of `current`

If A[`current`] > A[`min`] child

    swap A[`current`] and A[`min`]

    HeapifyDown(`min`)



HeapifyDown(current)

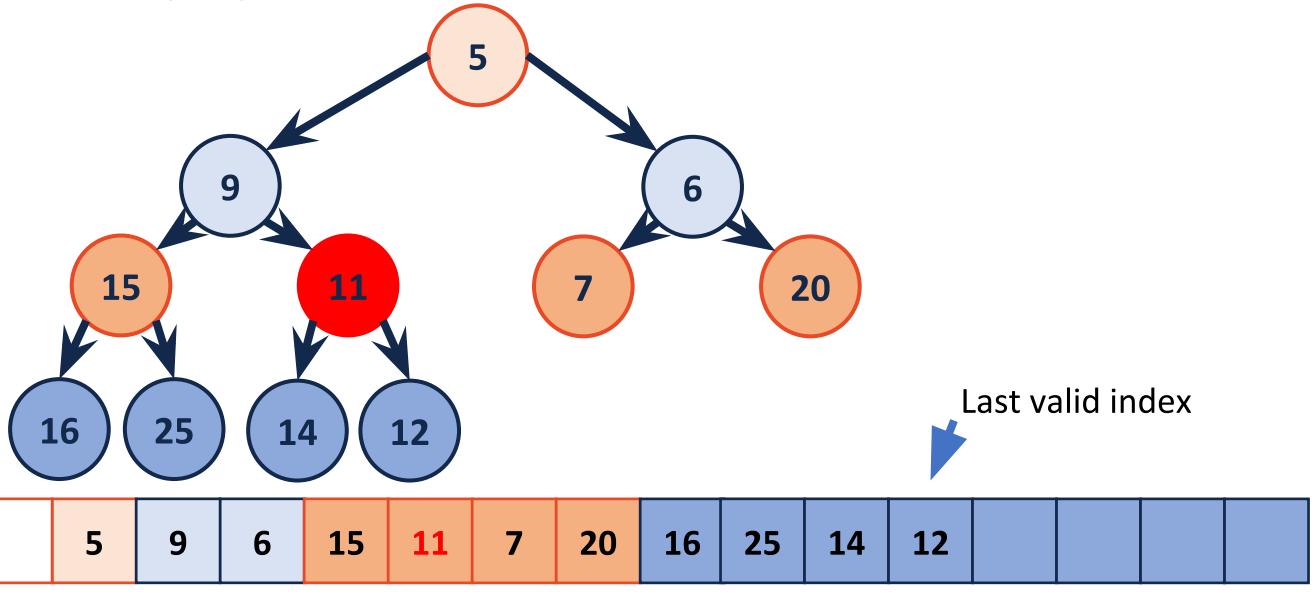
```
If ! isLeaf(current)
```

Find the `min` = index of min child of `current`

If A[`current`] > A[`min`] child

swap A[`current`] and A[`min`]

HeapifyDown(`min`)



HeapifyDown(current)

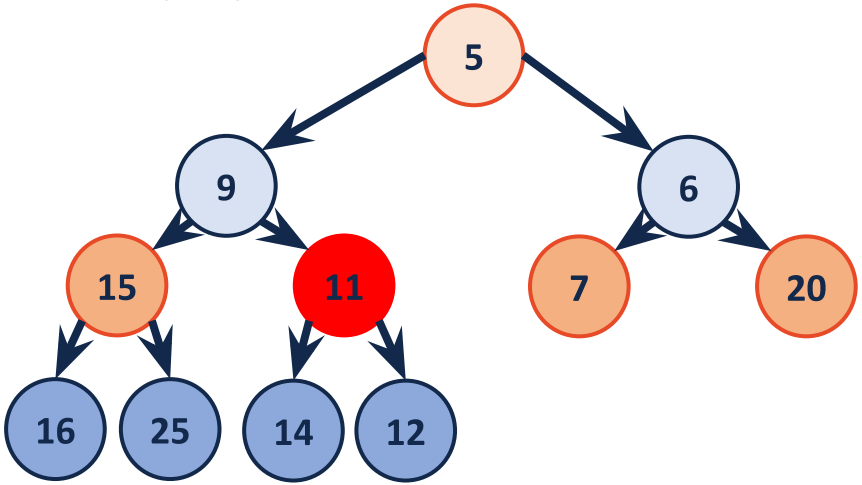
```
If ! isLeaf(current)
```

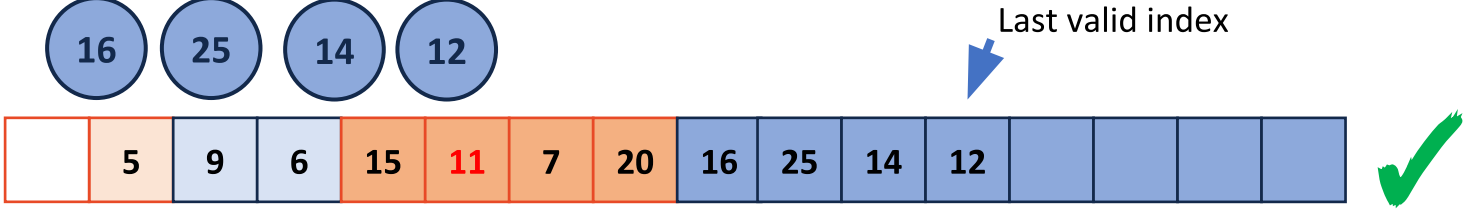
Find the `min` = index of min child of `current`

If A[`current`] > A[`min`] child

swap A[`current`] and A[`min`]

HeapifyDown(`min`)



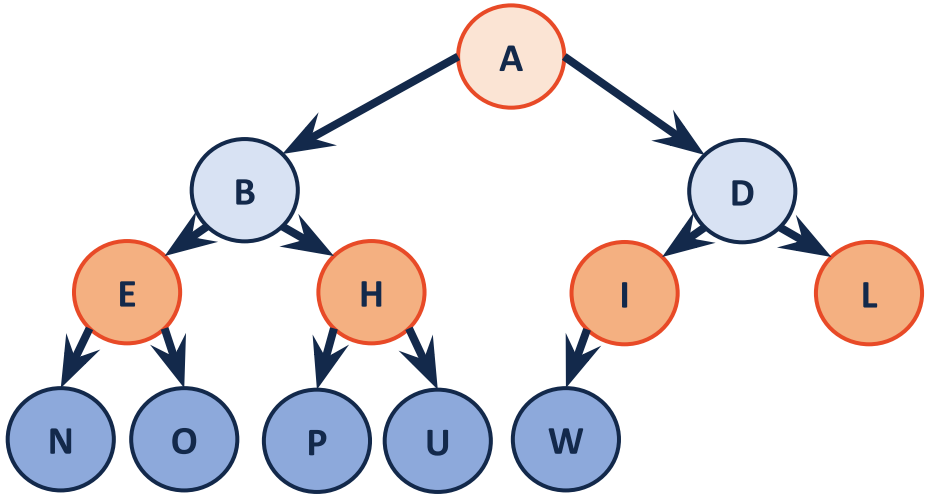


Runnign time of RemoveMin:

1. Swap the last element for the root – $O(1)$
2. return last element (which was the root) – $O(1)$
3. HeapifyDown(root) – $O(h) = O(\log n)$

Running time of remove: $O(h) = O(\log n)$

buildHeap – sorted array

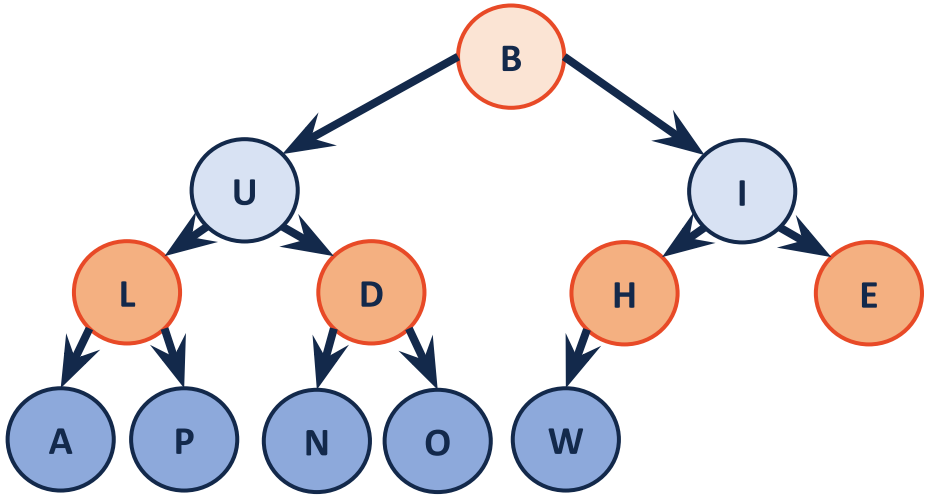


Running time: $O(n \log n)$

buildHeap - heapifyUp

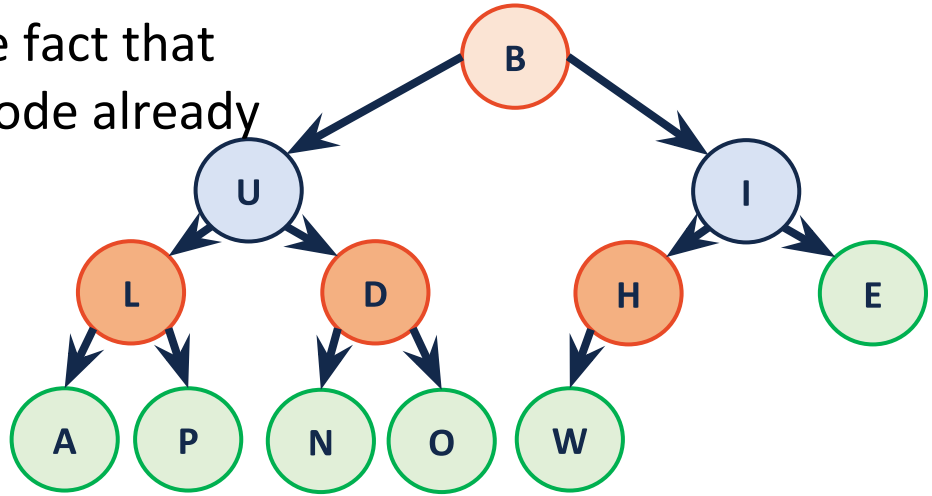
Repeated inserts:
Insert elements one by one

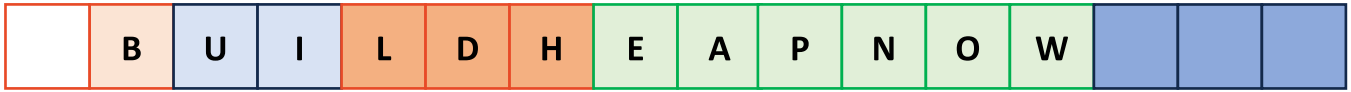
Running time: $O(n \log n)$



buildHeap - heapifyDown

(1) We can take advantage of the fact that subtrees containing only a leaf node already satisfy heap property!





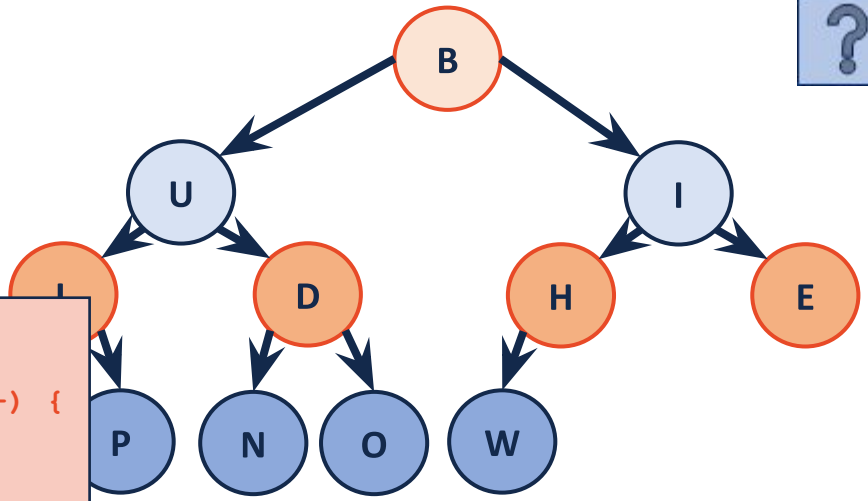
We have to restore heap properties in every subtree started from last element's parent (i), because of (1) we can call HeapfyDown on every element started from index i to root!

buildHeap - heapifyDown

```
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = parent(size); i > 0; i--) {
4      heapifyDown(i);
5    }
6  }
```

Running time: O(n)

1. Sort the array – it's a heap!



2.

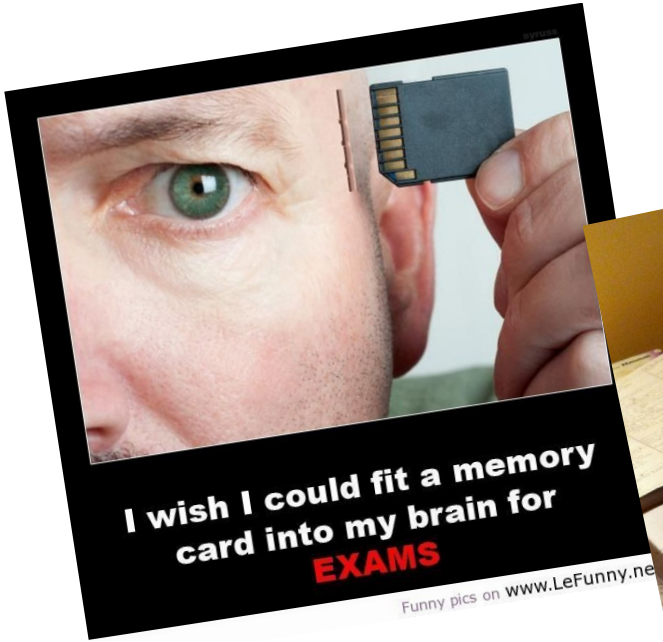
```
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = 2; i <= size_; i++) {
4      heapifyUp(i);
5    }
6  }
```

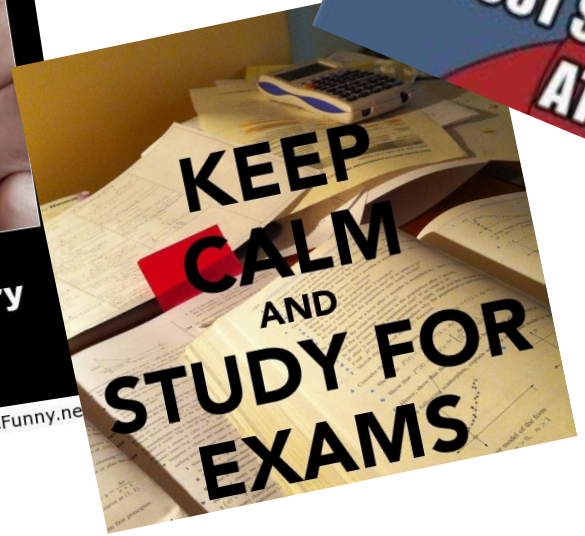
3.

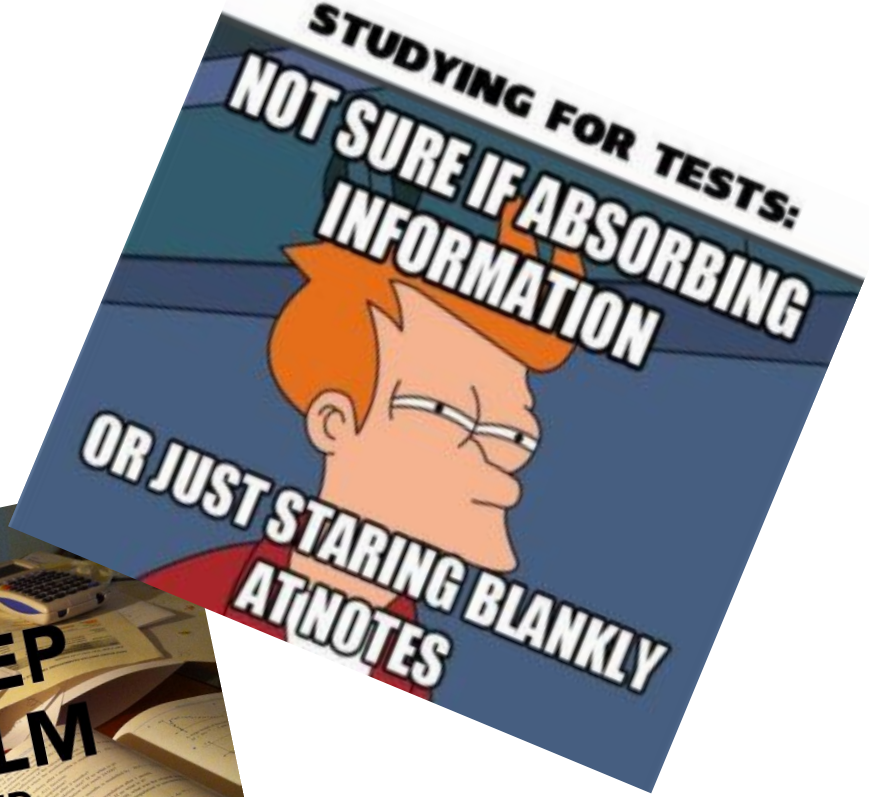
```
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = parent(size); i > 0; i--) {
4      heapifyDown(i);
5    }
6  }
```

	B	U	I	L	D	H	E	A	P	N	O	W			

Good luck!


I wish I could fit a memory card into my brain for EXAMS
Funny pics on www.LeFunny.ne


KEEP CALM AND STUDY FOR EXAMS


STUDYING FOR TESTS:
NOT SURE IF ABSORBING INFORMATION
OR JUST STARING BLANKLY AT NOTES