eecs 281 DATA STRUCTURES AND ALGORITHMS

Lecture 9: Priority Queue
Heap
Trie
Huffman Coding
AVL Trees

What is a Priority Queue

A list of items where each item is given a priority value
- priority values are usually numbers
- priority values should have relative order (e.g., $<$)
- dequeue operation differs from that of a queue or dictionary: item dequeued is always one with highest priority

Method	Description
`enqueue(item)`	insert item by its priority
`Item &dequeuemax()`	remove highest priority element
`Item& findmax()`	return a reference to highest priority element
`size()`	number of elements in pqueue
`empty()`	checks if pqueue has no elements

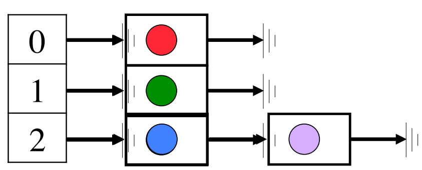
No `search()`!

Priority Queue Examples

Emergency call center:
- operators receive calls and assign levels of urgency
- lower numbers indicate higher urgency
- calls are dispatched (or not dispatched) by computer to police squads based on urgency

Example:
1. Level 2 call comes in
2. Level 2 call comes in
3. Level 1 call comes in
4. A call is dispatched
5. Level 0 call comes in
6. A call is disptached

Priority value used as array index



Priority Queue: Other Examples

Scheduling in general:
- shortest job first print queue
- shortest job first cpu queue
- discrete events simulation (e.g., computer games)

Priority Queue Implementations

Implementation	`dequeuemax()`	`enqueue()`
Unsorted list	$O(N)$	$O(1)$
Sorted list	$O(1)$	$O(N)$
Array of linked list (only for small number of priorities, items with same priority not differentiated)	$O(1)$	$O(1)$
Heap	$O(\log n)$	$O(\log n)$

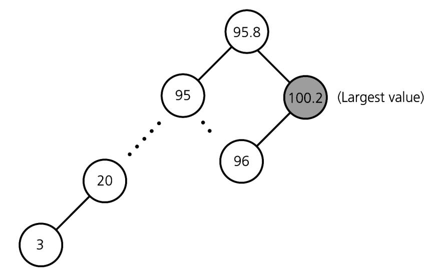
BST as Priority Queue

Where is the smallest/largest item in a BST?

Time complexity of `enqueue()` and `dequeuemax()`:

- `enqueue()`: $O(\log N)$
- `dequeuemax()`: $O(\log N)$

Why not just use a BST for priority queue?

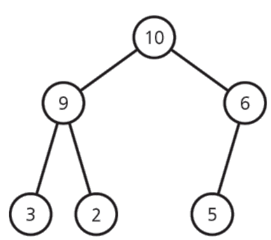


Heaps

A binary heap is a complete binary tree

A non-empty maxHeap T is an ordered tree whereby:

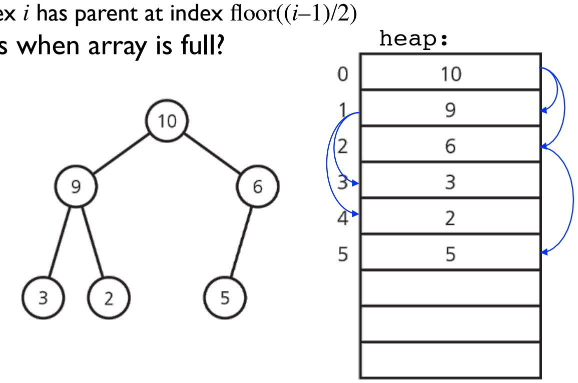
- the key in the root of T is $\geq$ the keys in every subtree of T
- every subtree of T is a maxHeap
- (the keys of nodes across subtrees have no required relationship)
- a `size` variable keeps the number of nodes in the whole heap (not per subtree)
- a minHeap is similarly defined



Heaps Implementation

A binary heap is a complete binary tree

- can be efficiently stored as an array
- if root is at node 0:
  - a node at index i has children at indices $2i+1$ and $2i+2$
  - a node at index i has parent at index $\text{floor}((i-1)/2)$
- what happens when array is full?



maxHeap::dequeuemax()

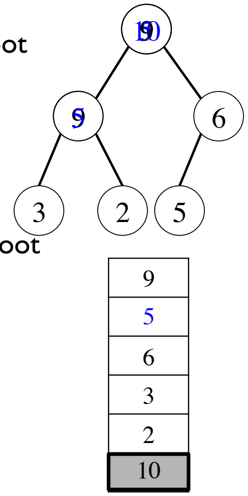
Item at root is the max, save it, to be returned

Move the item in the rightmost leaf node to root
- since the heap is a complete binary tree, the rightmost leaf node is always at the last index
- `swap(heap[0], heap(--size));`

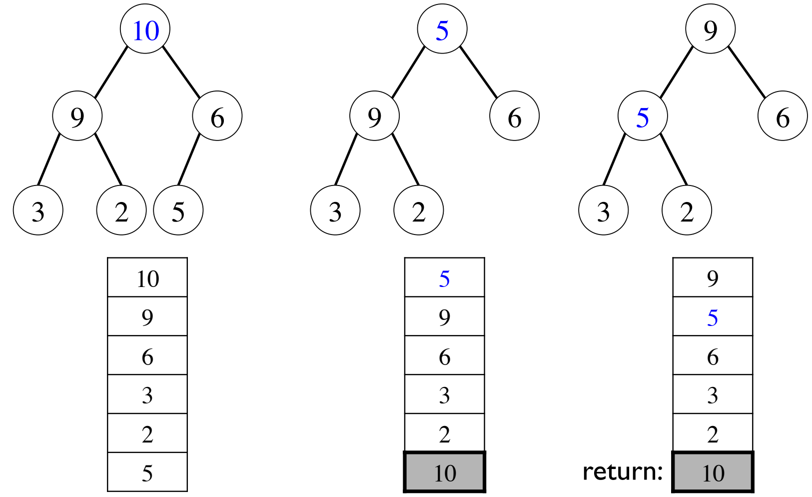
The tree is no longer a heap at this point

Trickle down the recently moved item at the root to its proper place to restore heap property
- for each subtree, recursively, if the root has a smaller search key than either of its children, swap the item in the root with that of the larger child

Complexity: $O(\log n)$



maxHeap::dequeuemax()



Heap Implementation

```
Item maxHeap::
dequeuemax() {
  swap(heap[0], heap[--size]);
  trickleDown(0);
  return heap[size];
}
```

`dequeuemax()`:
- remove root
- take item from end of array and place at root
- use `trickleDown()` to find proper position

Top Down Heapify

```
void maxHeap::
trickleDown(int idx) {
  for (j = 2*idx+1; j <= size; j = 2*j+1) {
   if (j < size-1 && heap[j] < heap[j+1]) j++;
   if (heap[idx] >= heap[j]) break;
   swap(heap[idx], heap[j]); idx = j;
  }
}
```

Pass index (`idx`) of array element that needs to be trickled down

Swap the key in the given node with the largest key among the node's children, moving down to that child, until either
- we reach a leaf node, or
- both children have smaller (or equal) key

Last node is at `heap.size`

maxHeap::enqueue()

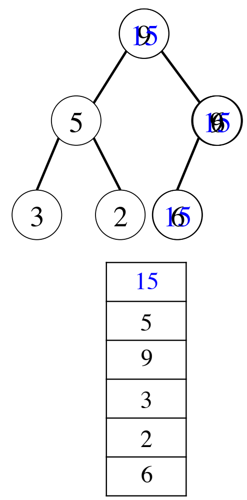
Insert `newItem` into the bottom of the tree

- `heap[size++] = newItem;`

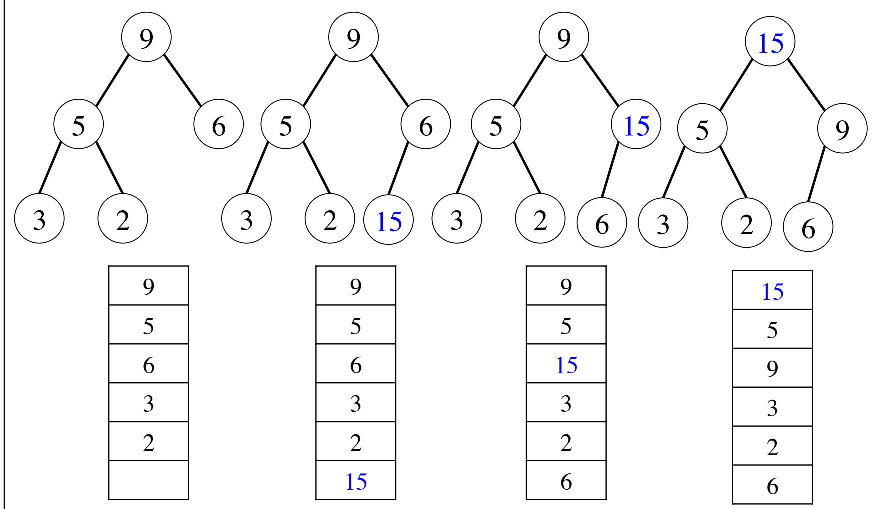
The tree may no longer be a heap at this point

Percolate `newItem` up to an appropriate spot in the tree to restore the heap property

Complexity: $O(\log n)$



maxHeap::enqueue()



Heap Implementation

```
void maxHeap::
enqueue(Item newItem) {
  heap[size++] = newItem;
  percolateUp(size);
}
```

`enqueue()`:

- put item at the end of the priority queue
- use `percolateUp()` to find proper position

Bottom Up Heapify

```
void maxHeap::
percolateUp(int idx) {
  while (idx >= 1 && heap[(idx-1)/2] < heap[idx]){
    swap(heap[idx], heap[(idx-1)/2]);
    idx = (idx-1)/2;
  }
}
```

Pass index (`idx`) of array element that needs to be percolated up

Swap the key in the given node with the key of its parent, moving up to parent until:

- we reach the root, or
- the parent has a larger (or equal) key

Root is at position 0

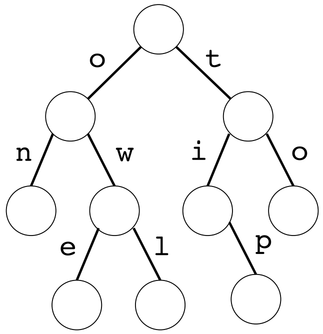
Trie

trie: from retrieval, originally pronounced to rhyme with retrieval, now commonly pronounced to rhyme with “try”, to differentiate from tree

A trie is a tree that uses parts of the key, as opposed to the whole key, to perform search

Whereas a tree associates keys with nodes, a trie associates keys with edges (though implementation may store the keys in the nodes)

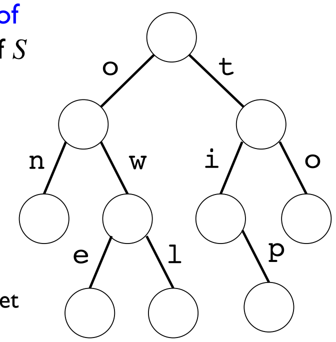
Example: the trie on the right encodes this set of strings: {`on`, `owe`, `owl`, `tip`, `to`}



Trie

For S a set of strings from an alphabet (not necessarily Latin alphabets) where none of the strings is a prefix of another, a trie of S is an ordered tree such that:

- each edge of the tree is labeled with symbols from the alphabet
  - the labels can be stored either at the children nodes or at the parent node
- the ordering of edges attached to children of a node follows the natural ordering of the alphabet
- labels of edges on the path from the root to any node in the tree forms a prefix of a string in S



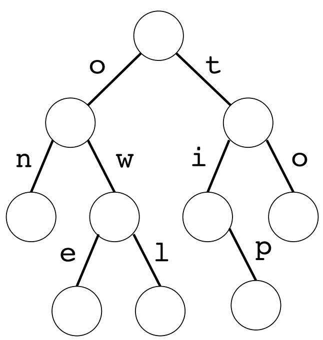
Partial Match

A trie is useful for doing partial match search:

longest-prefix match: a search for “`tin`” would return “`tip`”

- implementation: continue to search until a mismatch

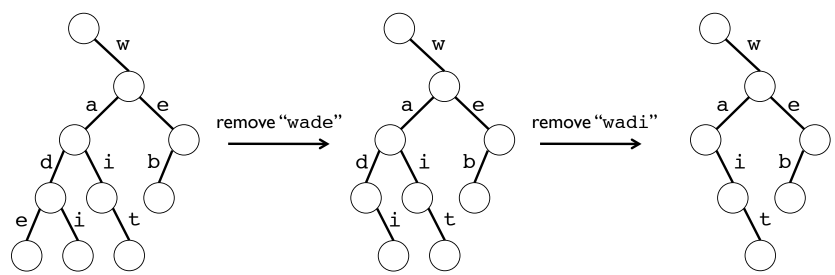
approximate match: allowing for one error, a search for “oil” would return “owl” in this example



Useful for suggesting alternatives to misspellings

Trie Deletion

By post-order traversal, remove an internal node only if it’s also a leaf node, e.g., remove “wade” then “wadi”:



String Encoding

How many bits do we need to encode this example string:

`If a woodchuck could chuck wood!`

- ASCII encoding: 8 bits/character (or 7 bits/character)
  - the example string has 32 characters, so we'll need 256 bits to encode it using ASCII

    4966206120776f6f64…
- There are only 13 distinct characters in the example string, 4 bits/character is enough to encode the string, for a total of 128 bits
- Can we do better (use less bits)? How?

Huffman Codes

In the English language, the characters `e` and `t` occur much more frequently than `q` and `x`

Can we use fewer bits for the former and more bits for the latter, so that the weighted average is less than 4 bits/character?

Huffman encoding main ideas:

1. variable-length encoding: use different number of bits (code length) to represent different symbols
2. entropy encoding: assign smaller code to more frequently occurring symbols

(For binary data, treat each byte as a "character")

Huffman Encoding

The example string:

`If a woodchuck could chuck wood!`

Can be encoded using the following code (for example)

symbol	frequency	code
I	1	11111
f	1	11110
a	1	11101
l	1	11100
!	1	1101
w	2	1100
d	3	101
u	3	100
h	2	0111
k	2	0110
o	5	010
c	5	001
' '	5	000

Resulting encoding uses only 111 bits

- 111111111000011101…

Where do the codes come from?

Prefix Codes

Since each character is represented by a different number of bits, we need to know when we have reached the end of a character

There will be no confusion in decoding a string of bits if the bit pattern encoding one character cannot be a prefix to the bit pattern encoding another

Known as a prefix-free code, or just, prefix code

We have a prefix code if all codes are always located at the leaf nodes of a proper binary trie

How to Construct a Prefix Code?

Minimize expected number of bits over text

- if you know the text, you should be able to do this perfectly
- but top-down allocations are hard to get right

Huffman's insight is to do this bottom-up, starting with the two least frequent characters

Huffman Trie

Better known as Huffman Tree

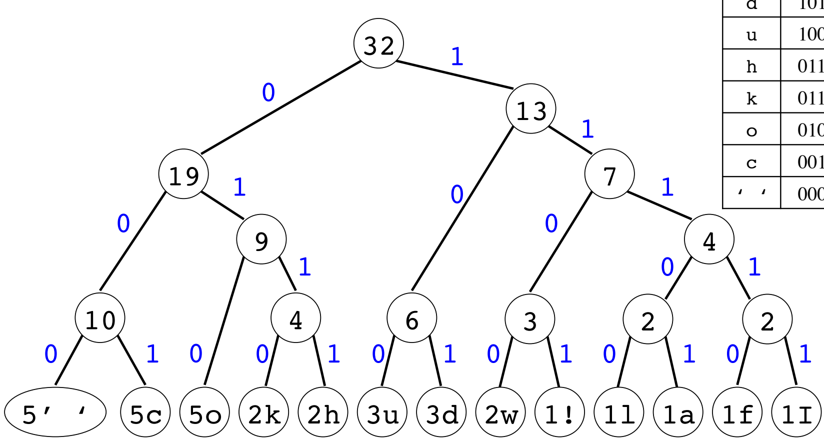
Huffman tree construction algorithm:

- count how many times each symbol occurs
- make each symbol a leaf node, with its frequency as its weight
- pair up the two least frequently occurring symbols (break ties arbitrarily)
- create a subtree of the pair with the root weighing the sum of its children's weights
- repeat until all subtrees have been paired up into a single tree
- encode each symbol as the path from the root, with a left represented as a 0, and a right a 1

Huffman Trie Example

I	11111
f	11110
a	11101
l	11100
!	1101
w	1100
d	101
u	100
h	0111
k	0110
o	010
c	001
' '	000

`If a woodchuck could chuck wood!`



Characteristics of Huffman Trees

All symbols are leaf nodes by construction, hence no code is a prefix of another code

More frequently occurring symbols at shallower depth, having smaller codes

Implementation:

- You are given a list of items with varying frequencies
  - need to repeatedly choose two that currently have the lowest frequency
  - need to repeatedly place sum of the above back into the list
- How would you implement the algorithm?

Encoding Time Complexity

Running times, string length: n, alphabet size: m

- frequency count: $O(\ )$
- Huffman tree construction: $O(\ )$
- Total time: $O(\ )$

To decode, we need the code table, so the code table must be stored with the encoded string

How to store/communicate the code table?

I	11111
f	11110
a	11101
l	11100
!	1101
w	1100
d	101
u	100
h	0111
k	0110
o	010
c	001
' '	000

Code Table Encoding

The Huffman code for any particular text is not unique

For example, all three sets of codes in the table are valid for the example string

The last column can be compressed into:
`3' 'cdou4!hkw5aflI`

sym	freq	code1	code2	code3
' '	5	000	001	000
c	5	001	010	001
d	3	101	100	010
o	5	010	000	011
u	3	100	101	100
!	2	1101	0110	1010
h	2	0111	1101	1011
k	2	0110	1110	1100
w	1	1100	0111	1101
a	1	11101	11100	11100
f	1	11110	11101	11101
l	1	11100	11111	11110
I	1	11111	11110	11111

Code Table Encoding

The last column was not created from a Huffman tree directly

The Huffman tree is used only to determine the code length of each symbol, then:

1. order symbols by code length
2. starting from all 0s for the shortest length code
3. add 1 to the code for each subsequent symbol
4. when transitioning from code of length k to code of length $k+1$, as determined by the Huffman trie, add 1 to the last length-k code and use it as the prefix for the first length $k+1$ code
5. set the $k+1^{st}$ bit to 0 and continue adding 1 for each subsequent code

Resulting code table has one encoding for each symbol and is prefix-free

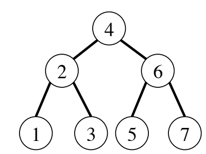
Worst-Case BST Performance

Exercise:

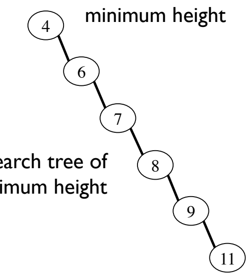
- insert 4, 2, 6, 3, 7, 1, 5
- remove 2, insert 8, remove 5, insert 9, remove 1, insert 11, remove 3

Moral: even a balanced tree can become unbalanced after a number of insertions and removals

Why is a balanced tree desirable?



a search tree of minimum height



a search tree of maximum height

Balanced Search Trees

What are your requirements to call a tree a balanced tree?

Would you require a tree to be perfect to call it balanced?

- a perfect binary tree of height h has exactly $2^{h+1} - 1$ internal nodes
- so by this criterion, only trees with $1, 3, 7, 15, 31, 63, \ldots$ internal node can be balanced
- too restrictive

Balanced Search Trees

Need another definition of "balance condition"

Want the definition to satisfy the following criteria:

1. height of tree of n nodes $= O(\log n)$
2. balance condition can be maintained efficiently:, e.g., $O(1)$ time to rebalance a tree

Several balanced search trees, each with its own balance condition: AVL trees, B-trees, 2-3 trees, 2-3-4 (a.k.a. 2-4) trees, red-black trees, AA-trees, treaps

AVL Trees

Adel'son-Vel'skii & Landis (AVL) tree

- AVL trees' balance condition:
  - an empty binary tree is AVL balanced
  - a non-empty binary tree is AVL balanced if both its left and right sub-trees are AVL balanced and differ in height by at most 1
- satisfies criterion 1: balance condition can be proven to maintain a tree of height $\Theta(\log n)$

  $\Rightarrow$ search is guaranteed to always be $O(\log n)$ time!
- satisfies criterion 2: requires far less work than would be necessary to keep the height exactly equal to the minimum
  - we'll see how an AVL tree keeps its balance condition and how this is an $O(1)$ operation

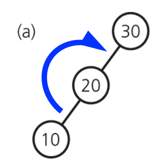
    $\Rightarrow$ both insert and remove are also guaranteed to be $O(\log n)$ time!

AVL Tree ADT

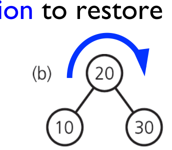
Search, insert, and remove all works exactly the same as with BST

However, after each insertion or deletion

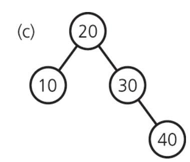
- must check whether the tree is still balanced, i.e., balance condition still holds
- if the tree has become unbalanced, "re-balance" the tree by performing one rotation to restore balance



(a) an unbalanced AVL tree



(b) a balanced AVL tree after rotation



(c) a balanced AVL tree after insertion

Tree Rotations

The rotation operation: interchange the role of a parent and one of its children in a tree

- while still preserving the BST ordering among the keys in the nodes

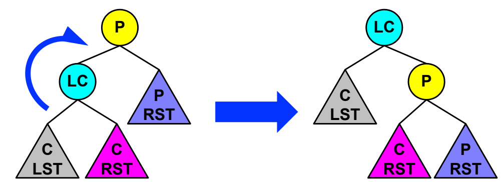
Two directions of rotations:

- right rotation: parent becomes right child of its left child
- left rotation: parent becomes left child of its right child

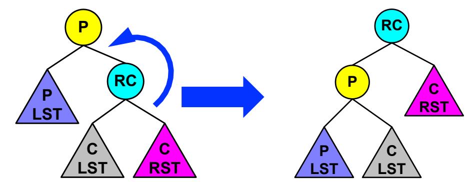
Tree Rotations

To preserve the BST ordering:

- right rotation: the right link of the left child becomes the left link of the parent; parent becomes right child of the old left child

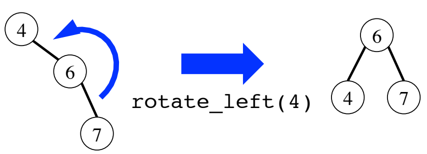


- left rotation: the left link of the right child becomes the right link of the parent; parent becomes left child of the old right child

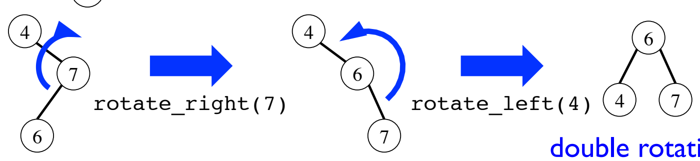


Tree Rotations

What rotations would you need to balance the following two trees:



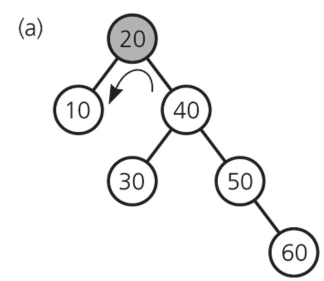
single rotation

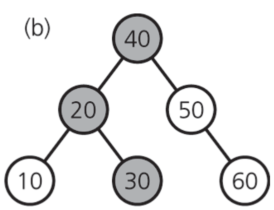


double rotation

Rotation is a local change involving only three links and two nodes ⇒ can be done in $O(1)$ time

Example: Single Rotation

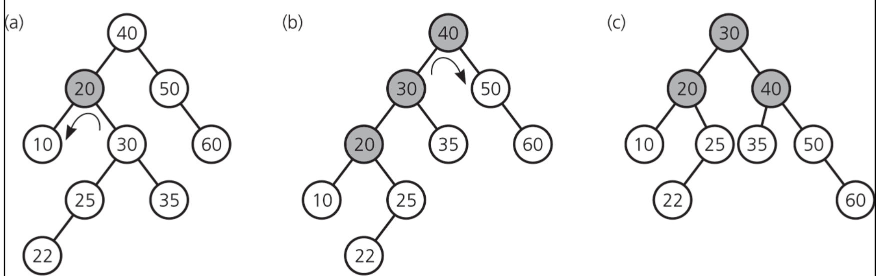




(a) an unbalanced AVL tree

(b) a balanced AVL tree after a single left rotation

Example: Double Rotation



(a) an unbalanced AVL tree, (b) left rotated, and (c) right rotated, AVL balanced restored after a double rotation

Restoring AVL Balance: Details

Let T_l and T_r be the left and right subtrees of a tree rooted at node T

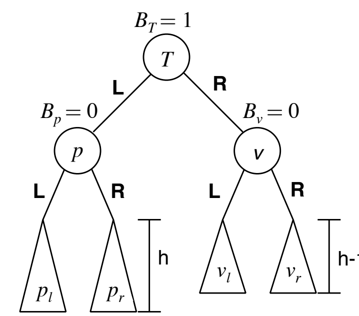
Let h_l be the height of T_l and h_r the height of T_r

Define the balance factor (B_T) of node T as: $B_T = h_l - h_r$

AVL trees' balance condition:

The tree rooted at node T is AVL balanced iff $|B_T| \leq 1$

- if T_l is deeper, then $B_T > 1$
- if T_r is deeper, then $B_T < -1$



AVL balanced at T, p, and v

Unbalanced AVL Trees

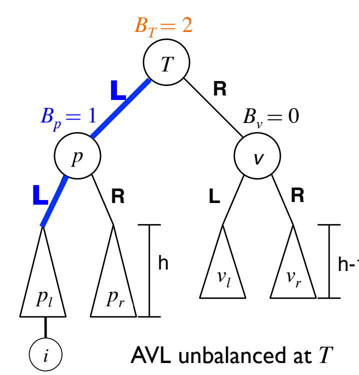
When an AVL tree becomes unbalanced, there are four cases to consider depending on the direction of traversal from the unbalanced node to the tallest grandchild:

1. Left-Left (LL)
2. Right-Right (RR)
3. Left-Right (LR)
4. Right-Left (RL)

Unbalanced AVL Tree

For example,

LL: a new node is added to subtree p_l, causing $B_p = 0 \rightarrow B_p = 1 \Rightarrow B_T = 2$, which violates the AVL balance condition, and the tree rooted at T is now unbalanced



AVL unbalanced at T

From T, to get to the tallest grandchild is by doing a Left-Left traversal (balance factor positive positive, B++)

Restoring Balance

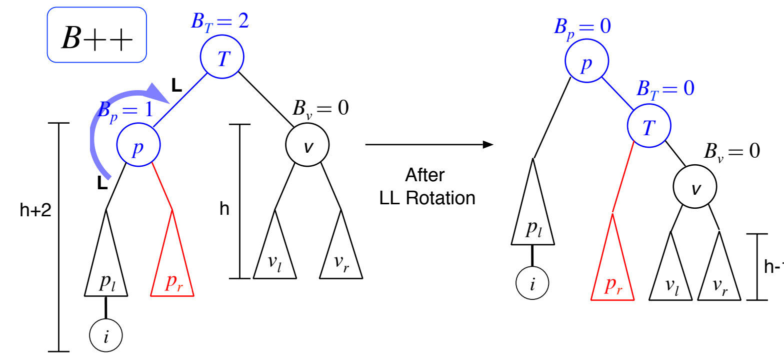
If the subtree rooted at node T has become unbalanced ($|B_T| > 1$), to restore balance at node T, rotate counter to the direction of traversal to tallest grandchild:

- for an LL traversal ($B{+}{+}$), do a single right rotation
- for an RR traversal ($B{-}{-}$), do a single left rotation
- for an LR traversal ($B{+}{-}$), do a double rotation, with the first rotation countering the last traversal, in this case, we do left rotation, then right rotation
- for an RL traversal ($B{-}{+}$), do a right-left double rotation

Must retain BST property at all times

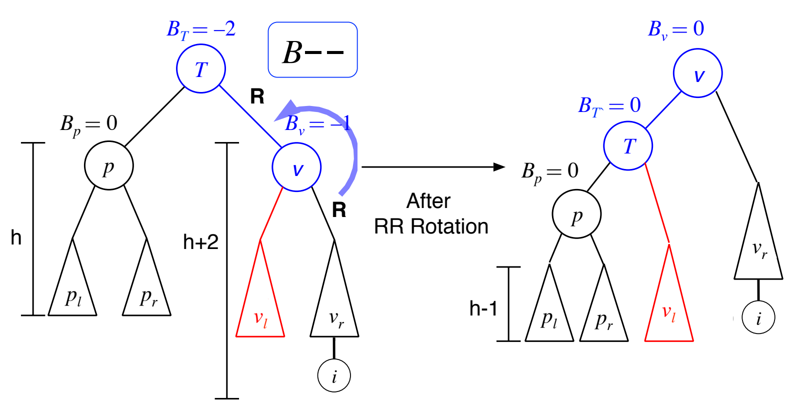
LL: Single Right Rotation

Make T the right child of p and p_r the left child of T



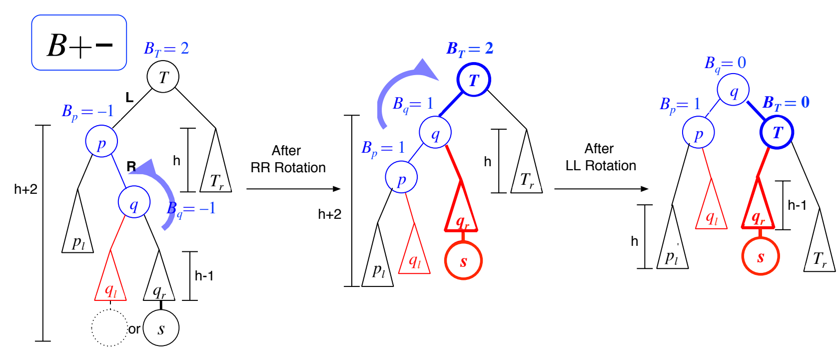
RR: Single Left Rotation

Make T the left child of v and v_l the right child of T



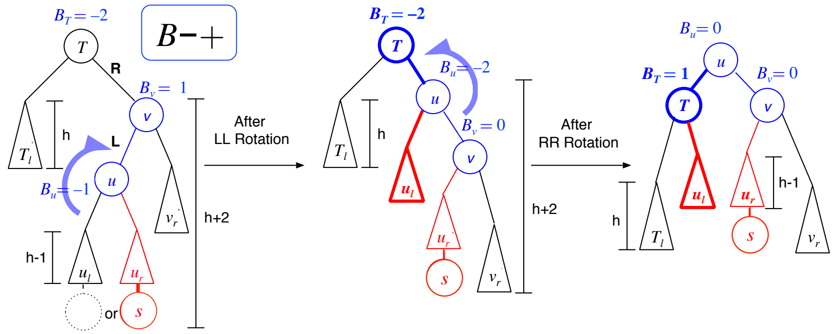
LR: Double Left-Right Rotation

Do a left rotation on q, then a right rotation



RL: Double Right-Left Rotation

Do a right rotation on u, then a left rotation



Exercise

Insert into an AVL tree: $42, 35, 69, 21, 55, 83, 71$

Compute the balance factors

Is the AVL tree balanced?

Insert $95, 18, 75$

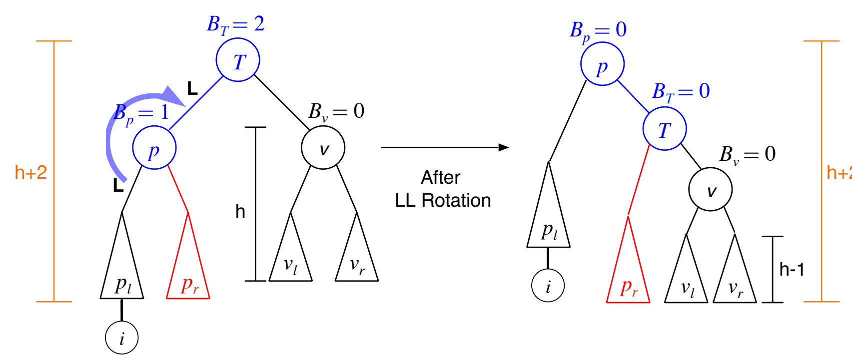
Rebalance Can be Done in $O(1)$

When an AVL tree becomes unbalanced, exactly one single or double rotation is required to balance the tree

- when adding a node, only the height of nodes in the access path between the root and the new node can be changed
- if adding a node doesn't change the height of node i in the access path, no rotation is needed at i or its ancestors
- if height of i changes, it can either:
  - remain balanced: no rotation needed at i, but may be necessary at its parent node (see LL figure, for example)
  - become unbalanced: after one rotation, the height of (sub)tree previously rooted at i is the same as before insertion! so, none of its ancestors needs to be rebalanced

Height Restored After Rotation

After one rotation, the height of (sub)tree previously rooted at T is the same as before insertion!



AVL Removal

First remove the node as with BST

Then update the balance factors of the node's ancestors in the access path and rebalance as needed