COMPLEXITIES

Big-O Notation

Given a function F(n), we say g(n) is an asymptotic upper bound of f(n), denoted as f(n) = O(g(n)), if there exists a constant c > 0, and a positive integer n_0 such that f(n) ≤ c * g(n) for all n ≥ n_0.

Recurrence relations

1. $T(n) = T(n-1) + O(1) = O(n)$
2. $T(n) = T(n/2) + O(1) = O(\log n)$
3. $T(n) = T(n-1) + O(n) = O(n^2)$
4. $T(n) = T(n-1) + O(n^k) = O(n^{k+1})$
5. $T(n) = 2T(n/2) + O(n) = O(n\log n)$
6. $T(n) = T(n/2) + O(n) = O(n)$
7. $T(n) = 2T(n/2) + O(1) = O(n)$
8. $T(n) = 2T(n-1) + O(1) = O(2^n)$

^O(x) represents growth of each step

SEARCHING

Binary Search

Takes O(logn)

Preconditions:
- array is of size n
- array is sorted

Invariant:
- A[begin] ≤ key ≤ A[end]
- (end – begin) <= $n/2^k$
- $n/2^k = 1$ → k = logn

```
int search (A, key, n)
begin = 0;
end = n – 1;
while begin < end
   mid = begin+(end-begin)/2;
   if key <= A[mid]
      end = mid;
   else begin = mid + 1;
return (A[begin] == key?begin:-1)
```

SORTING

BubbleSort

```
repeat n times:
for j = 1 to n-1,
      if A[j] > A[j+1] then
      swap(A[j], A[j+1])
```

- Invariant:
- At the end of iteration i, the biggest j items are correctly sorted in the final j positions of the array.
- Best: O(n) A.S., Worst: $O(n^2)$ R.S., Stable

InsertionSort

```
for j = 2 to n,
   key = A[j]
   i = j – 1
   while (i > 0) and (A[i] > key)
     A[i + 1] = A[i]
     i = i - 1
   A[i + 1] = key
```

- Invariant:
- At the end of iteration j, the first j items in the array are in sorted order
- Best: O(n) A.S., Worst: $O(n^2)$ R.S., Stable

SelectionSort

```
for j = 1 to n-1,
      find minimum element A[j]
      swap(A[j], A[k])
```

- Invariant:
- At the end of iteration j, the smallest j items are correctly sorted in the first j positions of the array.
- Best: O(n) A.S., Worst: $O(n^2)$ R.S., Not stable

MergeSort

```
MergeSort(A, n)
if (n = 1) then return;
else:
   X = MergeSort(A[1...n/2, n/2)
   Y = MergeSort(A[n/2+1..n, n/2)
return Merge(X, Y, n/2)
```

1.Divide: split array into two halves
2.Recurse: sort the two halves
3.Combine: merge the two sorted halves

- Merge: O(n) = cn
- In each iteration, move one element to the final list
- Recurrence relation no. 5
- Best: O(nlogn), Worst: O(nlogn), Stable

QuickSort

1.Divide: partition into sub-arrays around pivot x
2.Conquer: recursively sort sub-arrays
- Invariant:
- For every i < low, B[i] < pivot
- For every j > high, B[j] > pivot
- Run-time of partition: O(n)
- Best: O(nlogn), A.S.
- Worst: $O(n^2)$, all same
- Not stable

QuickSelect

- O(n): to find the kth smallest element
- after partitioning, the partition is always in the correct position

TREES

BST

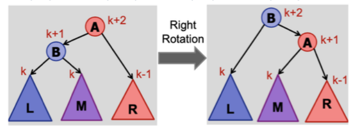
- a BST is either empty, or a node pointing to 2 BSTs
- height = O(logn)
- `h(v)=max(h(v.left),h(v.right))+1`
- `searchMax`: O(h) recurse until rightTree is null
- `searchMin` : O(h) recurse until leftTree is null
- worst case running time of search: O(n)
- order of insertion determines shape
- In order traversal: left child, parent, right child, takes O(n)
- Pre order traversal: parent, left child, right child
- Post order traversal: left child, right child, parent
- Level order traversal: root, left to right
- `successor`: key not in tree
  - if result > key, return result
  - if result <= key, successor(result)
- `successor`: has no right child
  - traverse up the tree (while child is parent's rightTree) to return parent
- `delete O(h)`:
  - no child → delete
  - 1 child → parent point to child
  - 2 children → find successor (which has at most 1 child)

AVL Tree

- node cv is height-balanced if children's height differ by ≤ 1
- BST is height balanced if every node is height-balanced
- maximum height: h < 2log(n)

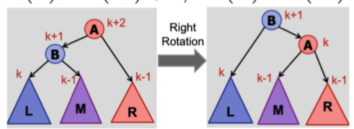
[case 1] B is **balanced: right-rotate**

$h(L) = h(M), \quad h(R) = h(M) - 1$



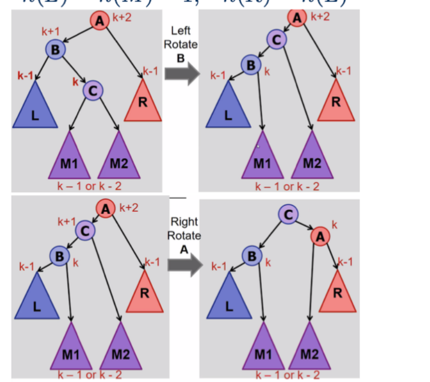
[case 2] B is **left-heavy: right-rotate**

$h(L) = h(M) + 1, \quad h(R) = h(M)$



[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**

$h(L) = h(M) - 1, \quad h(R) = h(L)$

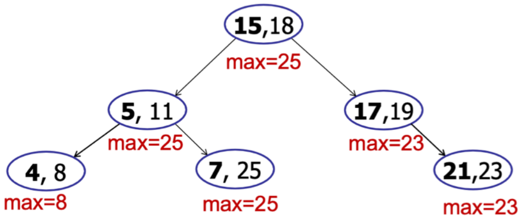


Rebalancing with rotation
- costs O(1)
- maintain ordering of keys
- left heavy: left sub-tree has larger height than right subtree
- right heavy: opposite
- A is the lowest unbalanced node
- Invariants:
- siblings height difference < 2
- parent's height > child's height

- if vi is out of balance and left heavy
  - v.left is balanced → right-rotate(v)
  - v.left is left-heavy → right-rotate(v)
  - v.left is right-heavy → left-rotate(v.left) & right-rotate(v)
- worst case: 2 rotations after insertion (on the lowest out of balance node)
- worst case: logn rotations after delete (at every step walking up tree)
- augmented with weight of each node

TRIES
- faster O(L) than a tree O(Lh)
- space to store: O(size of text * overhead)

INTERVAL TREES



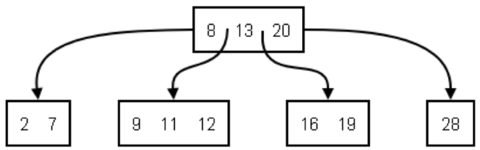
- augment with max interval of node
- maintain max as augmentation during rotation
- all-overlaps algorithm (all intervals that overlap with point):
- search for interval, add to list, delete, repeat until no more
- add all intervals back to the tree
- O(k log n) for k overlapping intervals

- search: O(logn)
  - value is in root interval, return
  - value > max(left subtree), recurse right
  - else recurse left (only when can't go right)

```
c == root;
while (c != null and x is not in
c.interval)
    if (c.left == null) then
      c = c.right;
    else if (x > c.left.max)
      c = c.right;
    else
      c = c.left;
return c.interval;
```

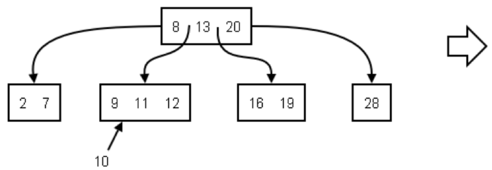
(A,B)-TREES
- $2 \le a \le (b + 1)/2$
- Each internal node except the root has at least a children and at most b children
- Root has at most b children
- B-Tree: a = ceil(b/2)

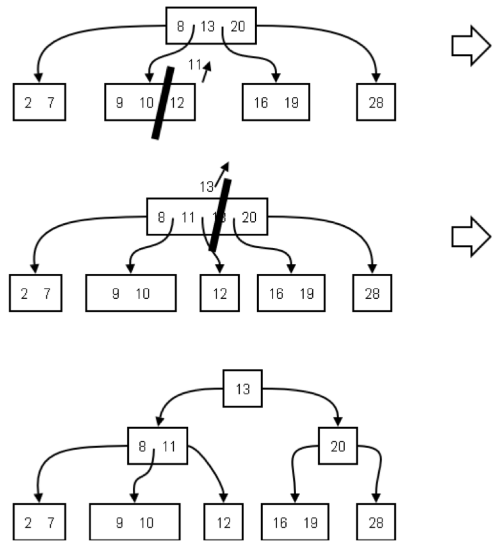
An example of a (2,4)-tree:



- Invariants:
- Siblings u and v:
  - |deg(u) – deg(v)| ≤ b
  - |height(u) – height(v)| < 1
- If node u has height h, then subtree rooted at u contains at least a^h nodes
- Insertion:
- Add into the leaf node → (if overflow) split and propagate middle item

Example: to insert 10 into the tree above:





ORTHOGONAL RANGE SEARCHING
One dimensional range queries:
- Use BST, store all points in leaves, each internal node stores the MAX of any leaf in left sub-tree

```
FindSplit(low, high)
v = root;
done = false;
while !done {
    if (high<=v.key) then v v.left;
    else if (low>v.key) then
v=v.right;
    else (done = true);
}
return v
```

- Find split node, do left then right traversal
  - split node: highest node where search includes both left and right subtrees

```
LeftTraversal(v, low, high)
if (low <= v.key) {
   all-leaf-traversal(v.right);
   LeftTraversal(v.left,low,high);
} else {
   LeftTraversal(v.right,low,high);
}
```

- Invariants:
- search interval for a left-traversal at node v includes the maximum item in the subtree rooted at v
- Split node finding: O(log n)
- (1) output all right sub-tree and recurse left: O(k), k is number of items found
- (2) recurse right: O(log n)
- Total query time: O(k + log n)
- Build tree time: O(n log n)
- Space complexity: O(n)
- To know how many points are in the range: increment count instead of all-leaf-traversal

Two dimensional range trees
- Build an x-tree using only x-coords
- For every node in the x-tree, build a y-tree out of nodes in subtree using only y-coord
- Query time: $O(\log^2 n + k)$
  - O(log n) to find split node
  - O(log n) recursing steps
  - O(log n) y-tree searches of cost O(log n)
  - O(k) enumerating output
- Space complexity: O(n log n)
  - Each point appears in at most one y-tree per level, O(log n) levels
- Building the tree: O(n log n)

PRIORITY QUEUE/HEAP SORT
- Sorted arr: insert O(n), extractMax O(1)
- Unsorted arr: insert O(1), extractMax O(n)
- AVL: insert O(logn), extractMax O(logn)
- Heap properties:
  - priority of parent >= priority of child
  - complete binary tree: all nodes as far left
  - max height: floor(log n), height O(log n)
- Insert (at leaf)/ increase key → bubble up
- Decrease key → bubble down, leftwards
- Delete: swap key with last, remove last, bubble key down
- Heap to sorted array: O(n log n)
- Unsorted list to heap: O(n)
- Heapsort: O(n log n) (faster than mergesort, slower than quicksort)