int BinarySearch(A, key, n)

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

Not just for searching arrays:
- Assume a complicated function:
  int complicatedFunction(int s)
- Assume the function is always increasing:
  complicatedFunction(i) < complicatedFunction(i+1)
- Find the minimum value j such that:
  complicatedFunction(j) > 100

PreCond Sorted, size n **PostCond** A[begin]=key

Int Peak (A , n)

```
if A[n/2+1] > A[n/2]  (divide&conquer – search from mid)
      return Peak (A[n/2+1..n] , n/2)
else if A[n/2–1] > A[n/2]
      return Peak (A[1..n/2 -1] , n/2 )
else return n/2;
```

Key property

1. If we recurse in the right half, then there exists a peak in the right half

Invariant

2. Every peak in[begin, end] is a peak in [1, n].
3. There exists a peak in the range [begin, end]

PreCond Peak in [begin,end],

PostCond peak at a[n/2]

Runtime : T(n) = T(n/2) + 0(1) → O(logn)

BubbleSort(A, n)

```
repeat (until no swaps):
for j ← 1 to n-1
   if A[j] > A[j+1]
      swap(A[j], A[j+1])
```

J biggest **CORRECTLY** sorted at j end pos after j iteration

SelectionSort(A, n)

```
for j← 1 to n-1:
   find minimum element A[j] in A[j..n] (time = n - j)
   swap(A[j], A[k])
```

J smallest **CORRECTLY** sorted at j first pos after j iteration

n + (n – 1) + (n – 2) + (n – 3) + ... + 1 = (n)(n+1)/2

Insertion-Sort(A, n)

```
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
```

Repeat at most j times.

first j items are sorted(may not be final) after j iterations

MergeSort(A, n)

```
if (n=1) 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);
```

QuickSort(A[1..n], n)

```
if (n == 1) then return;
else Choose pivot index pIndex.
p = partition(A[1..n], n, pIndex)
x = QuickSort(A[1..p–1], p–1)
y = QuickSort(A[p+1..n], n–p)
```

partition(A[1..n], n, pIndex)

```
// Assume no duplicates, n>1 pivot = A[pIndex];
// store pivot in A[1] low = 2; start after pivot in A[1]
// high = n+1; Define: A[n+1] = ∞
while (low < high)
      while (A[low] < pivot)&&(low < high)   low++;
      while (A[high] > pivot)&&(low < high)  high- -;
      if (low < high) then swap(A[low], A[high]);
swap(A[1], A[low–1]); return low–1;
```

Name	Best Case	Average Case	Worst Case	Extra Memory	Stable?
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n log n)	Yes
Quick Sort	O(nlog n)	O(nlogn)	O(n^2)	O(n)	NO
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	NO

QuickSelect(A[1..n], n, k)

```
if (n == 1) then return A[1];
else
Choose random pivot index pindex = random();
pIndex. p = partition(A[1..n], n, pIndex)
if (k == p) then return A[p];
else if (k < p) then return Select(A[1..p–1], k)
else if (k > p) then return Select(A[p+1], k – p)
```

Recurrence Relations

Unrolling the recurrence:

Rule: T(X) = T(X/2) + O(1)

$$T(n) = T(n/2) + \theta(1)$$
$$= T(n/4) + \theta(1) + \theta(1)$$
$$= T(n/8) + \theta(1) + \theta(1) + \theta(1)$$
$$\ldots$$
$$\ldots$$
$$= T(1) + \theta(1) + \ldots + \theta(1) =$$
$$= \theta(1) + \theta(1) + \ldots + \theta(1) =$$

Number of times you can divide n by 2 until you reach 1.

number $= 2^{level}$

$n = 2^h$

$\log n = h$

Trees

```
public int height(){
int leftHeight = -1;
int rightHeight = -1;
if (leftTree != null)
      leftHeight = leftTree.height();
if (rightTree != null)
      rightHeight = rightTree.height();
return max(leftHeight, rightHeight) + 1;}
```

public TreeNode searchMax(){

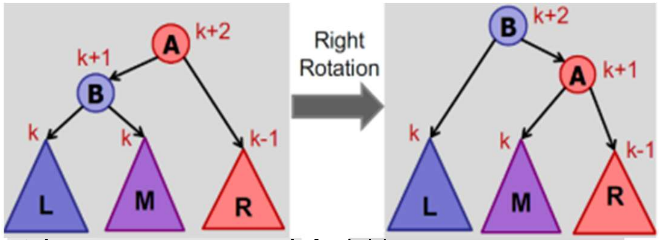
```
if (rightTree != null) { return rightTree.searchMax(); }
else return this; // Key is here!
```

public void in-order-traversal(){

```
if (leftTree != null) leftTree.in-order-traversal();
      visit(this);
if (rightTree != null) rightTree.in-order-traversal();
```

Pre: this, left, right **Post: left, right, this**

AVL TREES



Right rotation requires a **left** child

If v is unbalanced and left heavy	Balancing
1. v.left is balanced(deletion of node)	right-rotate(v)
2. v.left is left-heavy(root.height-1)	right-rotate(v)
3. v.left is right-heavy(root.height-1)	Left-rotate(v.left) right-rotate(v)

1. Insert key
2. Walk up Tree: Check for Balance and Rotate(at most 2)

Only need to fix ***LOWEST*** out-of-balance node

Deletion

1. If v has 2 children, swap with its successor
2. Delete node v from tree and reconnect children
3. For every ancestor to the root, check if height-balanced then perform rotation

Deletion may take up to **O(log(n))** rotations.(O(height))

TRIES

Space O(size of text + overhead) space is a problem

Insert string take **O(L)** time as comapared of **O(Llogn)** for AVL trees **where** n is number of levels

Search take O(L) time;

One Dimensional Range Queries

– v = FindSplit(low, high); **O(logn)**
– LeftTraversal(v, low, high);
– RightTraversal(v, low, high);

```
LeftTraversal(v, low , high) {
  If (low <= key) {
     All-leaf-traversal(v.right); // outputting nodes
      LeftTraversal(v.left, low ,high);
  Else {
      LeftTraversal(v.right, low, high);
```

Basically all leaf traversal as the subtree is between the split node and the range

Runtime O(k + log(n)) k- number of points found

Cost of all leave traversal depends on number of leaves

k number of leaves means **2k** total number of nodes(copies) which becomes **O(k)**

If u just want to know how many nodes?

Simply add weight to nodes similar to order statistic tree then find the weight of the split node.

PRIORITY QUEUE - HEAP

Every level is full, except possibly the last

ALL nodes are as far **LEFT** as possible

Maximum height of n elements – **Floor(log n)**

Heap vs **AVL Tree:**

- Same asymptotic cost for operations
- Faster real cost(no constant factors)
- Simpler: no rotation
- Slightly better concurrency

Heap Operations(listed below)**: O(logn)**

INSERT

1. Add leaf with the priority P
2. Bubble up

```
While (v! null) {
   If (priority(v) > priority(parent(v)) {
       Swap(v, parent(v));
   } else return;
   V = parent(v);
   }
}
```

IncreaseKey --> BubbleUp(node)

DecreaseKey

1. Update priority

2. **Bubble down**

While (!leaf(v)) {

Find max priority between the 2 children nodes AND current node

Swap with the child node with max priority

Else if current node has higher priority, return

DELETE

1. Swap with last key
2. Remove last key
3. Bubble down

ExtractMax – >Delete root node

Stored as Array

Array slot 0 : root node

Left(x) = 2 x + 1   Right(x) = 2 x + 2

Parent(x) = floor((x-1) / 2)

Unable to store **AVL** trees in array – blank elements & rotation

Unsorted list → Heap **v1**

```
for (int i=0; i<n; i++) {
    int value = A[i];
    A[i] = EMPTY:
   heapInsert(value, A, 0, i); // O(logn)
}
```

Unsorted list → Heap **v2 O(n)!!**

```
for (int i=(n-1); i>=0; i--) {
    bubbleDown(i, A); // O(log n) (height)
}
```

HEAP SORT

Heap array → Sorted List

```
for (int I = (n - 1);  I >= 0; i--) {
     int value = extractMax(A); // O(logn)
      A[i] = value;
}
```

Runtime ordering

Function
5
loglog(n)
log(n)
$\log^2(n)$
n
nlog(n)
n^3
$n^3\log(n)$
n^4
2^n
2^{2n}
n!

2D Dimensional range Tree

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

- Static 2d-range trees support efficient operations.
- We do not support insert/delete operations in 2d-range trees because rotations would be too expensive.

Query cost: $O(\log^d n + k)$

buildTree cost: $O(n \log^{d-1} n)$

Space: $O(n \log^{d-1} n)$

Done By: Justin Lim   5/3/2023
Version 001