Searching Algorithms

=====================================================================================

Interface used

======================

**public** **interface** ISearchingAlgo {

/\*\* 1. Linear Search \*\*/

**public** Integer linearSearch(**int** a[], **int** x);

/\*\* 2. Binary Search l=start index, and e end index \*\*/

**public** Integer binarySearch(**int** a[], **int** l, **int** e, **int** x);

/\*\* 3. Jump Search \*\*/

**public** Integer jumpSearch(**int** a[], **int** x);

/\*\* 4. Interpolation Search \*\*/

**public** Integer interpolationSearch(**int** a[], **int** x);

/\*\* 5. Exponential Search \*\*/

**public** Integer exponentialSearch(**int** a[], **int** x);

/\*\* 6. Sublist Search (Search a linked list in another list) \*\*/

/\*\* link list creation \*\*/

**public** SLNode createLinkList(SLNode node, **int** dt);

/\*\* print list \*\*/

**public** **void** printSingleLinkList(SLNode node);

**public** **boolean** subListInListSearch(SLNode list, SLNode subList);

/\*\* 7. Fibonacci Search \*\*/

**public** Integer fibonacciSearch(**int** a[], **int** x);

/\*\*

\* 8. The Ubiquitous Binary Search : Here array is sorted and element may be

\* duplicate

\*\*/

**public** **int** getRightPosition(**int** a[], **int** l, **int** r, **int** x);

**public** **int** getLeftPosition(**int** a[], **int** l, **int** r, **int** x);

**public** Integer ubiquitousBinarySearch(**int** a[], **int** x);

/\*\* 12.binary search for first occuracne if multiple value exists \*\*/

**public** **int** binarySearchFirstOccurance(**int** a[], **int** l, **int** h, **int** x);

/\*\* 13.binary search for last occurance if multiple value exists \*\*/

**public** **int** binarySearchLastOccurance(**int** a[], **int** l, **int** h, **int** x);

}

Implementation of Interface

=====================================================================================

**public** **class** SearchingAlgoImpl **implements** ISearchingAlgo {

/\*\* 1. Linear Search \*\*/

@Override

**public** Integer linearSearch(**int** a[], **int** x) {

**if** (a == **null** || a.length == 0)

**return** **null**;

**int** j = a.length - 1;

**for** (**int** i = 0; i <= j; i++, j--) {

**if** (a[i] == x)

**return** i;

**if** (a[j] == x)

**return** j;

}

**return** **null**;

}

/\*\*

\* 2. Binary Search l=start index, and e end index. Note: Binary search can be

\* apply on sorted array only

\*\*/

**public** Integer binarySearch(**int** a[], **int** l, **int** e, **int** x) {

**if** (a == **null** || a.length == 0 || x < a[l] || a[e] < x)

**return** **null**;

**if** (a[l] == x)

**return** l;

**if** (a[e] == x)

**return** e;

**int** mid = l + (e - l) / 2;

**if** (l < e) {

**if** (a[mid] == x)

**return** mid;

**else** **if** (x < a[mid])

**return** binarySearch(a, l, mid, x);

**else**

**return** binarySearch(a, mid + 1, e, x);

}

**return** **null**;

}

/\*\*

\* 3. Jump Search Note: Jump search can be apply on sorted array only

\*\*/

@Override

**public** Integer jumpSearch(**int** a[], **int** x) {

**if** (a == **null** || a.length == 0 || x < a[0] || a[a.length - 1] < x)

**return** **null**;

**else** {

**int** step = (**int**) Math.*sqrt*(a.length);

**int** i = 0;

**int** prev = 0;

**for** (; i < a.length; i = i + step) {

**if** (a[prev] <= x && x <= a[i])

**break**;

prev = i;

}

i = i - step;

**if** (i < 0)

i = 0;

**for** (**int** j = i; j <= i + step && j < a.length; j++) {

**if** (a[j] == x)

**return** j;

}

}

**return** **null**;

}

/\*\*

\* 4. Interpolation Search : Note this sort can be apply on sorted array only

\*\*/

@Override

**public** Integer interpolationSearch(**int** a[], **int** x) {

**if** (a == **null** || a.length == 0 || x < a[0] || a[a.length - 1] < x)

**return** **null**;

**if** (a[0] == x)

**return** 0;

**if** (a[a.length - 1] == x)

**return** a.length - 1;

**int** lo = 0;

**int** hi = a.length - 1;

**if** (x > a[hi] || x < a[lo])

**return** -1;

**while** (lo < hi) {

**int** pos = lo + (x - a[lo]) \* (hi - lo) / (a[hi] - a[lo]);

**if** (a[pos] == x)

**return** pos;

**else** **if** (a[pos] < x)

lo = pos + 1;

**else**

hi = pos - 1;

}

**return** **null**;

}

/\*\*

\* 5. Exponential Search : Note this sort can be apply on sorted array only

\*\*/

@Override

**public** Integer exponentialSearch(**int** a[], **int** x) {

**if** (a == **null** || a.length == 0 || x < a[0] || a[a.length - 1] < x)

**return** **null**;

**if** (a[0] == x)

**return** 0;

**if** (a[a.length - 1] == x)

**return** a.length - 1;

**int** i = 1;

**while** (i < a.length && a[i] <= x)

i \*= 2;

**return** binarySearch(a, i / 2, Math.*min*(i, a.length - 1), x);

}

/\*\* 6. Sublist Search (Search a linked list in another list) \*\*/

/\*\* link list creation \*\*/

@Override

**public** SLNode createLinkList(SLNode node, **int** dt) {

**if** (node == **null**)

**return** **new** SLNode(dt);

**else** **if** (node.next == **null**) {

node.next = **new** SLNode(dt);

} **else**

createLinkList(node.next, dt);

**return** node;

}

/\*\* print list \*\*/

@Override

**public** **void** printSingleLinkList(SLNode node) {

**if** (node != **null**) {

System.***out***.print(node.dt + "->");

printSingleLinkList(node.next);

}

}

@Override

**public** **boolean** subListInListSearch(SLNode list, SLNode subList) {

**if** (list == **null** || subList == **null**)

**return** **false**;

SLNode sbHead = subList;

SLNode lHead = list;

**for** (; lHead != **null**; lHead = lHead.next) {

**if** (sbHead.dt == lHead.dt) {

**for** (; sbHead != **null** && lHead != **null**; sbHead = sbHead.next, lHead = lHead.next) {

**if** (lHead.dt != sbHead.dt)

**break**;

}

**if** (sbHead == **null**)

**return** **true**;

}

}

**return** **false**;

}

/\*\*

\* 7. Fibonacci Search Note: This sort is applicable for sorted array only

\*\*/

@Override

**public** Integer fibonacciSearch(**int** a[], **int** x) {

**if** (a == **null** || a.length == 0 || x < a[0] || a[a.length - 1] < x)

**return** **null**;

**if** (x == a[0])

**return** 0;

**if** (a[a.length - 1] == x)

**return** a.length - 1;

**int** n = a.length;

**int** n1 = 0;

**int** n2 = 1;

**int** n3 = n1 + n2;

**while** (n3 < n) {

n1 = n2;

n2 = n3;

n3 = n1 + n2;

}

**if** (a[n2] < x && x <= a[n - 1])

**return** binarySearch(a, n2, n - 1, x);

**else** {

**while** (!(a[n1] < x && x <= a[n2])) {

n1 = n2 - n1;

n2 = n3 - n2;

n3 = n1 + n2;

}

**return** binarySearch(a, n1, n2, x);

}

}

/\*\*

\* 8. The Ubiquitous Binary Search : Here array is sorted and element may be

\* duplicate

\*\*/

**public** **int** getRightPosition(**int** a[], **int** l, **int** r, **int** x) {

**int** m;

**while** (r - l > 1) {

m = l + (r - l) / 2;

**if** (a[m] <= x)

l = m;

**else**

r = m;

}

**return** l;

}

**public** **int** getLeftPosition(**int** a[], **int** l, **int** r, **int** x) {

**int** m;

**while** (r - l > 1) {

m = l + (r - l) / 2;

**if** (a[m] >= x)

r = m;

**else**

l = m;

}

**return** r;

}

**public** Integer ubiquitousBinarySearch(**int** a[], **int** x) {

**int** left = getLeftPosition(a, 0, a.length - 1, x);

**int** right = getRightPosition(a, 0, a.length - 1, x);

**return** (a[left] == x && x == a[right]) ? (right - left + 1) : **null**;

}

/\*\* 12.binary search for first occuracne if multiple value exists \*\*/

@Override

**public** **int** binarySearchFirstOccurance(**int** a[], **int** l, **int** h, **int** x) {

**if** (l <= h) {

**int** mid = l + (h - l) / 2;

**if** ((mid == 0 || x > a[mid - 1]) && (x == a[mid]))

**return** mid;

**else** **if** (x > a[mid])

**return** binarySearchFirstOccurance(a, mid + 1, h, x);

**else**

**return** binarySearchFirstOccurance(a, l, mid - 1, x);

}

**return** -1;

}

/\*\* 13.binary search for last occurance if multiple value exists \*\*/

@Override

**public** **int** binarySearchLastOccurance(**int** a[], **int** l, **int** h, **int** x) {

**if** (l <= h) {

**int** mid = l + (h - l) / 2;

**if** ((mid == a.length - 1 || x < a[mid + 1]) && (x == a[mid])) {

**return** mid;

} **else** **if** (x < a[mid])

**return** binarySearchLastOccurance(a, l, mid - 1, x);

**else**

**return** binarySearchLastOccurance(a, mid + 1, h, x);

}

**return** -1;

}

}

Test cases

====================================================================

**public** **class** ISearchingAlgoTest {

**public** ISearchingAlgo isa = **null**;

@Before

**public** **void** init() {

isa = **new** SearchingAlgoImpl();

}

// \*\* 1. Linear Search \*\*/

@Test

**public** **void** linearSearchTest() {

**int** a[] = **null**;

**int** b[] = {};

**int** c[] = { 1 };

**int** d[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Assert.*assertTrue*(isa.linearSearch(a, 1) == **null**);

Assert.*assertTrue*(isa.linearSearch(b, 1) == **null**);

Assert.*assertTrue*(isa.linearSearch(c, 1) == 0);

Assert.*assertTrue*(isa.linearSearch(d, 7) == 6);

Assert.*assertTrue*(isa.linearSearch(d, 10) == **null**);

}

/\*\* 2. Binary Search l=start index, and e end index \*\*/

@Test

**public** **void** binarySearchTest() {

**int** a[] = **null**;

**int** b[] = {};

**int** c[] = { 1 };

**int** d[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Assert.*assertTrue*(isa.binarySearch(a, 0, 0, 1) == **null**);

Assert.*assertTrue*(isa.binarySearch(b, 0, b.length - 1, 1) == **null**);

Assert.*assertTrue*(isa.binarySearch(c, 0, c.length - 1, 1) == 0);

Assert.*assertTrue*(isa.binarySearch(d, 0, d.length - 1, 7) == 6);

Assert.*assertTrue*(isa.binarySearch(d, 0, d.length - 1, 10) == **null**);

Assert.*assertTrue*(isa.binarySearch(d, 0, d.length - 1, 9) == 8);

**int** a1[] = { 10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100 };

Assert.*assertTrue*(isa.binarySearch(a1, 5, 8, 85) == 8);

}

/\*\* 3. Jump Search \*\*/

@Test

**public** **void** jumpSearchTest() {

**int** a[] = **null**;

**int** b[] = {};

**int** c[] = { 1 };

**int** d[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Assert.*assertTrue*(isa.jumpSearch(a, 1) == **null**);

Assert.*assertTrue*(isa.jumpSearch(b, 1) == **null**);

Assert.*assertTrue*(isa.jumpSearch(c, 1) == 0);

Assert.*assertTrue*(isa.jumpSearch(d, 7) == 6);

Assert.*assertTrue*(isa.jumpSearch(d, 10) == **null**);

}

/\*\* 4. Interpolation Search \*\*/

@Test

**public** **void** interpolationSearchTest() {

**int** a[] = **null**;

**int** b[] = {};

**int** c[] = { 1 };

**int** d[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Assert.*assertTrue*(isa.interpolationSearch(a, 1) == **null**);

Assert.*assertTrue*(isa.interpolationSearch(b, 1) == **null**);

Assert.*assertTrue*(isa.interpolationSearch(c, 1) == 0);

Assert.*assertTrue*(isa.interpolationSearch(d, 7) == 6);

Assert.*assertTrue*(isa.interpolationSearch(d, 10) == **null**);

}

/\*\* 5. Exponential Search \*\*/

@Test

**public** **void** exponentialSearchTest() {

**int** a[] = **null**;

**int** b[] = {};

**int** c[] = { 1 };

**int** d[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Assert.*assertTrue*(isa.exponentialSearch(a, 1) == **null**);

Assert.*assertTrue*(isa.exponentialSearch(b, 1) == **null**);

Assert.*assertTrue*(isa.exponentialSearch(c, 1) == 0);

Assert.*assertTrue*(isa.exponentialSearch(d, 7) == 6);

Assert.*assertTrue*(isa.exponentialSearch(d, 10) == **null**);

}

/\*\* link list creation \*\*/

@Test

**public** **void** createLinkListTest() {

**int** a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode head = **null**;

**for** (**int** i = 0; i < a.length; i++)

head = isa.createLinkList(head, a[i]);

isa.printSingleLinkList(head);

}

@Test

**public** **void** subListInListSearchTest() {

**int** a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode head = **null**;

**for** (**int** i = 0; i < a.length; i++)

head = isa.createLinkList(head, a[i]);

**int** b[] = { 4, 5, 6 };

SLNode sHead1 = **null**;

**for** (**int** i = 0; i < b.length; i++)

sHead1 = isa.createLinkList(sHead1, b[i]);

Assert.*assertTrue*(isa.subListInListSearch(head, sHead1));

**int** c[] = { 4, 1, 6 };

SLNode sHead2 = **null**;

**for** (**int** i = 0; i < c.length; i++)

sHead2 = isa.createLinkList(sHead2, c[i]);

Assert.*assertTrue*(isa.subListInListSearch(head, sHead2) == **false**);

}

/\*\* 7. Fibonacci Search \*\*/

@Test

**public** **void** fibonacciSearchTest() {

**int** a[] = { 10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100 };

Assert.*assertTrue*(isa.fibonacciSearch(a, 85) == 8);

Assert.*assertTrue*(isa.fibonacciSearch(a, 85) == 8);

Assert.*assertTrue*(isa.fibonacciSearch(a, 100) == 10);

Assert.*assertTrue*(isa.fibonacciSearch(a, 10) == 0);

Assert.*assertTrue*(isa.fibonacciSearch(a, 108) == **null**);

Assert.*assertTrue*(isa.fibonacciSearch(a, 35) == 2);

Assert.*assertTrue*(isa.fibonacciSearch(a, 22) == 1);

Assert.*assertTrue*(isa.fibonacciSearch(a, 40) == 3);

Assert.*assertTrue*(isa.fibonacciSearch(a, 80) == 6);

Assert.*assertTrue*(isa.fibonacciSearch(a, 90) == 9);

}

@Test

**public** **void** getRightPositionTest() {

**int** a[] = { 10, 22, 35, 40, 45, 80, 80, 80, 85, 90, 100 };

Assert.*assertTrue*(isa.getRightPosition(a, 0, a.length - 1, 80) == 7);

}

@Test

**public** **void** getLeftPositionTest() {

**int** a[] = { 10, 22, 35, 40, 45, 80, 80, 80, 85, 90, 100 };

Assert.*assertTrue*(isa.getLeftPosition(a, 0, a.length - 1, 80) == 5);

}

@Test

**public** **void** ubiquitousBinarySearchTest() {

**int** a[] = { 10, 22, 35, 40, 45, 80, 80, 80, 85, 90, 100 };

Assert.*assertTrue*(isa.ubiquitousBinarySearch(a, 80) == 3);

}

@Test

**public** **void** binarySearchFirstOccuranceTest() {

**int** a[] = { 1, 2, 2, 2, 4, 5, 5, 5, 6, 7, 7, 7, 8, 8 };

Assert.*assertTrue*(isa.binarySearchFirstOccurance(a, 0, a.length - 1, 2) == 1);

Assert.*assertTrue*(isa.binarySearchFirstOccurance(a, 0, a.length - 1, 5) == 5);

Assert.*assertTrue*(isa.binarySearchFirstOccurance(a, 0, a.length - 1, 8) == 12);

}

@Test

**public** **void** binarySearchLastOccuranceTest() {

**int** a[] = { 1, 2, 2, 2, 4, 5, 5, 5, 6, 7, 7, 7, 8, 8 };

Assert.*assertTrue*(isa.binarySearchLastOccurance(a, 0, a.length - 1, 2) == 3);

Assert.*assertTrue*(isa.binarySearchLastOccurance(a, 0, a.length - 1, 5) == 7);

Assert.*assertTrue*(isa.binarySearchLastOccurance(a, 0, a.length - 1, 8) == 13);

}

}