|  |  |  |
| --- | --- | --- |
| Searching Report | March 12  2009 | |
| Sequential, Binary, Interpolation, BST-trees, AVL-trees | |  |

###### 

Alexandria University

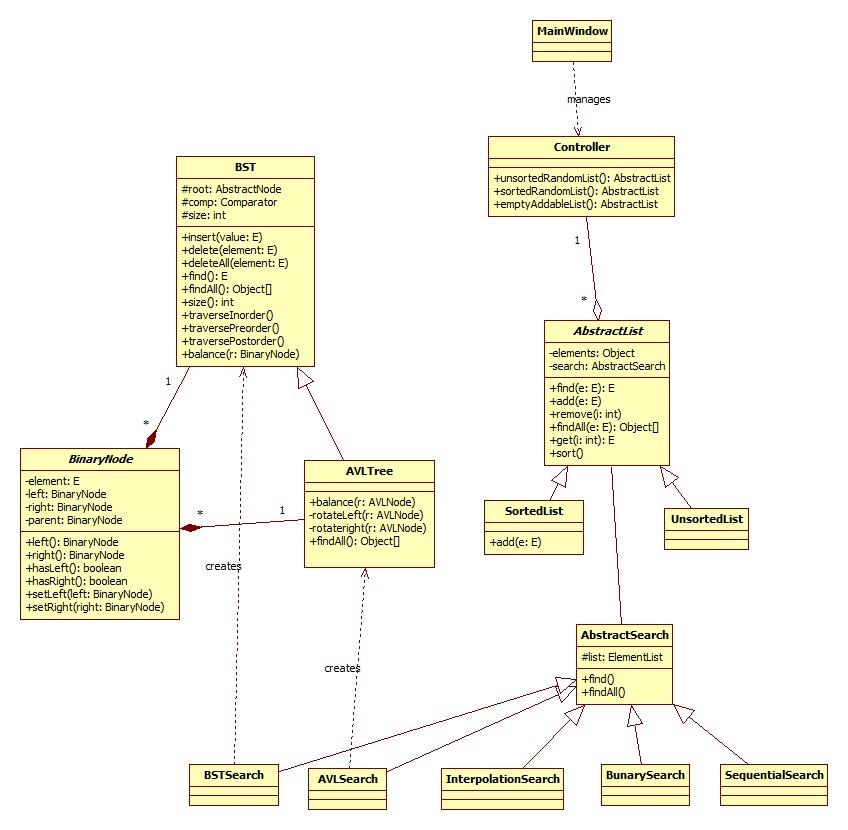
Faculty of Engineering

Computer and Systems Engineering Department

Automatic Control

Name: Mostafa Mahmod Mahmod Eweda

Seat No.: 65

**Class Diagram**

**Sequential Search**

**Advantages**

* Simple to implement.
* Maintainable and easy to change and use its internal data.
* Used in ever day’s program for simple non-requiring operations.
* Consumes no external or internal space for recursive operations as it can be easily implemented iteratively.
* Goes the same on sorted and unsorted lists.

**Disadvantages**

* Slow.
* Always eliminating one item per iteration even if it is much far from the desired destination. (Normal step <-- 1).
* O (n).

**Binary Search**

**Advantages**

* Fast as it eliminates half of the elements in the list by one iteration.
* Can be implemented both iteratively and recursively.
* O (log2(n))
* Runs the same for any distribution of the list.

**Disadvantages**

* Requiring a previously sorted list
* Always eliminating half of the list even if the desired element is in one of the edges if the list.

**Interpolation Search**

**Advantages**

* Fast as it eliminates most of the elements in the list by one iteration.
* Can be implemented both iteratively and recursively.
* O (log2log2(n)) if the list is normally distributed.

**Disadvantages**

* Requires previous knowledge that the data is normally distributed to achieve the best performance.

**BST (Binary Search Trees)**

**Advantages**

* Structured search.
* Can be used in implementing decision trees as in games or artificial intelligence.
* Having the data ready for searching instead of having the method ready for the list.
* O(log2n) for randomly inserted lists.

**Disadvantages**

* Can reach to a linked list behavior when the data sequence is already sorted O (n).
* Consumes additional memory to save the tree nodes.
* Requires additional effort for implementation.

**AVL Trees**

**Advantages**

* Structured search.
* Can be used in implementing decision trees as in games or artificial intelligence.
* Having the data ready for searching instead of having the method ready for the list.
* O(log2n) for the worst case.
* Consumes less memory than BST for dereferenced nodes (not noticeable).

**Disadvantages**

* Consumes additional memory to save the tree nodes. O(n).
* Requires much more additional effort for implementation.

**AVL Tree Algorithms (Specialized)**

**Algorithm** findAll(key e) {

**if** (size equals 0)

**return** **NOT\_FOUND**;

temp 🡨 find(root, element);

found 🡨 findAll(list, temp, element);

**return** found;

}

**Algorithm** findAll(list, Node temp, Element e) {

**if** (temp equals **NULL**)

**return**;

**if** (compare(temp.element, e) equals 0) then

list.add(temp);

findAll(list, temp.right, e);

findAll(list, temp.left, e);

}

**Algorithm** Node balance(Node r) {

**if** (r.hl > r.hr) then

**if** (r.left.hr > r.left.hl) then

Node node 🡨 rotateLeft(r.left);

node.parent 🡨 r;

r.left 🡨 node;

Node temp 🡨 r.parent;

r 🡨 rotateRight(r);

r.parent 🡨 temp;

**else**

**if** (r.right.hl > r.right.hr) then

Node node 🡨 rotateRight(r.right);

node.parent 🡨 r;

r.right 🡨 node;

Node temp 🡨 r.parent;

r 🡨 rotateLeft(r);

r.parent 🡨 temp;

}

**return** r;

}

**Algorithm** Node rotateLeft(Node A) {

Node B 🡨 A.right;

A.right 🡨 B.left;

A.hr 🡨 B.hl;

B.parent 🡨 A.parent;

A.parent 🡨 B;

B.left 🡨 A;

If (B.hasRight) then

B.hr 🡨 Max(B.right.hl, B.right.hr) + 1

else

B.hr 🡨 0;

If (B.hasLeft) then

B.hl 🡨 Max(B.left.hl, B.left.hr) + 1 : 0;

**return** B;

}

**Algorithm** Node rotateRight(Node A) {

Node B 🡨 A.left;

A.left 🡨 B.right;

A.hl 🡨 B.hr;

B.parent 🡨 A.parent;

A.parent 🡨 B;

B.right 🡨 A;

If (B.hasRight) then

B.hr 🡨 Max(B.right.hl, B.right.hr) + 1

Else

B.hr 🡨 0;

**if** (B.hasLeft)

B.hl 🡨 Max(B.left.hl, B.left.hr) + 1;

**else**

B.hl 🡨 0;

**return** B;

}

**BST Tree Algorithms (generalized)**

**Algorithm** insert(E value) {

**if** (size equals 0) then

addRoot(**new** Node(value));

**else**

Node node 🡨 **new** Node(value);

root 🡨 insertRec(root, node);

size = size + 1;

}

**Algorithm** Node insertRec(Node r, Node n) {

**if** (r equals **NULL**) then

**return** n;

**int** comparison 🡨 compare(n.element, r.element);

**if** (comparison < 0) then

Node temp 🡨 insertRec(r.left, n);

r.left 🡨 temp;

temp.parent 🡨 r);

r.hl 🡨 Max(r.left.hl, r.left.hr) + 1;

} **else** {

Node temp 🡨 insertRec(r.right, n);

r.right 🡨 temp;

temp.parent 🡨 r;

r.hr 🡨 Max((r.right.hl, (r.right.hr) + 1;

}

**if** (Math.*abs*(r.hl - r.hr) > 1)

r 🡨 balance(r);

**return** r;

}

**Algorithm** find(e) {

**if** (size equals 0)

**return** **NULL**;

Node temp 🡨 find(root, element);

**if** (temp equals **NULL**)

**return** **NULL**;

**else**

**return** temp.element;

}

**Algorithm** Node find(Node root, e) {

Node next 🡨 root;

comparison 🡨 0;

comparisons 🡨 0;

**while** (next not equals **NULL**) {

comparisons = comparisons + 1;

comparison 🡨 compare(element, next.element);

**if** (comparison < 0) then

next 🡨 next.left;

**else** **if** (comparison > 0) then

next 🡨 next.right;

**else**

**return** next;

**return** **NULL**;

}

**Algorithm** findAll(E element) {

**if** (size equals 0)

**return** **NULL**;

Node temp 🡨 find(root, element);

**int** counter 🡨 0;

**while** (temp not equals **NULL** AND compare(element, temp.element)

equals 0)

found[counter++] 🡨 temp.element;

temp 🡨 temp.right;

}

**return** found;

}

**Algorithm** delete(element) {

root 🡨 deleteRec(root, element);

size = size - 1;

}

**Algorithm** deleteAll(element) {

**int** n 🡨 findAll(element).length;

size 🡨 n;

**for** (**int** i 🡨 0; i < n; i = i + 1)

root 🡨 deleteRec(root, element);

}

**Algorithm** Node deleteRec(Node node, element) {

**if** (node equals **NULL**)

**return** **NULL**;

**int** comparsion 🡨 compare(element, node.element);

**if** (comparsion < 0) then

node.left 🡨 deleteRec(node.left, element));

node.hl 🡨 Max(node.hasLeft() ? node.left.hl : 0, node

.hasLeft() ? node.left.hr : 0) + 1;

**if** (Math.*abs*(node.hl - node.hr) > 1)

node 🡨 balance(node);

**return** node;

} **else** **if** (comparsion > 0) {

node.right 🡨 deleteRec(node.right, element));

node.hr 🡨 Max(node.hasRight() ? node.right.hl : 0, node

.hasRight() ? node.right.hr : 0) + 1;

**if** (Math.*abs*(node.hl - node.hr) > 1)

node 🡨 balance(node);

**return** node;

} **else** {

**if** (!(node.hasLeft() OR node.hasLeft())) then

**return** **NULL**;

**else** **if** (NOT node.hasRight()) then

**return** node.left;

**else** **if** (NOT node.hasLeft() then

**return** node.right;

**else**

Node temp 🡨 successor(node);

node.setElement(temp.element);

node.right 🡨 deleteRec(node.right, temp.element));

**return** node;

}

}

**Algorithm** Node successor(Node node) {

Node temp 🡨 node.right;

**while** (temp.hasLeft)

temp 🡨 temp.left;

**return** temp;

}

**Algorithm** postOrder(Node node, StringBuffer buffer) {

**if** (node equals **NULL**)

**return**;

postOrderHelper(node.left, buffer);

postOrderHelper(node.right, buffer);

visit(node.element);

}

Maximum Comparisons

Sorted Lists

Unsorted Lists

Maximum Running Time

Sorted Lists

Unsorted Lists

Average Comparisons

Sorted Lists

Unsorted Lists

Average Time

Sorted Lists

Unsorted Lists

**Comparison between BST and AVL trees**

|  |  |  |
| --- | --- | --- |
| Construction time | More time is needed for balancing the tree but when the tree gets excessively big, the steps done for balancing is negligible. | Good in time management but when the tree gets too big, the insertion becomes costly either. |
| so they are two much equal. |
| Search time | Better | Good |
| Sorted list (worst) | Doesn’t matter. O(n log2n) | Changes to a linked list. O(n) |
| Unordered | Doesn’t matter. O(n log2n) | O(log2n) for average. |

Statistical Charts

Average Comparisons **Sorted Lists**

Average Running Time **Sorted Lists**

Average Comparisons **Unsorted Lists**

Average Time **Unsorted Lists**