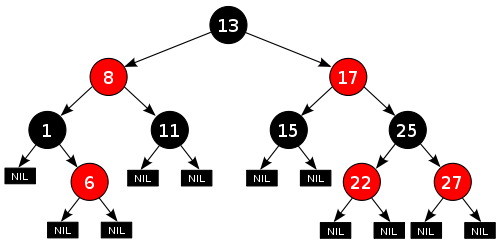
# Structures:

TREES:

|  |  |  |  |
| --- | --- | --- | --- |
| tree | BST | AVL | RBT |
| property |  | First dynamically balanced trees   1. The sub-trees of every node differ in height by at most one.   Root H:0  Left H:1/2/1  Right H:2 /1/1   1. Every sub-tree is an AVL tree. |  |
| search time |  | ***O(*log*n)*** |  |
| Insertion |  | ***ALL the basic logic is in the Insertion and the point of insertion determined like BST .***  ***2. Check the Grand parent(as +1 is ht diff is allowed) Balance factor***  ***i) LSH / RSH***  ***and inserting on the heavy Node then needs rotation***  ***ii) If Equal or inserting on the lite side then no need of Rotation.***  ***Cases :***   1. ***A->rB->rC***   ***Insertion at C***  ***LL rotation***   1. ***A->lB->lC***   ***Insertion at C***  ***RR rotation***   1. ***A->lB->rC***   ***Insertion at C***  ***LR Double rotation***   1. ***A->rB->lC***   ***Insertion at C***  ***RL Double rotation***  ***Balance factor :***  ***Left side heavy***  ***Right side heavy***  ***Equal*** |  |
| addition time |  | ***O(*log*n)*** |  |
|  |  |  |  |
|  |  |  |  |
| Traversing Time |  |  |  |
| Uses |  |  |  |
|  |  |  |  |
|  |  |  |  |

***RED BLACK TREE***

## Properties

[](http://en.wikipedia.org/wiki/File:Red-black_tree_example.svg)

[http://bits.wikimedia.org/static-1.21wmf9/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Red-black_tree_example.svg)

An example of a red–black tree

In addition to the requirements imposed on a [binary search trees](http://en.wikipedia.org/wiki/Binary_search_trees), with red–black trees:[[4]](http://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-4)

1. A node is either red or black.
2. The root is black. (This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice-versa, this rule has little effect on analysis.)
3. All leaves (NIL) are black. (All leaves are same color as the root.)
4. Both children of every red node are black.
5. Every [simple path](http://en.wikipedia.org/wiki/Path_(graph_theory)) from a given node to any of its descendant leaves contains the same number of black nodes.

***BINARY SEARCH TREES:***

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a **binary search tree** (**BST**), sometimes also called an **ordered** or **sorted binary tree**, is a [node-based](http://en.wikipedia.org/wiki/Node_(computer_science)) [binary tree](http://en.wikipedia.org/wiki/Binary_tree) [data structure](http://en.wikipedia.org/wiki/Data_structure) which has the following properties:[[1]](http://en.wikipedia.org/wiki/Binary_search_tree#cite_note-1)

* The left [subtree](http://en.wikipedia.org/wiki/Tree_(data_structure)#Subtree) of a node contains only nodes with keys less than the node's key.
* The right subtree of a node contains only nodes with keys greater than the node's key.
* Both the left and right subtrees must also be binary search trees.
* There must be no duplicate nodes.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the r

|  |  |  |
| --- | --- | --- |
| **Binary search tree** | | |
| [**Type**](http://en.wikipedia.org/wiki/List_of_data_structures) | [Tree](http://en.wikipedia.org/wiki/Tree_(data_structure)) | |
| [**Time complexity**](http://en.wikipedia.org/wiki/Time_complexity) **in**[**big O notation**](http://en.wikipedia.org/wiki/Big_O_notation) | | |
|  | Average | Worst case |
| **Space** | O(n) | O(n) |
| **Search** | O(log n) | O(n) |
| **Insert** | O(log n) | O(n) |
| **Delete** | O(log n) | O(n) |

To insert a new node into a tree, the following method can be used. We first start at the root of the tree, and compare the ordinal value of the root to the ordinal value of the node to be inserted. If the ordinal values are identical, then we have a duplicate and we return to the caller indicating so. If the ordinal value is less than the root, then we follow the left branch of the root, else we follow the right branch. We now start the comparison again but at the branch we took, comparing the ordinal value of the child with the node to be inserted. Traversal of the tree continues in this manner until we reach a left or right node which is empty and we can go no further. At this point, we insert the new node into this empty location. Note that new nodes are always inserted as leaves into the tree, and strictly speaking, nodes are thus appended rather than inserted.

Searching a binary search tree is almost identical to inserting a new node except that we stop the traversal when we find the node we're looking for (during an insertion, this would indicate a duplicate node in the tree). If the node is not located, then we report this to the caller.

Both insertion and searching are naturally recursive and are, arguably, easier to understand when considered in terms of their unit operation. A basic recursive search algorithm will look like:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/18976/A-simple-Binary-Search-Tree-written-in-C)

node search (node, key) {

if node is null then return null;

ifnode.key = key then

return node

if key < node then

return search (node.left, key);

else

return search (node.right, key);

In the source code provided with this article, insertion is implemented recursively, while searching uses an iterative approach.

Deletion is a little bit more complicated but boils down to three rules. The three rules refer to deleting nodes without any children, nodes with one child, and nodes with two children. If a node has no children, then the node is simply deleted. If the node has one child, then the node is deleted and the child node is brought forward to link to the parent. The complication occurs when a node has two children. However, even here, the rules are straightforward when stated. To delete a node with two children, the next ordinal node (called the successive node) on the right branch is used to replaced the deleted node. The successive node is then deleted. The successive node will always be the left most node on the right branch (likewise, the predecessor node will be the right most node on the left branch). The figure below illustrates the deletion rules.

SEARCHING IN BST

algorithm Find(key, root):

current-node := root

**while** current-node **isnot** Nil do

**if** current-node.key= key then

**return** current-node

**elseif** key < current-node.key then

current-node := current-node.left

**else**

current-node := current-node.right

recursive version is equivalent:

algorithm Find**-**recursive(key**,** node): **//** call initially with node **=** root

**if** node **=** Nil **or**node**.**key**=** key **then**

node

**elseif** key **<**node**.**key**then**

Find**-**recursive(key**,**node**.**left)

**else**

Find**-**recursive(key**,**node**.**right)

**Insertion :**

void insert(int value)

{

if(root ==NULL)

root=new Node(value);

else

insertHelper(root, value);

}

voidinsertHelper(Node\* node, int value)

{

if(value < node->key)

{

if(node->leftChild==NULL)

node->leftChild=new Node(value);

else

insertHelper(node->leftChild, value);

}

else

{

if(node->rightChild==NULL)

node->rightChild=new Node(value);

else

insertHelper(node->rightChild, value);

}

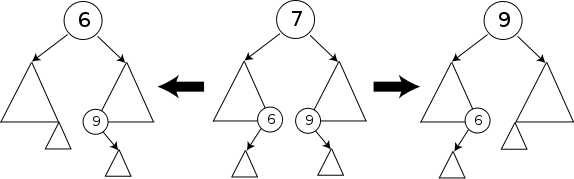
}

### Deletion

There are three possible cases to consider:

* **Deleting a leaf (node with no children):** Deleting a leaf is easy, as we can simply remove it from the tree.
* **Deleting a node with one child:** Remove the node and replace it with its child.
* **Deleting a node with two children:** Call the node to be deleted *N*. Do not delete *N*. Instead, choose either its [in-order](http://en.wikipedia.org/wiki/Tree_traversal) successor node or its in-order predecessor node, *R*. Replace the value of *N* with the value of *R*, then delete *R*.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

[](http://en.wikipedia.org/wiki/File:Binary_search_tree_delete.svg)

Deleting a node with two children from a binary search tree. The triangles represent subtrees of arbitrary size, each with its leftmost and rightmost child nodes at the bottom two vertices.

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an [unbalanced](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) tree, so good implementations add inconsistency to this selection.

Running time analysis: Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

template<typename T>

bool BST<T>::Delete(Node<T>\*&ptr, const T& key)//helper delete function

{

if(ptr==nullptr)

{

returnfalse;// item not in BST

}

if(key <ptr->data)

{

Delete(ptr->LeftChild, key);

}

elseif(key >ptr->data)

{

Delete(ptr->RightChild, key);

}

else

{

//At this point we found the Node to be deleted and delete the node out

Node<T>\*temp;

// 1 right children bring the child to the deleted node

if(ptr->LeftChild==nullptr)

{

temp=ptr->RightChild;

deleteptr;

ptr= temp;

}

elseif(ptr->RightChild==nullptr)

{ // 1 Left children bring the child to the deleted node

temp=ptr->LeftChild;

deleteptr;

ptr= temp;

}

else//2 children

{

temp=ptr->RightChild;

Node<T>\*parent =nullptr;

//From the right child Parse to left reach the end of all left childs

while(temp->LeftChild!=nullptr)

{

parent= temp;

temp= temp->LeftChild;

}

//replace the right child data with the leftmost data of Grandchilds

(basically the idea is to save the lowest value in thye place of the deleted Node . AS the right Node is greater we go for left of right so that we get the node smaller then right node replace that node with deleted node and we r done.

)

ptr->data = temp->data;

// delete the parent left child or elsee the right child

if(parent!=nullptr)

Delete(parent->leftChild,parent->leftChild->data);

else

Delete(ptr->rightChild,ptr->RightChild->data);

}

}

}

***Delete All:***

voidfreetree( structnode \*tree )

{

    if( tree )

    {

        if( tree->left )

        {

            freetree( tree->left )

        }

        if( tree->right )

        {

            freetree( tree->right )

        }

        if(tree->left == NULL) && (tree->right == NULL)

{

delete tree

}

    }

}

***-----------------------------------------------------------------------------***

void tree::delete\_tree(node \*root){

if(root!=NULL)

{

delete\_tree(root->left);

delete\_tree(root->right);

delete(root);

}

}

***Deleting a Subtree:***

***AVL TREES :***

#### Insertion

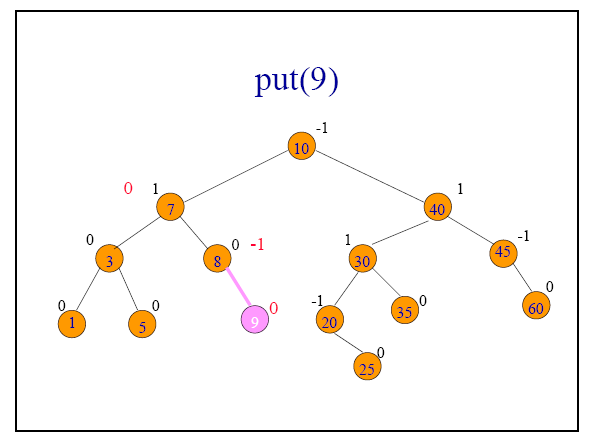
As with the red-black tree, insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the **balance factor** to each node. This factor indicates whether the tree is *left-heavy* (the height of the left sub-tree is 1 greater than the right sub-tree), *balanced* (both sub-trees are the same height) or *right-heavy* (the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.

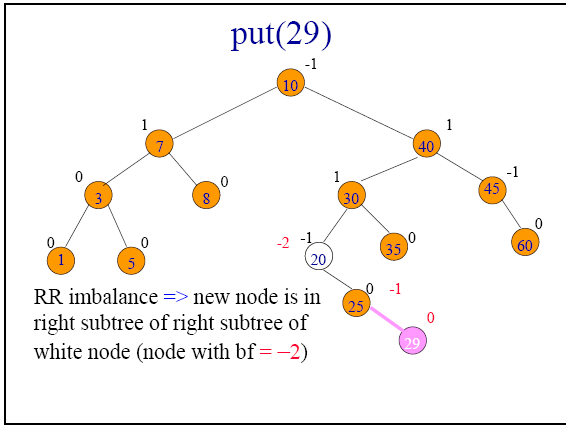
|  |  |
| --- | --- |
| http://www.cs.auckland.ac.nz/~jmor159/PLDS210/fig/AVL_case1.gif | A new item has been added to the left subtree of node 1, causing its height to become 2 greater than 2's right sub-tree (shown in green). A right-rotation is performed to correct the imbalance. |

**Criteria to for Unbalanced Node:**

The Balance criteria(BC) = *Height of the Left Subtree – Height of the right Subtree*

For an AVL tree the Height criteria (-1,0,1)

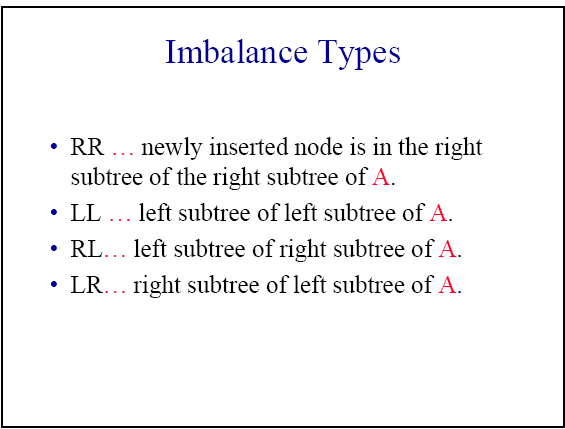


****

The AVL tree balance is broken

When the

**BC =**-2/+2

****

**Set up the pointers:**

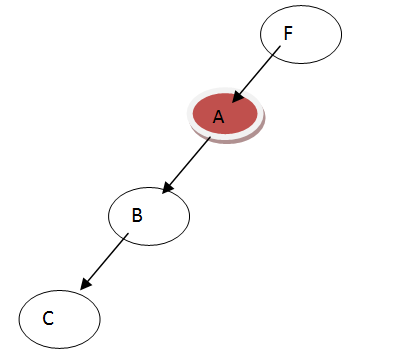
A - points to the node that is out of balance.

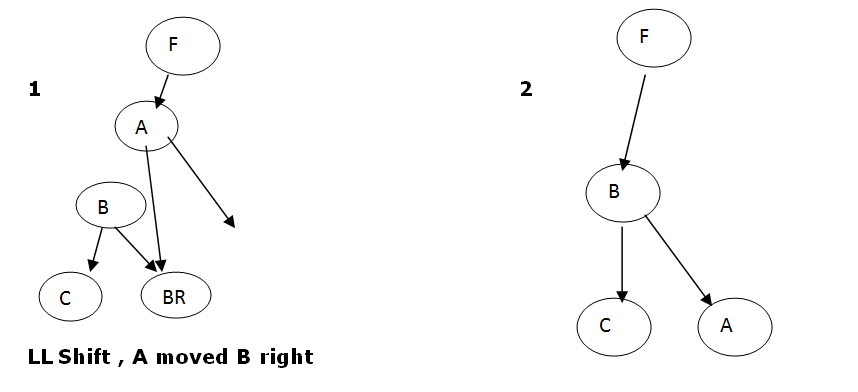
If more than one node is out of balance then select the one that is furthest from the root.

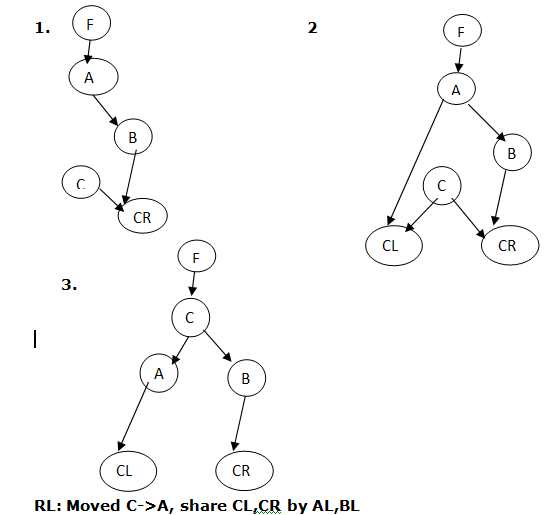
B – points to the child of A in the direction of the out-of-balance

C – points to the child of B in the direction of the out-=of-balance

F – points to the parent of A. This is the only pointer of these 4 that is allowed to be NULL.

****

****

****

**Follow the algorithm:**

RR:                                                                        LL:

          A.Right = B.Left                                                        A.Left = B.Right

          B.Left = A                                                               B.Right = A

          If F = NULL                                                              If F = NULL

                   B is new Root of tree                                                    B is new Root of tree

          Else                                                                        Else

                   If F.Right = A                                                          If F.Right = A

                             F.Right = B                                                                  F.Right = B

                   Else                                                                       Else

                             F.Left = B                                                                   F.Left = B

                   End If                                                                    End If

          End If                                                                     End If

RL:                                                                        LR:

          B.Left = C.Right                                                        A.Left = C.Right

          A.Right = C.Left                                                        B.Right = C.Left

          C.Right = B                                                              C.Left = B

          C.Left = A                                                               C.Right = A

          If F = NULL                                                              If F = NULL

                   C is new Root of tree                                                    C is new Root of tree

          Else                                                                        Else

                   If F.Right = A                                                          If F.Right = A

                             F.Right = C                                                                  F.Right = C

                   Else                                                                       Else

                             F.Left = C                                                                   F.Left = C

                   End If                                                                    End If

          End If                                                                     End If

Single & Double Rotations

• Single

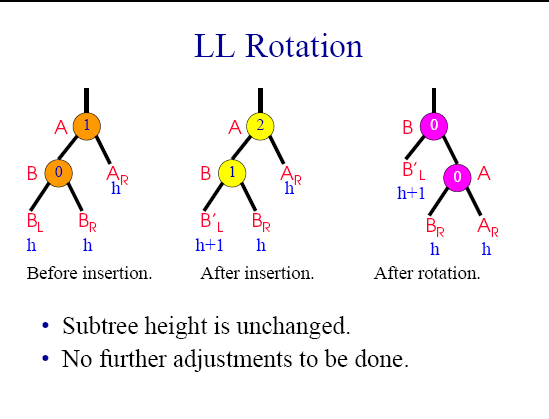
. LL and RR

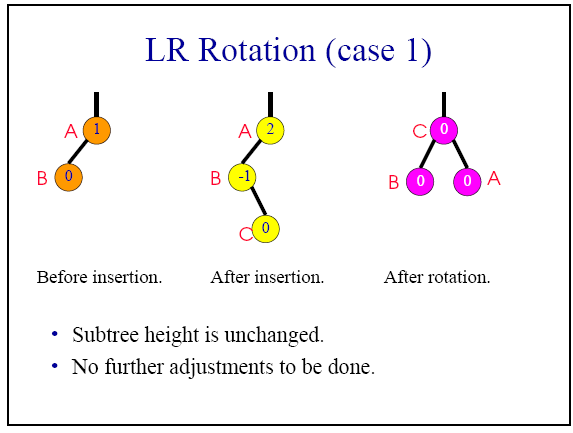
• Double

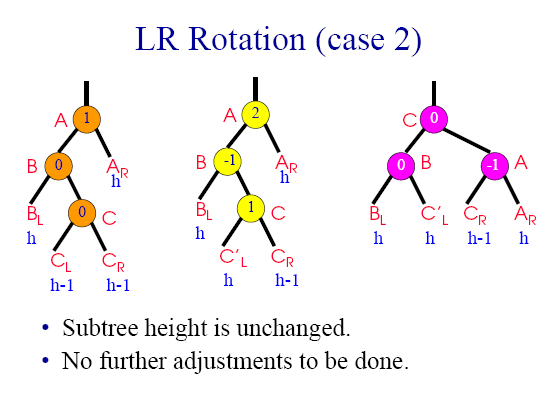
. LR and RL

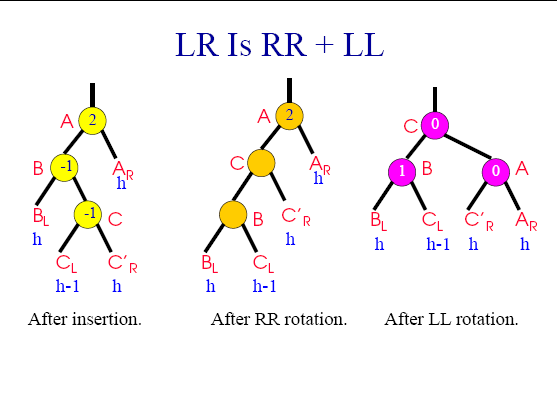
. LR is RR followed by LL

. RL is LL followed by RR

******

******

******

******

***TRIE***

GRAPH

HASH

The main use of hash is for searching as hash table an d it does it in O(1) time so that it can be possible to search it.

If a the number of elemnts are small then the array can be used as hash table , with Array indeces as Hash Keys.

A[i]=[yuy,oowo,eiwi,ewew]

So when we generate a key for yuy it should return us 1 which is the index of that element , this indexes a reuniqly mapped to the elements in the arrays , Search algorithms that use hashing consist of two separate parts. The first step is to compute a *hash function* that transforms the search key into an array index. Ideally, different keys would map to different indices. This ideal is generally beyond our reach, so we have to face the possibility that two or more different keys may hash to the same array index. Thus, the second part of a hashing search is a *collision-resolution* process that deals with this situation.

*load factor =* number of entries /number of buckets

load factor is kept reasonable, the hash table should perform well

A low load factor is not especially beneficial. As load factor approaches 0, the proportion of unused areas in the hash table increases, but there is not necessarily any reduction in search cost. This results in wasted memory.

Hashing Functions:(HASHING)

**perfect hash function** for a set S is a [hash function](http://en.wikipedia.org/wiki/Hash_function) that maps distinct elements in S to a set of integers, with no [collisions](http://en.wikipedia.org/wiki/Hash_collision). A perfect hash function has many of the same [applications](http://en.wikipedia.org/wiki/Hash_function#Applications) as other hash functions, but with the advantage that no collision resolution has to be implemented

Cuckoo Algorithm:

**Cuckoo hashing** is a scheme in [computer programming](http://en.wikipedia.org/wiki/Computer_programming) for resolving [hash collisions](http://en.wikipedia.org/wiki/Hash_collision) of values of [hash functions](http://en.wikipedia.org/wiki/Hash_function) in a [table](http://en.wikipedia.org/wiki/Hash_table), with [worst-case](http://en.wikipedia.org/wiki/Worst_case_analysis) [constant](http://en.wikipedia.org/wiki/Constant_time) lookup time. The name derives from the behavior of some species of [cuckoo](http://en.wikipedia.org/wiki/Cuckoo), where the cuckoo chick pushes the other eggs or young out of the nest when it hatches; analogously, inserting a new key into a cuckoo hashing table may push an older key to a different location in the table.

## Theory

The basic idea is to use two hash functions instead of only one. This provides two possible locations in the hash table for each [key](http://en.wikipedia.org/wiki/Unique_key). In one of the commonly used variants of the algorithm, the hash table is split into two smaller tables of equal size, and each hash function provides an index into one of these two tables.

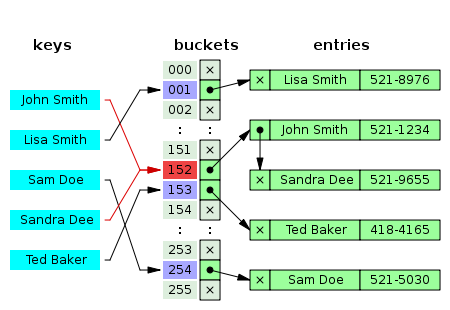
When a new key is inserted, a [greedy algorithm](http://en.wikipedia.org/wiki/Greedy_algorithm) is used: The new key is inserted in one of its two possible locations, "kicking out", that is, displacing, any key that might already reside in this location. This displaced key is then inserted in its alternative location, again kicking out any key that might reside there, until a vacant position is found, or the procedure enters an [infinite loop](http://en.wikipedia.org/wiki/Infinite_loop). In the latter case, the [hash table](http://en.wikipedia.org/wiki/Hash_table) is rebuilt [in-place](http://en.wikipedia.org/wiki/In-place_algorithm) using new [hash functions](http://en.wikipedia.org/wiki/Hash_function):

## Collision resolution

Hash [collisions](http://en.wikipedia.org/wiki/Collision_(computer_science)) are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,500 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the [birthday problem](http://en.wikipedia.org/wiki/Birthday_problem) there is a 95% chance of at least two of the keys being hashed to the same slot.

Therefore, most hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below.

### Separate chaining

[](http://en.wikipedia.org/wiki/File:Hash_table_5_0_1_1_1_1_1_LL.svg)

[http://bits.wikimedia.org/static-1.21wmf12/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Hash_table_5_0_1_1_1_1_1_LL.svg)

Hash collision resolved by separate chaining.

In the method known as *separate chaining*, each bucket is independent, and has some sort of [list](http://en.wikipedia.org/wiki/List_(abstract_data_type)) of entries with the same index. The list grows

So Basically with this approach of chaning we can support Elemets are more then the number of the buckets by having a list of the elements with the each of the bucket.

### Open addressing

[](http://en.wikipedia.org/wiki/File:Hash_table_5_0_1_1_1_1_0_SP.svg)

[http://bits.wikimedia.org/static-1.21wmf12/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Hash_table_5_0_1_1_1_1_0_SP.svg)

Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith".

In another strategy, called [open addressing](http://en.wikipedia.org/wiki/Open_addressing), all entry records are stored in the separate Hash if it gets a collision the Function will go down/Up the list for an Empty Position based on the Hash element. So now during the search same Algorithm to reach its bucket and then go down/Up According to that.

**Linear probing** is a scheme in [computer programming](http://en.wikipedia.org/wiki/Computer_programming) for resolving [hash collisions](http://en.wikipedia.org/wiki/Hash_collision) of values of [hash functions](http://en.wikipedia.org/wiki/Hash_function) by sequentially searching the [hash table](http://en.wikipedia.org/wiki/Hash_table) for a free location.[[1]](http://en.wikipedia.org/wiki/Linear_probing#cite_note-1) This is accomplished using two values - one as a starting value and one as an interval between successive values in [modular arithmetic](http://en.wikipedia.org/wiki/Modular_arithmetic). The second value, which is the same for all keys and known as the *stepsize*, is repeatedly added to the starting value until a free space is found, or the entire table is traversed. (In order to traverse the entire table the stepsize should be [relatively prime](http://en.wikipedia.org/wiki/Relatively_prime) to the arraysize, which is why the array size is often chosen to be a prime number.)

newLocation = (startingValue + stepSize) % arraySize

## Quadratic Probing Insertion

The problem, here, is to insert a key at an available key space in a given Hash Table using quadratic probing.

**Quadratic probing** is an open addressing scheme in [computer programming](http://en.wikipedia.org/wiki/Computer_programming) for resolving collisions in [hash tables](http://en.wikipedia.org/wiki/Hash_table)—when an incoming data's hash value indicates it should be stored in an already-occupied slot or bucket. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary [quadratic polynomial](http://en.wikipedia.org/wiki/Quadratic_polynomial) until an open slot is found.

For a given hash value, the indices generated by [linear probing](http://en.wikipedia.org/wiki/Linear_probing) are as follows:

H + 1 , H + 2 , H + 3 , H + 4 , ... , H + k

This method results in primary clustering, and as the cluster grows larger, the search for those items hashing within the cluster becomes less efficient.

An example sequence using quadratic probing is:

H + 1^2 , H + 2^2 , H + 3^2 , H + 4^2 , ... , H + k^2

Quadratic probing can be a more efficient algorithm in a closed hash table, since it better avoids the clustering problem that can occur with linear probing, although it is not immune. It also provides good memory caching because it preserves some [locality of reference](http://en.wikipedia.org/wiki/Locality_of_reference); however, linear probing has greater locality and, thus, better cache performance.

### Algorithm to Insert key in Hash Table

1. Get the key k

2. Set counter j = 0

3. Compute hash function h[k] = k % SIZE

4. If hashtable[h[k]] is empty

(4.1) Insert key k at hashtable[h[k]]

(4.2) Stop

Else

(4.3) The key space at hashtable[h[k]] is occupied, so we need to find the next available key space

(4.4) Increment j

(4.5) Compute new hash function h[k] = ( k + j \* j ) % SIZE

(4.6) Repeat Step 4 till j is less than SIZE of hash table

5. The hash table is full

6. Stop

### [[edit](http://en.wikipedia.org/w/index.php?title=Quadratic_probing&action=edit&section=4)]C function for Key Insertion

intquadratic\_probing\_insert(int\*hashtable,int key,int\*empty)

{

*/\* hashtable[] is an integer hash table; empty[] is another array which indicates whether the key space is occupied;*

*If an empty key space is found, the function returns the index of the bucket where the key is inserted, otherwise it*

*returns (-1) if no empty key space is found \*/*

int j =0,hk;

hk= key % SIZE;

while(j < SIZE)

{

if(empty[hk]==1)

{

hashtable[hk]= key;

empty[hk]=0;

return(hk);

}

j++;

hk=(key + j \* j)% SIZE;

}

return(-1);

}

The hash code

struct Node

{

char key[size];

Node \* next;

char name[];

}

structhash\_table

{

 NODE\*\* table; // Table of Node pointer

inttable\_size;

}

Hashtable::Hashtable(intT)

{

   size = 0;

   table\_size = T;

   table = newNODE\*[table\_size];

   for(inti=0; i<table\_size; i++)

   {

      table[i] = NULL;

   }

}

\\* \*\

bool contains(char \* key)

{

find(key)

}

// Find is used to convert the Key into an index and return the index Node or NuLLwahatevr is found

find(char \* key)

{

 intbucket = hashString(Key);

   NODE\* temp = table[bucket];

   while(temp != NULL)

   {

      if(strcmp(Key, temp->Key) == 0)

      {

         returntemp;

      }

      temp = temp->next;

   }

   returnNULL;

}

//key to index conversion called by find and the output is copied back

longHashtable::hashString(char\* Key)

{//start hashString

   intn = strlen(Key);

   longh = 0;

   for(inti=0; i<n; i++)

   {

      //To get almost fair distributions of nodes over the array

      h = (h << 3) ^ Key[i];

   }

    returnabs(h % table\_size );

}//en

Main function to add an element

boolHashtable::put(NODE \*N)

{//start put if the Key added is found then no need to enter the node in hash

   if(find(N->Key) != NULL)

   {

      returnfalse;

   }

//Addition of the new node

   NODE\* entry = newNODE(N->Key, N->FullName);

// Create a Key and intialise the Bucket

//hash string is the Bucket Index

   intbucket = hashString(N->Key);

   entry->next = table[bucket];

//

   table[bucket] = entry;

   size++;

   returntrue;

}//

voidHashtable::initIterator()

{//start initIterator

   current\_entry = NULL;

   current\_index = table\_size;

   for(inti=0; i<table\_size; i++)

   {

      if(table[i] == NULL)

      {

          continue;

      }

      else

      {

         current\_entry = table[i];

         current\_index = i;

         break;

      }

   }

}//end initIterator

boolHashtable::hasNext()

{

   if(current\_entry == NULL)

   {

      returnfalse;

   }

   else

   {

      returntrue;

   }

}

voidHashtable::getNextKey(char\* Key)

{

   if(current\_entry == NULL)

   {

      Key[0] = '\0';

      return;

   }

   strcpy(Key, current\_entry->Key);

   if(current\_entry->next != NULL)

   {

      current\_entry = current\_entry->next;

   }

   else

   {

     for(inti=current\_index+1; i<table\_size; i++)

     {

        if(table[i] == NULL)

        {

           continue;

        }

        current\_entry = table[i];

        current\_index = i;

        return;

     }

     current\_entry = NULL;

     current\_index = table\_size;

   }

}

voiddispAll(Hashtable\* hashtable);

intmain()

{

   chartemp1[SIZE\_KEY];

   Hashtable\* hashtable = newHashtable();

   NODE N1("389","Mariam","8216734",22123.267);

   if(!hashtable->contains(N1.Key))

   {

      cout<< "\nAdding node:  ";

      disp(&N1);

      hashtable->put(&N1);

   }

   strcpy(N1.Key, "314");

   strcpy(N1.FullName, "Zeki");

   strcpy(N1.Tele\_No, "8765623");

   N1.Salary = 98124.567;

   if(!hashtable->contains(N1.Key))

   {

      cout<< "\nAdding node:  ";

      disp(&N1);

      hashtable->put(&N1);

   }

   strcpy(N1.Key, "320");

   strcpy(N1.FullName, "Murad");

   strcpy(N1.Tele\_No, "7231144");

   N1.Salary = 19834.575;

   if(!hashtable->contains(N1.Key))

   {

      cout<< "\nAdding node:  ";

      disp(&N1);

      hashtable->put(&N1);

   }

# Sorting:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Name** | **Best** | **Average** | **Worst** | **Memory** | **Stable** | **Method** | **Other notes** |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} n^2 | \mathcal{} \log n | Depends | Partitioning | Quicksort is usually done in place with O(log(*n*)) stack space.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] Most implementations are unstable, as stable in-place partitioning is more complex. [Naïve](http://en.wikipedia.org/wiki/Na%C3%AFve_algorithm)variants use an O(*n*) space array to store the partition.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] |
| [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | Depends; worst case is  \mathcal{} n | Yes | Merging | [Highly parallelizable](http://en.wikipedia.org/wiki/Merge_sort#Parallel_processing) (up to O(log(*n*)) using the Three Hungarian's Algorithm or more practically, Cole's parallel merge sort) for processing large amounts of data. |
| [In-place](http://en.wikipedia.org/wiki/In-place) [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) | \mathcal{} - | \mathcal{} - | \mathcal{} {n \left( \log n \right)^2} | \mathcal{} {1} | Yes | Merging | Implemented in Standard Template Library (STL);[[2]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-2) can be implemented as a stable sort based on stable in-place merging.[[3]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-3) |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {1} | No | Selection |  |
| [Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Insertion | O(*n* + *d*), where *d* is the number of [inversions](http://en.wikipedia.org/wiki/Permutation_groups#Transpositions.2C_simple_transpositions.2C_inversions_and_sorting) |
| [Introsort](http://en.wikipedia.org/wiki/Introsort) | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} \log n | No | Partitioning & Selection | Used in several [STL](http://en.wikipedia.org/wiki/Standard_Template_Library) implementations |
| [Selection sort](http://en.wikipedia.org/wiki/Selection_sort) | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | No | Selection | Stable with O(n) extra space, for example using lists.[[4]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-4)Used to sort this table in Safari or other Webkit web browser.[[5]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-5) |
| [Timsort](http://en.wikipedia.org/wiki/Timsort) | \mathcal{} {n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} n | Yes | Insertion & Merging | \mathcal{} {n}  comparisons when the data is already sorted or reverse sorted. |
| [Shell sort](http://en.wikipedia.org/wiki/Shell_sort) | \mathcal{} n | \mathcal{} n (\log n)^2  or  \mathcal{} n^{3/2} | Depends on gap sequence; best known is \mathcal{} n (\log n)^2 | \mathcal{} 1 | No | Insertion | Small code size, no use of call stack, reasonably fast, useful where memory is at a premium such as embedded and older mainframe applications |
| [Bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging | Tiny code size |
| [Binary tree sort](http://en.wikipedia.org/wiki/Binary_tree_sort) | \mathcal{} n | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} n | Yes | Insertion | When using a [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) |
| [Cycle sort](http://en.wikipedia.org/wiki/Cycle_sort) | — | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | No | Insertion | In-place with theoretically optimal number of writes |
| [Library sort](http://en.wikipedia.org/wiki/Library_sort) | — | \mathcal{} {n \log n} | \mathcal{} n^2 | \mathcal{} n | Yes | Insertion |  |
| [Patience sorting](http://en.wikipedia.org/wiki/Patience_sorting) | — | — | \mathcal{} n \log n | \mathcal{} n | No | Insertion & Selection | Finds all the [longest increasing subsequences](http://en.wikipedia.org/wiki/Longest_increasing_subsequence) within O(*n*log *n*) |
| [Smoothsort](http://en.wikipedia.org/wiki/Smoothsort) | \mathcal{} {n} | \mathcal{} {n \log n} | \mathcal{} {n \log n} | \mathcal{} {1} | No | Selection | An [adaptive sort](http://en.wikipedia.org/wiki/Adaptive_sort) - \mathcal{} {n}  comparisons when the data is already sorted, and 0 swaps. |
| [Strand sort](http://en.wikipedia.org/wiki/Strand_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} n | Yes | Selection |  |
| [Tournament sort](http://en.wikipedia.org/wiki/Tournament_sort) | — | \mathcal{} n \log n | \mathcal{} n \log n | \mathcal{} n[[6]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-6) |  | Selection |  |
| [Cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging |  |
| [Comb sort](http://en.wikipedia.org/wiki/Comb_sort) | \mathcal{} n | \mathcal{} n \log n | \mathcal{} n^2 | \mathcal{} {1} | No | Exchanging | Small code size |
| [Gnome sort](http://en.wikipedia.org/wiki/Gnome_sort) | \mathcal{} n | \mathcal{} n^2 | \mathcal{} n^2 | \mathcal{} {1} | Yes | Exchanging | Tiny code size |
| [Bogosort](http://en.wikipedia.org/wiki/Bogosort) | \mathcal{} n | \mathcal{} n \cdot n! | \mathcal{} {n \cdot n! \to \infty} | \mathcal{} {1} | No | Luck | Randomly permute the array and check if sorted. |

The following table describes [integer sorting](http://en.wikipedia.org/wiki/Integer_sorting) algorithms and other sorting algorithms that a

Comparisions between the Algos

when to use :

Search times:

worst case time:

benifts:

use:

Comparisions between the Algos

when to use :

Search times:

worst case time:

benifts:

use:

ALGO: RAdix SORT:

# Radix Sort

By Eric Suh

Radix Sort is a clever and intuitive little sorting algorithm. Radix Sort puts the elements in order by comparing the **digits of the numbers**. I will explain with an example.

Consider the following 9 numbers:

493   812   715   710   195   437   582   340   385

We should start sorting by comparing and ordering the **one's** digits:

|  |  |
| --- | --- |
| **Digit** | **Sublist** |
| 0 | 340 710 |
| 1 |  |
| 2 | 812 582 |
| 3 | 493 |
| 4 |  |
| 5 | 715 195 385 |
| 6 |  |
| 7 | 437 |
| 8 |  |
| 9 |  |

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340   710   812   582   493   715   195   385   437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit: If we get two elemnts in same bucket the previous sorted Array is taken into consideration

(582,385) now this results in same bucket so look back at the 0 digit Array so (582,385)

|  |  |
| --- | --- |
| **Digit** | **Sublist** |
| 0 |  |
| 1 | 710 812 715 |
| 2 |  |
| 3 | 437 |
| 4 | 340 |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 | 582 385 |
| 9 | 493 195 |

Now the sublists are gathered in order from 0 to 9:

710   812   715   437   340   582   385   493   195

Finally, the sublists are created according to the **hundred's** digit:

|  |  |
| --- | --- |
| **Digit** | **Sublist** |
| 0 |  |
| 1 | 195 |
| 2 |  |
| 3 | 340 385 |
| 4 | 437 493 |
| 5 | 582 |
| 6 |  |
| 7 | 710 715 |
| 8 | 812 |
| 9 |  |

At last, the list is gathered up again:

195   340   385   437   493   582   710   715   812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

Comparisions between the Algos:

when to use :set of bins: each bin must be large enough to accommodate the whole array, so RadixSort can be very expensive in its memory usage! so to be used when memory is not a constarint.

NAME: Merge Sort

ALGO:

To sort the entire sequence A[1 .. n], make the initial call  to the procedure MERGE-SORT (A, 1, n).

MERGE-SORT (A, start, *end*)

1.     IF start < end                                                    // Check for base case  
2.         THEN med = FLOOR[(start + end)/2]                 // Divide step  
3.                 MERGE (A, start, med)                          // Conquer step.  
4.                 MERGE (A, med + 1, end)                     // Conquer step.  
5.                 MERGE (A, start, med, end)                       // Conquer step.

4 3 2 1

As the algo will be in recursion this

the 3rd Merge Line will get stopped called till start = end ; at that place the array is having 1 elemnt and its sorted.

now [4,3] and [2,1]

then the 4th line Merge which will call which will sort the full arrays

[3, 4] [1,2]

basically in this case it will call line 5 which is for the last merge and it will return the full array from 2 sorted arrays

[1 2 3 4]

SUDO CODE:

r

void sort(intarr[],intlow,intmid,int high){

inti,j,k,l,b[20];

l = low;

i=low;

j=mid+1;

//Create a new array from the 2 arrays by aranging the lower of left /right arrays,, then compare each sub elemnts

//of thesmaller array to the higher array and put it in correct position.

while((l<=mid)&&(j<=high)) // from low to mid and med +1 to high

{

if(arr[l]<=arr[j]) // mid +1 elemnt is gt or equal to low thentheseelmtns are sorted

{

b[i]=arr[l]; // then put left lower in the new array

l++; // incremnt the left array lower

}

else

{

b[i]=arr[j]; // else put the right lower in the new array

j++;

}

i++; //icremnt the new array elemnt

}//while end

// Now we have a sorted half of the array till the middle/end for this the mid to higher are already sorted

// by the previous loop just add them in the in new array

if(l>mid) //

{

for(k=j;k<=high;k++)

{

b[i]=arr[k];

i++;

}

}

else // lower list is already sorted copy

{

for(k=l;k<=mid;k++)

{

b[i]=arr[k];

i++;

}

}

for(k=low;k<=high;k++)

{

arr[k]=b[k];

}

}

void partition(intarr[],intlow,int high)

{

int mid;

if(low<high)

{

mid=(low+high)/2;

partition(arr,low,mid);

partition(arr,mid+1,high);

sort(arr,low,mid,high);

}

}

Search times:

O(n) (n = number of the elements merged)

worst case time: T(n) = Θ(n lg n).

Reminder: lg n stands for log2 n.

irst, let's consider the number of recursive calls, as this will shed some light onto our understanding of the list merge operation. Each recursive call will either be a base case, or will result in two future recursive calls. The first call starts off by making two calls; each of those makes four, and so forth. What does this sound like?   
  
If you thought, "a binary tree", then you're absolutely right. An easy way to visualize merge sort is as a tree of recursive calls. To save a bit of space, I will use m(lower, upper) to indicate merge sort called from element lower to element upper. For instance, m(0, n-1) would be the merge sort call for an array of size n in C/C++.

m(0, 3)

/ \

/ \

/ \

m(0, 1) m(2, 3)

/ \ / \

/ \ / \

m(0, 0) m(1, 1) m(2, 2) m(3, 3)

So we see that for an array of four elements, we have a tree of depth three. Now let's say we doubled the number of elements in the array to eight; each merge sort at the bottom of this tree would now have double the number of elements -- two rather than one. This means we'd need one additional recursive call at each element. This suggests that the total depth of the tree is log(n) + 1, the number of times we need to halve the number of elements in the array to reach the base case.   
  
Now, what about the amount of work done at each recursive call? At first, you might think that every merge in the tree should equate to O(n) time, but this is incorrect. At each level, the number of elements is being dramatically reduced; at the bottom branch, it is certainly not taking O(n) time to perform a non-operation. At the level where the results of the base case are being merged (at depth 1 in the above tree), each merge sort call is merging exactly half the list. At the root node is the only time the entire list is merged together at a single node.

benifts:

Compared to insertion sort [Θ(n2) worst-case time], merge sort is faster. Trading a factor of n for a factor of lg n is a good deal. On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sorts.

Summary:

**Summary** Merge sort is a fast, stable sorting routine with guaranteed O(n\*log(n)) efficiency. When sorting arrays, merge sort requires additional scratch space proportional to the size of the input array. Merge sort is relatively simple to code and offers performance typically only slightly below that of quicksort.

Quick Sort:

The quiik short results in quickest short   
Quicksort's running time depends on the result of the partitioning routine - whether it's balanced or unbalanced. This is determined by the**pivot** element used for partitioning. If the result of the partition is unbalanced, quicksort can run as slowly as insertion sort; if it's balanced, the algorithm runs asymptotically as fast as merge sort. That is why picking the "best" pivot is a crucial design decision.

***Median-of-Three Way:*** best case partitioning would occur if PARTITION produces two subproblems of almost equal size - one of size [n/2] and the other of size [n/2]-1. In order to achieve this partition, the pivot would have to be the median of the entire input; unfortunately this is hard to calculate and would consume much of the time, slowing down the algorithm considerably. A decent estimate can be obtained by choosing three elements randomly and using the median of these three as the pivot.

*Short Example of a Quicksort Routine* (Pivots chosen "randomly")

Input: [13 81 92 65 43 31 57 26 75 0]

Pivot: 65

Partition: [ 13 0 26 43 31 57] 65 [ 92 75 81]

Pivot: 31 81

Partition: [13 0 26] 31 [43 57] 65 [75] 81 [92]

Pivot: 13

Partition: [0] 13 [26] 31 [43 57] 65 [75] 81 [92]

Combine: [0 13 26] 31 [43 57] 65 [75 81 92]

Combine: [0 13 26 31 43 57] 65 [75 81 92]

Combine: [0 13 26 31 43 57 65 75 81 92]

Quicksort, like [mergesort](http://www.cprogramming.com/tutorial/computersciencetheory/mergesort.html), is a [divide-and-conquer](http://www.cprogramming.com/cgi-bin/glossary/glossary.pl?state=lookup&word=divide+and+conquer) [recursive](http://www.cprogramming.com/tutorial/lesson16.html) algorithm. The basic divide-and-conquer process for sorting a subarrayS[p..r] is summarized in the following three easy steps: 

**Divide:** Partition S[p..r] into two subarrays S[p..q-1] and S[q+1..r] such that each element of S[p..q-1] is less than or equal to S[q], which is, in turn, less than or equal to each element of S[q+1..r]. Compute the index q as part of this partitioning procedure   
  
**Conquer:** Sort the two subarraysS[p...q-1] and S[q+1..r] by recursive calls to quicksort.   
  
**Combine:** Since the subarrays are sorted in place, no work is needed to combing them: the entire array S is now sorted.

Before a further discussion and analysis of quicksort a presentation of its implementation procedure below: 

QUICKSORT(S, P, r)

1 If p < r

2 then q <- PARTITION(S, p, r)

3 QUICKSORT(S, p, q-1)

4 QUICKSORT(S, q+1, r)

The idea of quick sort is to select a position (Pivot) randomly and to sort the full array on basis of the pivot in two halves . Left (less then pivot and right bigger ). the Pi

suedo code:

voidquickSort(int numbers[], intarray\_size)

{

  q\_sort(numbers, 0, array\_size - 1);

}

voidq\_sort(int numbers[], int left, int right)

{

  int pivot, l\_hold, r\_hold;

  l\_hold = left;

  r\_hold = right;

  pivot = numbers[left]; // **the 1st elemnt is Pivot**

  while (left < right)

  {

    while ((numbers[right] >= pivot) && (left < right))

      right--; **give any right elemnt less then Pivot break the loop**

    if (left != right)

    {

      numbers[left] = numbers[right]; **Assign that right elemnt to 1st Left(we stored it as pivot already) and incrmnt left**

      left++;

    }

    while ((numbers[left] <= pivot) && (left < right))

      left++;**give any left elemnt greater then Pivot break the loop**

    if (left != right)

    {

      numbers[right] = numbers[left];**that left elemnt to right elmnt replaced and decrmnt right**

      right--;

    }

  }

  numbers[left] = pivot; // **the Pivot will replace the latest left(this is at a replaced value)**

  pivot = left;// **Take the pivot as left INDEX this time**

  left = l\_hold;

  right = r\_hold;

**//From the old right n left find th sort again wd this pivot**

  if (left < pivot)

    q\_sort(numbers, left, pivot-1);

  if (right > pivot)

    q\_sort(numbers, pivot+1, right);

}

worst case time: T(n^2)

Sorting time:

benifts:

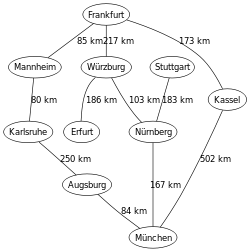
use:

# Searching:

DFS

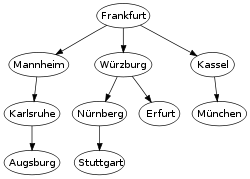
BFS:

## Algorithm

[](http://en.wikipedia.org/wiki/File:MapGermanyGraph.svg)

[http://bits.wikimedia.org/static-1.21wmf10/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:MapGermanyGraph.svg)

An example map of [Germany](http://en.wikipedia.org/wiki/Germany) with some connections between cities

[](http://en.wikipedia.org/wiki/File:GermanyBFS.svg)

[http://bits.wikimedia.org/static-1.21wmf10/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:GermanyBFS.svg)

The breadth-first tree obtained when running BFS on the given map and starting in [Frankfurt](http://en.wikipedia.org/wiki/Frankfurt)

|  |
| --- |
| [**Graph**](http://en.wikipedia.org/wiki/Graph_traversal)**and**[**tree search algorithms**](http://en.wikipedia.org/wiki/Tree_traversal) |
| * [α–β](http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning) * [A\*](http://en.wikipedia.org/wiki/A*_search_algorithm) * [B\*](http://en.wikipedia.org/wiki/B*) * [Beam](http://en.wikipedia.org/wiki/Beam_search) * [Bellman–Ford](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm) * [Best-first](http://en.wikipedia.org/wiki/Best-first_search) * [Bidirectional](http://en.wikipedia.org/wiki/Bidirectional_search) * [Borůvka](http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm) * [Branch & bound](http://en.wikipedia.org/wiki/Branch_and_bound) * **BFS** * [British Museum](http://en.wikipedia.org/wiki/British_Museum_algorithm) * [D\*](http://en.wikipedia.org/wiki/D*) * [DFS](http://en.wikipedia.org/wiki/Depth-first_search) * [Depth-limited](http://en.wikipedia.org/wiki/Depth-limited_search) * [Dijkstra](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm) * [Edmonds](http://en.wikipedia.org/wiki/Edmonds%27_algorithm) * [Floyd–Warshall](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) * [Hill climbing](http://en.wikipedia.org/wiki/Hill_climbing) * [Iterative deepening](http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search) * [Kruskal](http://en.wikipedia.org/wiki/Kruskal%27s_algorithm) * [Johnson](http://en.wikipedia.org/wiki/Johnson%27s_algorithm) * [Lexicographic BFS](http://en.wikipedia.org/wiki/Lexicographic_breadth-first_search) * [Prim](http://en.wikipedia.org/wiki/Prim%27s_algorithm) * [Uniform-cost](http://en.wikipedia.org/wiki/Uniform-cost_search) |
| **Listings** |
| * [*Graph algorithms*](http://en.wikipedia.org/wiki/Category:Graph_algorithms) * [*Search algorithms*](http://en.wikipedia.org/wiki/Category:Search_algorithms) * [*List of graph algorithms*](http://en.wikipedia.org/wiki/List_of_algorithms#Graph_algorithms) |
| **Related topics** |
| * [Dynamic programming](http://en.wikipedia.org/wiki/Dynamic_programming) * [Graph traversal](http://en.wikipedia.org/wiki/Graph_traversal) * [Tree traversal](http://en.wikipedia.org/wiki/Tree_traversal) * [Search games](http://en.wikipedia.org/wiki/Search_game) |
| * [v](http://en.wikipedia.org/wiki/Template:Graph_search_algorithm)      * [t](http://en.wikipedia.org/wiki/Template_talk:Graph_search_algorithm)      * [e](http://en.wikipedia.org/w/index.php?title=Template:Graph_search_algorithm&action=edit) |

The algorithm uses a [queue](http://en.wikipedia.org/wiki/Queue_(data_structure)) data structure to store intermediate results as it traverses the graph, as follows:

1. Enqueue the root node
2. Dequeue a node and examine it
   * If the element sought is found in this node, quit the search and return a result.
   * Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

**Note**: Using a [stack](http://en.wikipedia.org/wiki/Stack_(data_structure)) instead of a queue would turn this algorithm into a [depth-first search](http://en.wikipedia.org/wiki/Depth-first_search).

### [[edit](http://en.wikipedia.org/w/index.php?title=Breadth-first_search&action=edit&section=2)]Pseudocode

**Input**: A graph *G* and a root *v* of G

1 **procedure** BFS(*G*,*v*):

2 create a queue *Q*

3 enqueue*v* onto *Q*

4 mark *v*

5 **while***Q* is not empty:

6 *t* ← Q.dequeue()

7 **if***t* is what we are looking for:

8 return *t*

9 **for all** edges e in G.adjacentEdges(t) **do**

12 *u* ← G.adjacentVertex(*t*,*e*)

13 **if***u* is not marked:

14 mark *u*

15 enqueue*u* onto *Q*

16 return *none*

# Miscellneious:

------------------------------------------------------------------------------------------------------------------------------------------

Knaosack problem:

Problem weights and benifits to hav eth emaxbenifits

dynamic Programing