COSC262 Assignment 1 — Data Structures and Sorting

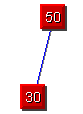
## Problem 1

**(1). Insert the above numbers into a binary search tree (BST). Show the tree after each insertion and the path from the root to the insertion point for each tree.**

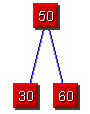
Firstly, pick up 10 numbers smaller than 100 for make binary search tree, and they are

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 50 | 30 | 60 | 40 | 55 | 10 | 20 | 5 | 70 | 80 |

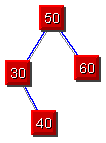
Assume the root is 50, and start at root. First insert number is 30. Because of the binary search tree so the number 30 should go to the left-child, which showing below:



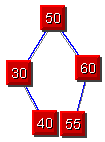
Secondly, add number 60 into the tree and it should go to the right-child position



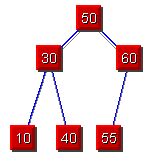
The number between 50 and 30 is 40, so now add the number 40 into the binary tree. Because that 40 is smaller than 50, so it should go to left-side of the tree. But 40 is bigger than 30 so it should be the right-child of number 30.



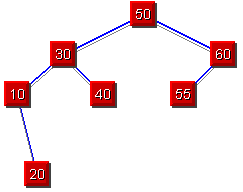
The number between 50 and 60 is 55, so now add the number 55 into the binary tree, because 55 is bigger than 50, it should go to right-side of the tree, and it is smaller than 60 so it should be the left-child of number 60.



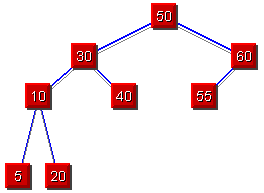
The next node which adds into the left side should be number 10, because 10 is smaller than 50 so it should be go to left-side of the tree, but also 10 is smaller than 30. Therefore, it should be the left-child of number 30.



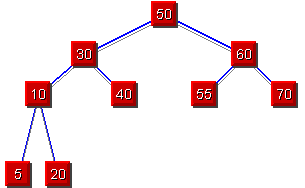
The number between 10 and 30 is number 20, so add 20 into the tree. However, the 20 is smaller than 50 so it go to the left-side of the tree, it’s also smaller than 30 then keep check the left-child of 30 which is 10, but it is bigger than 10 then it should be the right-child of 10.



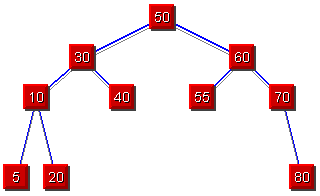
After add number 5 into the list, because number 5 is smaller than 50 so it should be to the left-side of the tree, it also smaller than 30 and 10. So it should be the left-child of number 10.



According to the number of the list, there are only two numbers left. They are 70 and 80, let’s add number 70 into the tree, because 70 is bigger than 50 so it should go to the right-side of the tree, but also it bigger than 60 and the left-child of 60 which is 55, so number 70 should be the right-child of 60.



The last number 80 add into the tree, it should be right-child of 70. Because that number 80 is bigger than 50 so it should go to the right-side of the tree. But also it is bigger than 55, 60 and 70. Therefore, 80 is the right-child of number 70.



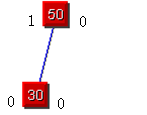
**(2) Insert them into an AVL tree. Show the tree after each insertion and the tree after a rotation if a rotation is required. Balance labels (L, E, R) must be shown.**

According the numbers of the list from question 1, and assume that root is number 50.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 50 | 30 | 60 | 40 | 55 | 10 | 20 | 5 | 70 | 80 |

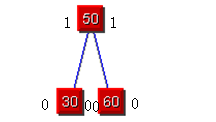
1st Step:

Add number 50 and make it be the root, then add number 30, because it’s smaller than 50 so it should be the left-child of 50.



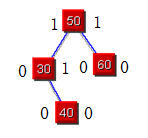
2nd Step:

Add number number 60, because it’s bigger than root, so it should be the right-child of number 50. So far the tree is balanced, no rotation required.



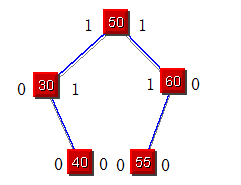
3rd Step:

Add number 40 into the AVL tree, because it’s smaller than 50 so it should goes to left-side of the tree. So after check the next node which is number 30, because 30 is smaller than 40, so 40 should be the right-child of number 30. So far the tree is balanced and no rotation required.



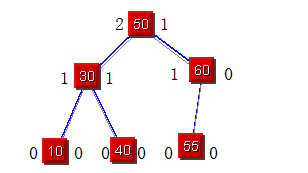
4th Step:

The next number which needs to add into the AVL tree is 55, so start from the root. It is bigger than the root so it should go to right-side of the tree, and it’s smaller than number 60. So number 55 should be the left-child of number 60. So far the tree is still balanced and no rotation required.



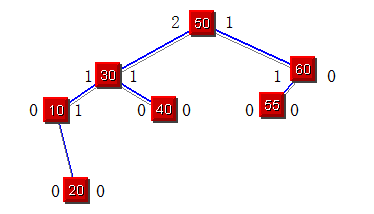
5th Step:

Add number 10 into the AVL tree, since it is smaller than the root so it should goes to the left-side to the tree. Also it is smaller than 30, so it should be the left-child of 30. The tree is balanced and no rotation required.



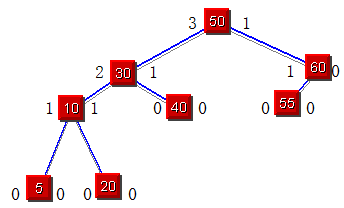
6th Step:

Add number 20 into the AVL tree. Checking from the root, 20 is smaller than 50, so it goes to the left-side of the tree. And also it smaller than 30 then check the left-child, but it is bigger than the left-child which is 10, so number 20 should be the right-child of number 10. So far the tree is balanced and no rotation required.



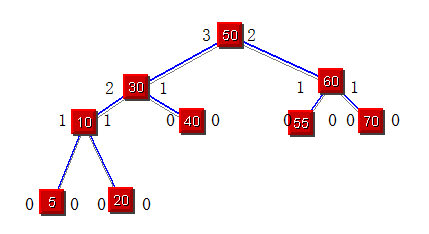
7th Step:

Add number 5 into the AVL tree. Checking start at the root, it’s smaller than the root so should goes to the left-side of the tree. Number 5 is smaller than 30 and 10, so it should be the left-child of number 10. Until now the tree is on balanced and no rotation required.



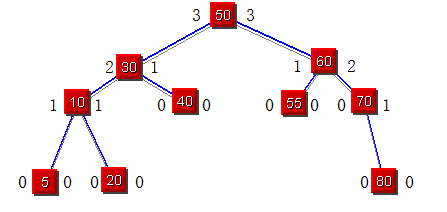
8th Step:

The next number is 70, and adds it into the AVL tree. Checking the root, it is bigger than 50, so it goes right-side of the tree. But also it is bigger than number 60, so number 70 should be the right-child of number 60. So far the tree is balanced and no rotation required.



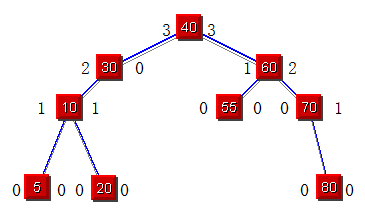
9th Step:

The last number of the list is 80, it adds into the AVL tree. Checking start at the root, 80 is bigger than the root so it goes to right-side of the tree. And it is also bigger than 60. So keep checking the right-child of 60, and it is bigger than 70 so it should be the right-child of number 70. At the end, all insertion is done, the check back each nodes they are all on balanced, so no rotation required.

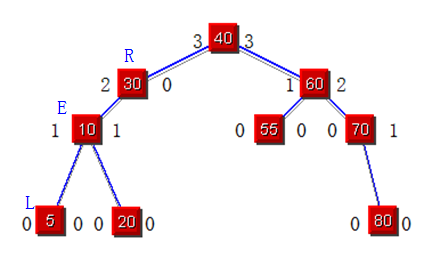


(3)Delete the root twice from the above AVL tree. The AVL property must be kept. Balance labels (L, E, R) must be shown.

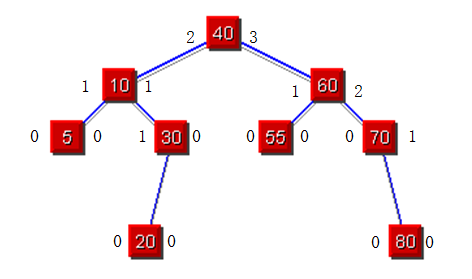
First time deletes the root which is number 50, then 40 will goes up and replace 50 became the root.



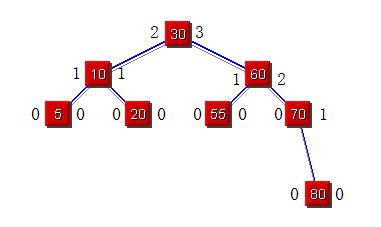
However, the AVL tree is not on balance, so we need to rotate it.



After the LR rotation has been done, the AVL tree is should be like that which showing below.

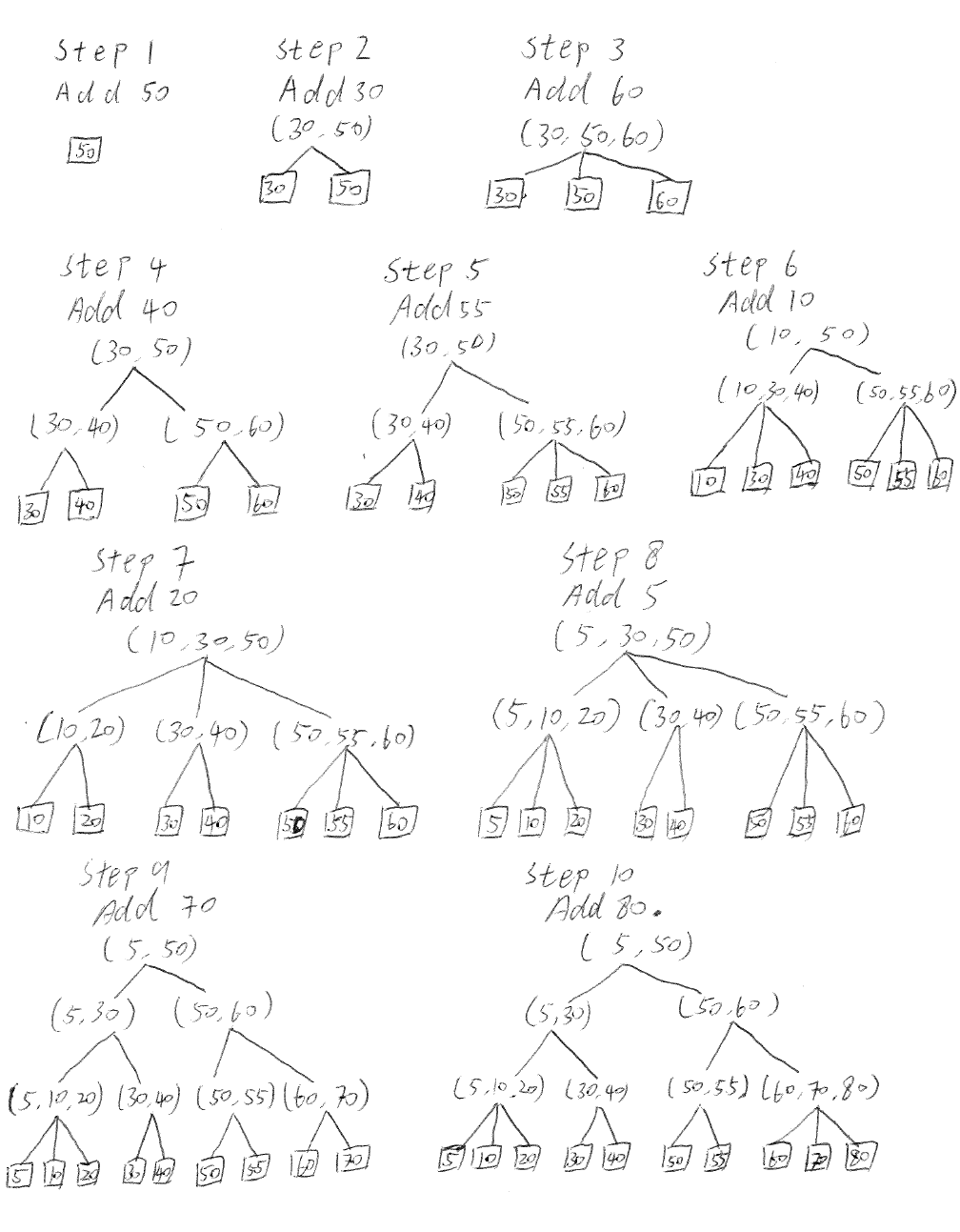


Second time deletes the root which is number 40, and then the right-child of 10 which is number 30 will replace the position of the root. Also the left-child of 30 will goes up and become the right-child of 10, after checking the balancing of the AVL tree.



(4) Insert the above numbers into a 2-3 tree. Show the tree before and after each insertion.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 50 | 30 | 60 | 40 | 55 | 10 | 20 | 5 | 70 | 80 |

**Scanned** 

# Problem 2

1. **Code for binary search tree:**

from time import clock, time

from random import random

## This is to insert into a binary search tree by Python objects

Counter=0

class node:

key=0

left=0

right=0

def insert(x) :

global Counter

p=root

while p!=0 :

Counter+=1

q=p

if x<=p.key:

p=p.left

else: p=p.right

p=node()

p.key=x;

if x<=q.key:

Counter+=1

q.left=p # p inserted to the left of q

else: q.right=p # p inserted to the right of q

def traverse(p):

if p!=0:

p1=p.left

traverse(p1)

t.append(p.key)

p2=p.right

traverse(p2)

n=int(input('input n '))

t=[]

for i in range(0,n): t=t+[random()];

n=len(t)

tt=clock()

x=t[0]

root=node(); node.key=x; node.right=0; node.right=0;

for i in range(1,n):x=t[i];insert(x)

traverse(root)

tt=clock()-tt

print t

print "CPU times: ", tt

print 'Number of Comparesions: ', Counter

print "Finished"

**Code for quicksort:**

# This is quicksort

# sort(left,right) sorts array a from position left to right

# partition(left,right) partitions array a from position left to right

# with pivot x=a[left], and returns m such that after partition

# a[left .. m-1] <= x=a[m] <= a[m+1 .. right]

import random

from time import time, clock

Counter=0

def out(n):

for i in range(1, n+1): print a[i],

print '\n'

def sort(left,right):

if left<right:

m=partition(left,right)

sort(left,m-1)

sort(m+1,right)

def partition(left,right): ## x is the pivot

global Counter

x=a[left]; i=left; j=right+1 ## i goes right, j goes left

while i<j:

j=j-1;

if i==j:

break

while a[j]>=x: ## while a[j]>=x,

Counter += 1

j=j-1; ## j goes left

if i==j : break

if i==j: break

a[i]=a[j] ## a[j] is copied to a[i]

i=i+1

if i==j: break

while a[i]<=x: ## while a[i]<=x,

Counter += 1

i=i+1; ## i goes right

if i==j : break

if i==j: break

a[j]=a[i] ## a[i] is copied to a[j]

a[i]=x ## pivot x settles down at i

return i

# {main program}

n=input('input n ')

a=[]

for i in range(0,n+1): a=a+[int(100\*random.random())]

t=clock()

sort(1,n)

out(n)

print 'CPU time: ',clock()-t

print "Number of Comparesions: ", Counter

n=raw\_input('finished ')

**Code for Mergesort:**

# This is mergesort. mergesort(p,q) sorts array a from position p to q

import random

from time import time, clock

c=0

def out(n):

for i in range(1, n+1):print a[i]

print '\n'

def mergesort(p, q):

if p < q :

m = (p+q) / 2;

mergesort(p, m);

mergesort(m+1, q);

merge(p, m+1, q+1) # This is to merge a[p .. m] and a[m+1 .. q] into b[p .. q]

def merge(p, r, q):

global c ### c is comparison counter

i=p; j=r; k=p;

while i < r and j < q :

c=c+1

if a[i] <= a[j]:

b[k] = a[i]; i=i+1 # if a[i]<=a[j] copy a[i] to b[k]

else : b[k] = a[j]; j=j+1 # else copy a[j] to b[k]

k=k+1

while i < r : # Flash out the remaining elements in the left half to b

b[k]= a[i]; i=i+1; k=k+1

while j < q : # Flash out the remaining elements in the left half to b

b[k] = a[j];

j=j+1; k=k+1;

for k in range(p,q): a[k] = b[k]

# main program

n=input('input n ')

a=[]

for i in range(0,n+1): a=a+[int(100\*random.random())]

b=[]

for i in range(0,n+1): b=b+[0]

t=clock()

mergesort(1, n); out(n)

print 'time ',clock()-t, 'c=',

print 'Comparisons: ',c

n=raw\_input('finished ')

**CPU times (If your computer is fast, larger values of n can be tested. Specify the CPU)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **N** | **1000** | **5000** | **10000** | **15000** | **20000** |
| **BST-sort** | **0.0276732246339** | **0.0906321278602** | **0.202133173461** | **0.317152309217** | **0.45105424908** |
| Quick-sort | **0.00850561507612** | **0.0481473364231** | **0.108993526764** | **0.167422528549** | **0.225243017603** |
| Heap-sort | **0.0541531510083** | **0.143980166657** | **0.111495811386** | **0.171840435874** | **0.234155680544** |
| **Modified heap-sort** | **0.0162544390878** | **0.0537349715799** | **0.110742840421** | **0.169581522979** | **0.230766846214** |
| **Alternate merge-sort** | **0.00914009060423** | **0.0291608724144** | **0.0618618443609** | **0.0963251162932** | **0.132960806572** |

**Number of comparisons**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **N** | **1000** | **5000** | **10000** | **15000** | **20000** |
| **BST-sort** | **11883** | **77604** | **151947** | **246285** | **357833** |
| Quick-sort | **12760** | **81173** | **176965** | **278705** | **383688** |
| Heap-sort | **16816** | **107358** | **234060** | **368786** | **508033** |
| **Modified heap-sort** | **16433** | **104083** | **225429** | **355887** | **491086** |
| **Alternate merge-sort** | **8712** | **55150** | **120238** | **188827** | **260480** |

**(2)Modified heap-sort to ternary heap-sort, code showing below:**

# sort() sorts array a in descending order

# sift(p,q) heapifies array a from position p to q

# heap is a max-heap, that is, maximum at the root

import random

Counter=0

from time import time, clock

def out(n):

for i in range(1, n+1): print a[i],

print '\n'

def swap(i,j): # This swaps a[i] and a[j]

w=a[i]; a[i]=a[j]; a[j]=w

def siftup(p, q): # This is to heapify a when a[p] is wrong

global Counter

y=a[p]; j=p; k=3\*p-1 # a[p] is saved to y

k\_mid = k+1; k\_right = k+2

while k <= q :

z=a[k]

Counter += 1

if k\_mid <= q:

Counter += 1

if z > a[k\_mid]:

z=a[k\_mid]

k=k\_mid

if k\_right <= q:

Counter += 1

if z > a[k\_right]:

z = a[k\_right]

k = k\_right

if y <= z :break

a[j]=z;j=k;k=3\*j-1; k\_mid=k+1; k\_right=k+2

a[j]=y # y settles down at position j

def build\_heap(n):

for i in reversed(range(1,(n+1)/3)): siftup(i, n)

def sort():

build\_heap(n)

for i in reversed(range(2,n+1)) :

swap(1, i) # swap a[1] and a[i]

siftup(1,i-1) # Heapify a from position to position i-1

# {main program}

n=input('input n ')

a=[]

for i in range(0,n+1): a=a+[int(100\*random.random())]

t=clock()

sort()

out(n)

print 'CPU times: ',clock()-t

print 'Number of comparisons: ',Counter

n=raw\_input('finished ')

**(3) Improve merge-sort to alternate merge-sort, code showing below:**

# This is alternate mergesort

# mergesort(p,q) sorts array a from position p to q

# This is alternate mergesort

import random

from time import time, clock

c=0

def out(a, n):

for i in range(1, n+1): print a[i],

print '\n'

def mergesort(p, q, t): ## parameter t controls switching between merge1 and

if p < q : ## merge2

m = (p+q) / 2

mergesort(p, m, -t); # t alternates between + and -

mergesort(m+1, q, -t); # meaning merge1 and merge 2 alternate

if t>0 :

merge1(p, m+1, q+1) # from level to level in recursion.

else:

merge2(p, m+1, q+1)

# merge(p,m+1,q+1) merges the portion of array a from position p to m

# and that from position m+1 to q

def merge1(p, r, q): ## This is the original merge from a to b

global c

i=p; j=r; k=p;

while i < r and j < q :

c=c+1

if a[i] <= a[j]:

b[k] = a[i]; i=i+1

else : b[k] = a[j]; j=j+1

k=k+1

while i < r :

b[k]= a[i];

i=i+1; k=k+1

while j < q :

b[k] = a[j];

j=j+1; k=k+1

def merge2(p, r, q): # This is the merge from b to a

global c

i=p; j=r; k=p;

while i < r and j < q :

c=c+1

if b[i] <= b[j]:

a[k] = b[i]; i=i+1

else : a[k] = b[j]; j=j+1

k=k+1

while i < r :

a[k]= b[i];

i=i+1; k=k+1

while j < q :

a[k] = b[j];

j=j+1; k=k+1

#{ main program }

n=input('input n ')

a=[]

for i in range(0,n+1): a=a+[int(100\*random.random())]

b=[]

for i in range(0,n+1): b=b+[a[i]]

t=clock()

mergesort(1, n, 1)

out(b, n)

print 'CPU time: ',clock()-t

print 'Number of Comparesions: ',c

n=raw\_input('finished ')

**(4)** **Discuss the observed performances, such as the number of comparisons and computing time in (1), (2) and (3) above. Discuss why or why not the gain was achieved.**

According the information above, the binary search tree sort compare with other sort method, this one has longer time for operation than others but heapsort, . O (log n) for average case, worst case is O (n).

The quicksort is O (n log n) for average case, the mergesort is O(n log n) for average case, and the heapsort is O (n log n) for average case. In the average case, quicksort, heapsort, and mergesort might be similar. However, they cannot be similar. The reason for this is the quicksort has worst case O (n^2), the worst case for mergesort and heapsort are O(n log n). Significantly, quicksort is fast when the data is small. If the data is huge, it will take much more times, because in a huge data it has more chance to go the worst case O (n^2).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **N** | **1000** | **5000** | **10000** | **15000** | **20000** |
| Heap-sort | **0.0541531510083** | **0.143980166657** | **0.111495811386** | **0.171840435874** | **0.234155680544** |
| Heap-sort | **16816** | **107358** | **234060** | **368786** | **508033** |
| Modified heap-sort | **0.0162544390878** | **0.0537349715799** | **0.110742840421** | **0.169581522979** | **0.230766846214** |
| Modified heap-sort | **16433** | **104083** | **225429** | **355887** | **491086** |

The heapsort is O (n log n) for average case. However, the ternary heapsort is about 12% faster than the simple variant of binary heapsort. The reasons that ternary heapsort faster that is the ternary heapsort build up three children with one parent in the tree and each time one child compare with other two children, the binary heapsort only build up two children with one parent in the tree and each time one child to compare with other one child. So it will take less time than the binary heapsort. The comparisons between two sort are similar.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **N** | **1000** | **5000** | **10000** | **15000** | **20000** |
| **merge-sort** | **0.0155002975925** | **0.0586914362805** | **0.121973546509** | **0.188282730213** | **0.254046327709** |
| **merge-sort** | **8676** | **55098** | **120188** | **188916** | **260417** |
| **Alternate merge-sort** | **0.00914009060423** | **0.0291608724144** | **0.0618618443609** | **0.0963251162932** | **0.132960806572** |
| **Alternate merge-sort** | **8712** | **55150** | **120238** | **188827** | **260480** |

The method of comparison between mergesort and alternate mergesort are same, so the number of comparisons between two sort are similar or same. However, the time of CPU operation between mergesort and alternate mergesort are different. In the alternate mergesort, it make an alternating directions for the mergesort. If in the small list data, alternate mergesort won’t make much different, because there are only few alternating directions made. If the data is huge then there will be many alternating directions made, then the time of operation will taking less than the original mergesort.