**MIDTERM 3 REVIEW QUESTIONS**

**1. Show me how the type of a template class gets defined in the main() program.**

Templates are a way of making your classes more abstract by letting you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class.

node<string>\* head\_ptr1 = nullptr;

node<string>\*, node<int>\* ect.

**2. How would you define an array of big\_numbers using a template array class like the ones from the sorting codes?**

class calc  
{  
 public:  
 int multiply(int x, int y);  
 int add(int x, int y);  
 };  
int calc::multiply(int x, int y)  
{  
 return x\*y;  
}  
int calc::add(int x, int y)  
{  
 return x+y;  
}

We now have a perfectly harmless little class that functions perfectly well for integers; but what if we decided we wanted a generic class that would work equally well for floating point numbers? We would use a template.

template <class A\_Type> class calc  
{  
 public:  
 A\_Type multiply(A\_Type x, A\_Type y);  
 A\_Type add(A\_Type x, A\_Type y);  
};  
template <class A\_Type> A\_Type calc<A\_Type>::multiply(A\_Type x,A\_Type y)  
{  
 return x\*y;  
}  
template <class A\_Type> A\_Type calc<A\_Type>::add(A\_Type x, A\_Type y)  
{  
 return x+y;  
}

array <big\_number> arrayname;

**3. How could you count the frequency of a number in a binary search tree?**

To do this we would need to travel the tree and have a counter for everytime we see a specific number. The way we travel a tree would be as follows

Depth First Traversals:

(a) Inorder

(b) Preorder

(c) Postorder

Breadth First or Level Order Traversal

Please see [this](http://geeksforgeeks.org/?p=2686) post for Breadth First Traversal.

**Inorder Traversal:**

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder itraversal s reversed, can be used.

Example: Inorder traversal for the above given figure is 4 2 5 1 3.

**Preorder Traversal:**

Algorithm Preorder(tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left-subtree)

3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Pre-order

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see <http://en.wikipedia.org/wiki/Polish_notation> to know why prefix expressions are useful.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

**Postorder Traversal:**

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)

2. Traverse the right subtree, i.e., call Postorder(right-subtree)

3. Visit the root.

Uses of Postorder

Postorder traversal is used to delete the tree. Please see [the question for deletion of tree](http://geeksforgeeks.org/?p=654) for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see <http://en.wikipedia.org/wiki/Reverse_Polish_notation> to for the usage of postfix expressio

**4. How could you tell if 2 binary search trees contained exactly the same numbers?**

eq(t1, t2) = t1.data=t2.data && eq(t1.left, t2.left) && eq(t1.right, t2.right)

The reason is that mismatches are likely to be common, and it is better to detect (and stop comparing) early - before recursing further. Of course, I'm assuming a short-circuit && operator here.

I'll also point out that this is glossing over some issues with handling structurally different trees correctly, and with ending the recursion. Basically, there need to be some null checks for t1.left etc. If one tree has a null .left but the other doesn't, you have found a structural difference. If both have null .left, there's no difference, but you have reached a leaf - don't recurse further. Only if both .left values are non-null do you recurse to check the subtree. The same applies, of course, for .right.

You could include checks for e.g. (t1.left == t2.left), but this only makes sense if subtrees can be physically shared (same data structure nodes) for the two trees. This check would be another way to avoid recursing where it is unnecessary - if t1.left and t2.left are the same physical node, you already know that those whole subtrees are identical.

A C implementation might be...

bool tree\_compare (const node\* t1, const node\* t2)

{

// Same node check - also handles both NULL case

if (t1 == t2) return true;

// Gone past leaf on one side check

if ((t1 == NULL) || (t2 == NULL)) return false;

// Do data checks and recursion of tree

return ((t1->data == t2->data) && tree\_compare (t1->left, t2->left )

&& tree\_compare (t1->right, t2->right));

}

**5. Could we make a template class for binary search trees? Why or why not?**

Certainly we can. Templates are made for keeping things generic. BST can hold any data - Char, int, long anything even your own data type. So this data type can be used in Template class object creation and class can be defined as class BST.

**6. What’s the expected run time for quicksort? Why? What’s the worst run time it can get, and when does this happen?**

Average runtime is O(n log n), Worst run time is O(n^2) when the partitioning routine produces

one subproblem with n 1 elements and one with 0 elements.

**7. What’s the expected run time for mergesort? Why? What does mergesort do that’s inefficient compared to the other sorting algorithms?**

Expected runtime is nlog2(n) in both best and worst cases. Mergesort splits a size-n array in half log2(n) times. It then merges each array; merging 2 arrays of size n/2 to get one array of size n takes n comparisons. Just as there were log2(n) splits, there are log2(n) merges. The downside of mergesort is that each of the little merges requires us to make an extra array to hold the merged data. We copy this array back at each step to the main array and write the merges into the second array. The overhead of this extra array is pretty expensive, for large n.

**8. What’s the best case run time for insertion sort? What produces this run time? What’s the expected run time for insertion sort? What about bubble sort and selection sort?**

# Insertion sort:

Our best case scenario for this would be (o(n)) but this would only happen if the array is already sorted. the worst case is it is completely backwards and that would lead us to a quadradic runtime. O(n^2)

“The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than [quicksort](http://en.wikipedia.org/wiki/Quicksort); indeed, good [quicksort](http://en.wikipedia.org/wiki/Quicksort) implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.”

# Bubble sort:

* **Worst case performance** *O(n²)*
* **Best case performance** *O(n)*
* **Average case performance** *O(n²)*

*Bubble sort has worst-case and average complexity both* [*О*](http://en.wikipedia.org/wiki/Big_o_notation)*(n2), where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of O(n log n). Even other О(n2) sorting algorithms, such as* [*insertion sort*](http://en.wikipedia.org/wiki/Insertion_sort)*, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.*

*The only significant advantage that bubble sort has over most other implementations, even* [*quicksort*](http://en.wikipedia.org/wiki/Quicksort)*, but not* [*insertion sort*](http://en.wikipedia.org/wiki/Insertion_sort)*, is that the ability to detect that the list is sorted is efficiently built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only O(n). By contrast, most other algorithms, even those with better* [*average-case complexity*](http://en.wikipedia.org/wiki/Average-case_complexity)*, perform their entire sorting process on the set and thus are more complex. However, not only does* [*insertion sort*](http://en.wikipedia.org/wiki/Insertion_sort) *have this mechanism too, but it also performs better on a list that is substantially sorted (having a small number of* [*inversions*](http://en.wikipedia.org/wiki/Inversion_(discrete_mathematics))*).*

*Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a reverse-ordered collection.*

# Selection sort

It has [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n*2) time complexity, making it inefficient on large lists, and generally performs worse than the similar [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort). Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

**9. Convert 23415 in base 7 to base 5, using the digit-wise algorithm from class.**

Convert to base 10 first and then convert to base 5 to solve this problem.

**23415 base 7 to base 5 is 143124**

e.g Convert 1024 (or any number) in base 10 to number m in base 2

Start with m = 0

for each digit *d* in the old base *k*, highest to lowest: m = m \* k; m = m + d

1: m = 0 \* 1010

m = 0 + 1

0: m = 1 \* 1010 = 1010

m = 1010 + 0 = 1010

2: m = 1010 \* 1010 = 1100100

m = 1100100 + 10 = 1100110

4: m = 1100110 \* 1010 = 1111111100

m = 1111111100 + 100 = 10000000000

Ex: 100(base 10) = ? in base 2

v(k)/ r

in this case r = 2

100/2 = 50 with a remainder of 0

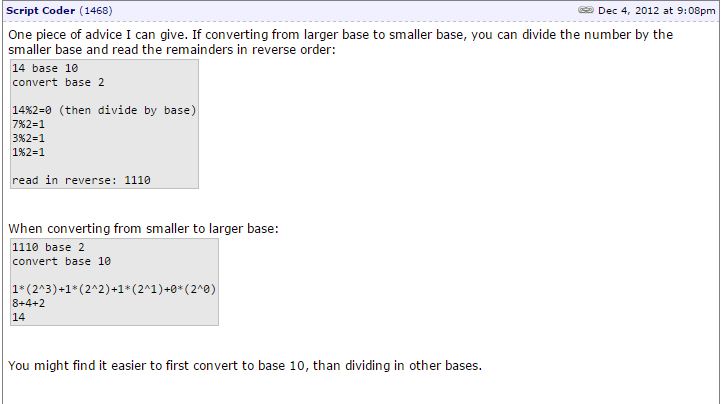
50/2 = 25 with a remainder of 0

25/2 = 12 with a remainder of 1

12/2 = 6 remainder of 0

6/2 =

|  |  |
| --- | --- |
| Number | Binary |
| 0 | 0 |
| 1 | 1 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 1010 |



To understand the above code remember that you must be looking at the remainder, and then when dividing

Convert 10000000000 in base 2 to number m in base 10

Start with m = 0

for each digit d in the old base k, highest to lowest:

m = m \* k

m = m + d

1: m = 0 \* 2 = 0

m = 0 + 1 = 1

0: m = 1 \* 2 = 2

m = 2 + 0 = 2

0: m = 2 \* 2 = 4

m = 4 + 0 = 4

// m doubles 8 more times: 8, 16, 32, 64, 128, 256, 512, 1024!

**10. What is the problem with this code?**

**int& no\_no\_nanette() {**

**int answer = 9;**

**return answer;}**

the variable is being destroyed after the code

**Why is that not a problem with this code?**

**big-number& operator =(const big\_number& m) {**

**…**

**return \*this;**

**}**

we are returning the reference to ‘m’ so it does not get destroyed after code runs.r

**11. Tell me 2 big differences between a copy constructor and an assignment operator (operator =).**

**Justify your answers.**

We use constructor for two purpose one is for dynamic memory allocation of deta member and for initilised the data member..

assignment operator only for copy the data one object to existing object....

simple where ever new object going to come in existance with the help of old one (no metter how it can be by function call or it can be by = operator ) copy constructor call and if object is existing and you want to assign or copy the value then assignment operator overloding will call thats it...

myclass a=b;//here first for a memory will allot then copy data

myclass c;

c=a; //here assignment operator should call object exist

So really what the difference here is just that the copy constructor will create a new object (int..whatever type it is) while the assignment operator will assign an already known variable to the same value that is being stored. Heres another example that should be helpful in understanding this.

|  |
| --- |
| **t2 = t1; // calls assignment operator, same as "t2.operator=(t1);"**  **Test t3 = t1; // calls copy constructor, same as "Test t3(t1);"**  **copy constructor is deep copy** |

**12. Given the code for operator >, what could you do to get operator < for very little work? How would you get operator == from the > and < operators?**

Given the code for >, we could simply do if (!(a>b)) to find a<b. To get ==, we could utilize logic operators such that if(!(a<b)&&!(a>b)) then true;

**13. For the pattern code, how many pattern calls result from calling:**

**pattern(outs, 4, 0);** 2 calls to pattern(2), 7 calls in all

**pattern(outs, 16, 0);** 31 counting pattern(16)

**pattern(outs, 1024, 0);** 2047 counting pattern(1024)

**What formula describes this relationship between the starting n and the number of calls pattern makes?**

# of calls = # of lines = 2\*n-1

**14. What formula describes the relationship between the starting n and the number of stars pattern prints?**

n\*2^n-1

# of lines = 2\*n-1

**15. Given the array below: 6 3 9 8 8 3 1 7 3 9 1**

**Show me the array after one pass (one inner loop) of selection sort has run.**

selection sort finds the smallest value in the list and swaps it with the value in the front of the array

Insertions sort is like a deck of cards you order it at first

merge sort splits them into individual and combines them into sets of 2 then swaps, set of 4 swap.

Here is an example of this sort algorithm sorting five elements:

64 25 12 22 11 // this is the initial, starting state of the array  
  
11 25 12 22 64 // sorted sublist = {11}  
  
11 12 25 22 64 // sorted sublist = {11, 12}  
  
11 12 22 25 64 // sorted sublist = {11, 12, 22}  
  
11 12 22 25 64 // sorted sublist = {11, 12, 22, 25}  
  
11 12 22 25 64 // sorted sublist = {11, 12, 22, 25, 64}

Given an array of elements, find the lowest one and swap it into the first element, we then do this again until the entire array is put together correctly

Therefore after one pass, the array will look like 163988317391

**Show me the array after one pass (one inner loop) of insertion sort has run.**

similar to bubble sort but we do it all at once, after we sort the first one, we then swap the rest of them until everything is in order all in one time through

from my understanding this would probably be easy to do with recursion so I dont really know how far it will go after one pass.

Example: The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the key under consideration is underlined. The key that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1

**3** 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

**2** 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

**1** 2 3 4 5 6 7 9

**Show me the array after one pass (one inner loop) of bubble sort has run.**

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

First Pass:

( 5 1 4 2 8 ) \to ( 1 5 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.

( 1 5 4 2 8 ) \to ( 1 4 5 2 8 ), Swap since 5 > 4

( 1 4 5 2 8 ) \to ( 1 4 2 5 8 ), Swap since 5 > 2

( 1 4 2 5 8 ) \to ( 1 4 2 5 8 ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

( 1 4 2 5 8 ) \to ( 1 4 2 5 8 )

( 1 4 2 5 8 ) \to ( 1 2 4 5 8 ), Swap since 4 > 2

( 1 2 4 5 8 ) \to ( 1 2 4 5 8 )

( 1 2 4 5 8 ) \to ( 1 2 4 5 8 )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

( 1 2 4 5 8 ) \to ( 1 2 4 5 8 )

( 1 2 4 5 8 ) \to ( 1 2 4 5 8 )

( 1 2 4 5 8 ) \to ( 1 2 4 5 8 )

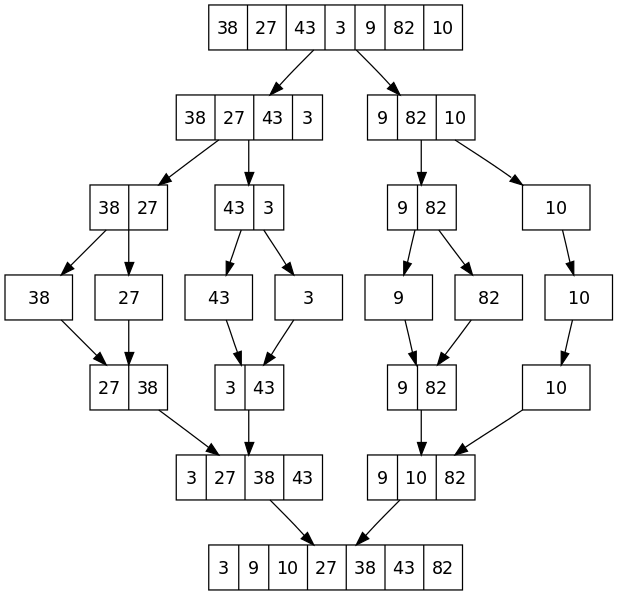
( 1 2 4 5 8 ) \to ( 1 2 4 5 8 )

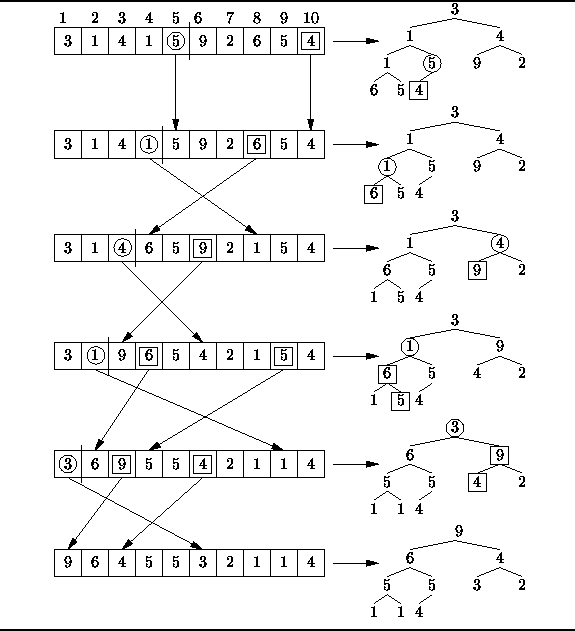
compare the first two elements of an array and swap them if they are out of order

do this until we reach the end. After one loop, I think it would look like 36883173919

**Show me the array after one partition step of quicksort has run. Which subarrays will the code sort next?**

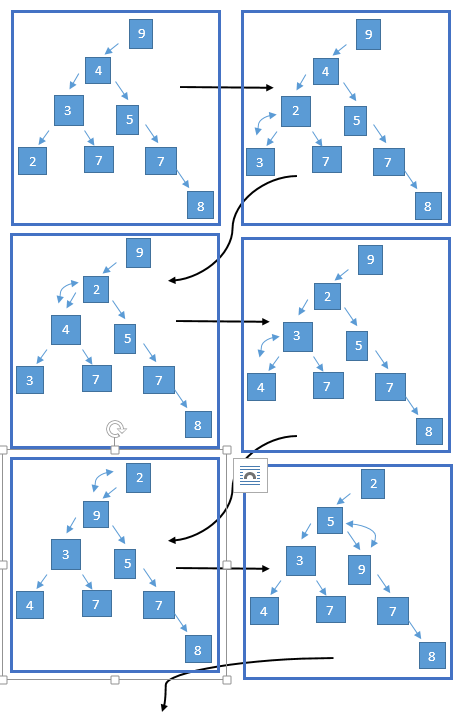
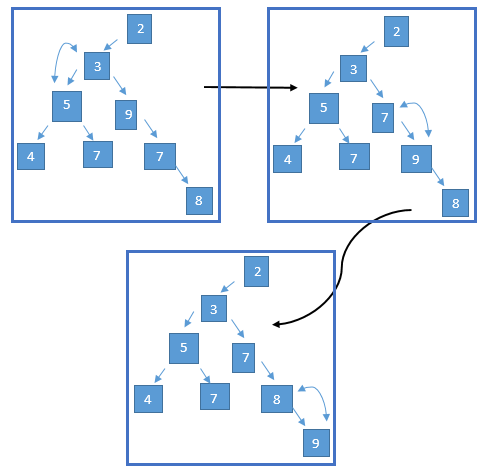
3 6 9 8 8 3 1 7 3 9 1.

**Show me how mergesort will process this array to sort it.**

In [computer science](http://en.wikipedia.org/wiki/Computer_science), merge sort (also commonly spelled mergesort) is an [*O*](http://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*) [comparison-based](http://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm). Most implementations produce a [stable sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Stability), which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a [divide and conquer algorithm](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm) that was invented by [John von Neumann](http://en.wikipedia.org/wiki/John_von_Neumann) in 1945.[[1]](http://en.wikipedia.org/wiki/Merge_sort#cite_note-1) A detailed description and analysis of bottom-up mergesort appeared in a report by [Goldstine](http://en.wikipedia.org/wiki/Herman_Goldstine) and Neumann as early as 1948.[[2]](http://en.wikipedia.org/wiki/Merge_sort#cite_note-2)

**16. Draw me the heap you get from adding the numbers 9, 4, 5, 3, 2, 7, 8, 7 to an empty heap.**

**Think about this like building a tree. we build the tree from the numbers given as you can see. the numbers above the array corespond to the i value. for example at the begining to find the parent of the 9 you divide the i value by 2 and then truncate.6/2=3. at i = 3 the number is 4 which is the parent.**

These pictures refer to question 17. I think this is how the heap would look if we created a min heap. not totally sure because the next question asks about removing the number 9 which makes me think it might be a max heap where the largest number is at the top. Regardless, this is how we would put an array into a heap and it works the same way if we do a max heap (largest value at the top) we would just swap the smallers with the largers instead of the other way around.

**18. Draw me the heap you get after removing the 9 from the heap in the previous question.**

If 9 is the root, we would just delete that node and make the next largest value the root(since it only would be pointing to 1 node, or swap the next largest value into its place (think about if it was pointing to 2 nodes instead of 1). If it was at the end it would be very easy because all we would have to do is set the pointer pointing to it to null

**19. Show me the code that checks for self assignment in operator =, and tell me what it’s checking.**

**Where else in big\_number might we need to check for this?**

if (this == &m) return \*this;

its checking if the location of this (the pointer to the current location) is equal to the location of m. if so then we will only need to return this. You need to check for self assignment anytime you are assigning a pointer to a value that could be equal to itself. For example if you are pointing to a location in memory you cannot point to itself

**20. What does partition do to make the array ‘less unsorted’ than before?**

At the end of one partition anything less than goes to the left greater than goes to the right of the pivot. Sorts it

**21. Given an array representation of a heap, tell me a formula to get the parent of a heap item in the array. Tell me a formula to get an array heap item’s left child, and its right child.**

iParent = floor((i / 2)  
 iLeftChild = 2\*i   
 iRightChild = 2\*i + 1

**22. Given the answers to the question above, write me a loop that re-heapifies a heap after one new element has been added.**

While (child > start)

Parent = floor((child-1)/2);

If (arr[parent] < arr[child])

Swap (arr[parent], arr[child]);

Child = parent;

Else

Return;

**23. Given the answers to the question above, write me a loop that re-heapifies a heap after the root element has been added.**

**oid min\_heapify(std::vector<int> &A, int i, int heapsize)**

{

int smallest;

int l = left(i);

//std::cout << "left = " << l << std::endl;

int r = right(i);

//std::cout << "right = " << r << std::endl;

if(l <= heapsize && A[l] < A[i])

smallest = l;

else

smallest = i;

//std::cout << "smallest = " << smallest << std::endl;

if(r <= heapsize && A[r] < A[smallest])

smallest = r;

if(smallest != i) {

print(A);

exchange(A, i, smallest);

min\_heapify(A, smallest, heapsize);

}

}

**24. What is particularly good to eat around Boulder? Delight the tastebuds of Dr. White.**

mcdonalds, 1125 10th street ←- why would you post your address?