LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
-LIGO-
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Technical Note **LIGO-T-030168- 00- D** August 13, 2003

Efficient Algorithm for computing a Running Median

Soumya D. Mohanty
Max Planck Institut für Gravitationsphysik
Am Mühlenberg 1, Golm D14476, Germany

This is an internal working note
of the LIGO Project.

California Institute of Technology
LIGO Project - MS 18-34
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS NW17-161
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-4824
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

Contents

I. OUTLINE

The *median* ν of a sample $z[i]$, $i = 1, \ldots, n$ is defined as follows. Let $Z[i]$ be the sequence obtained by sorting $z[i]$ in ascending order, $Z[1] < Z[2] < \ldots Z[n]$. (The ordering of equal elements is immaterial.) Then,

$$\nu = \text{median}(\{z[1], z[2], \ldots, z[n]\}) = \begin{cases} Z[\frac{n+1}{2}] & n \text{ odd} \\ \frac{Z[n/2]+Z[n/2+1]}{2} & n \text{ even} \end{cases} . \tag{1}$$

The median is a better estimator of the typical value of a sample than the mean when there are large extraneous outliers in the sample. For more details, see [1].

Let $x[k]$, $k = 0, 1, 2, \ldots, N-1$ be a sequence such as a time series or a Power Spectral Density (PSD). Then the *running median* is defined as the sequence $y[k] = \text{median}(\{x[k], x[k+1], , x[k+M-1]\})$, $k = 0, 1, , N-M$. The running median estimates the trend of the sequence $x[k]$ more faithfully than a running average when there are outliers in the data such as a short transient in a time series or a narrowband line feature in a power spectrum. This has led to applications of the running median in transient resistant trend estimation in the time domain [2, 3], PSD floor estimation [3, 4] and automated line detection [5].

Computation of the running median is very expensive if done in a brute force way, i.e., by sorting each block of M samples. Sorting is, in the worst case, an $O(n^2)$ operation. But if one utilises the fact that the computation of $y[k]$ has already sorted most of the samples required for computing $y[k+1]$, then the computation can be made much more efficient. This note describes one such algorithm and presents its pseudo code (the implementation programming language must allow *pointers* [7]). In contrast to the worst case floating point operations count (flops) of $\sim NM^2$ for the brute force algorithm, the algorithm presented here involves $\sim N\sqrt{M}$ flops in the worst case. We have also included a very detailed documentation of the C code which implements the algorithm presented below in appendices A and B. The C code has been used in [2–4]. The documentation has been generated using DOXYGEN [8].

II. ALGORITHM AND PSEUDO-CODE

Inputs to the code:

- X : the sequence $x[k]$, $k = 0, \ldots, N-1$.
- M : The number of points per block.

Output of the code:

- Y : sequence $y[k]$, $k = 0, \ldots, N-M$.

1. Sort the first M samples $x[k]$, $k = 0, \ldots, M-1$, in ascending order. The ANSI C <stdlib.h> library comes with a routine for sorting called *qsort* which can be used for this first step. Let the sorted list, in ascending order, be $Z[k]$, $k = 0, \ldots, M-1$. Thus, $Z[0] \le Z[1] \le Z[2] \ldots \le Z[M-1]$.

2. Load the sorted samples into the nodes of a *linked list* [6] with each node containing one sample. Each node of the linked list has three types of links to other nodes.

   Sequential link If the current node has sample $x[p]$, then this link points to the node containing $x[p+1]$.

   Next Sorted link If the current node has sample $Z[p]$, then this link points to the node containing $Z[p+1]$.

   Previous Sorted link If the current node has sample $Z[p]$, then this link points to the node containing $Z[p-1]$.

   In the *qsort* algorithm, the ordering of equal samples is arbitrary and so is it assumed here.

3. Set up an array, **checks**[ ], of pointers to nodes of the linked list such that **checks**[n] points to the node containing $Z\left[n * \text{floor}\left(\sqrt{M}\right)\right]$. The special nodes pointed to by elements of **checks**[ ] are called *checkpoints* in the following. Further denote the samples contained in checkpoint p by $C[p] = Z\left[p * \text{floor}\left(\sqrt{M}\right)\right]$, $p = 0, \ldots, \text{floor}(\sqrt{M}) - 1$.

   Why are the checkpoints spaced $\sqrt{M}$ samples apart? Once one obtains a sorted block of M samples, the *sequentially next* sample outside this block must be placed in the sorted list and the *sequentially first* sample in the block must be deleted. This is done by first comparing the new sample sequentially against the samples

in checkpoints. Once the checkpoints that bracket the new sample are found, comparisons are made with only the samples within this bracket to locate the exact position of the new sample in the sorted list. This implies that in the worst case P comparison operations are needed, if P is the number of checkpoints used, to find the bracketing checkpoints. After this one may have to make M/P further comparisons to locate the exact position of the new sample in the ordered list. Thus the total operation count K, in the worst case, is

$$K = P + M/P \; . \tag{2}$$

The value of P which minimises K is $P = \sqrt{M}$. This is why checkpoints are spaced $M/P = M/\sqrt{M} = \sqrt{M}$ samples apart in this algorithm.

4. Find the element n_0 of **checks**[ ] such that the sample in the corresponding node is nearest to the node containing the median (for M odd) or the first member of the pair which needs to be averaged (for M even). This element will provide a fast access to the node containing the samples needed to compute the median. This is necessary because we are dealing with a linked list and not an array that can be randomly accessed.

5. FOR $j = M$ TO $N - 1$ DO

   (a) Get sample $x[j]$.

   (b) Locate the element p_j of **checks** such that $C[p_j] \leq x[j] < C[p_j + 1]$.

   (c) Start from the node **checks**$[p_j]$ and follow the **Next Sorted** link until a node is found such that the sample value h it contains satisfies $x[j] \leq h$. We have thus found the exact place where the new sample $x[j]$ must be inserted.

   The next few steps find out the checkpoints that are bracketed by $x[j]$ and the sample to be deleted, $x[j - M]$. When the corresponding nodes are respectively inserted and deleted from the list, the bracketed checlpoints must be shifted to adjacent nodes.

   (d) Consider the node containing the first element of the *sequential* list. That sample will be $x[j - M]$.

   (e) Find elements q and p of **checks** such that

      i. If $x[j - M] < x[j]$
      $x(j - M) < C[q] < \ldots < C[p] < x[j]$

      ii. else if $x[j - M] > x[j]$
      $x[j] < C[p + 1] < \ldots < C[q] < x[j - M]$.

      iii. else if $x[j - M] == x[j]$
      Do nothing. No shifting of checkpoints required since in this case all samples in the sorted list between $x[j]$ and $x[j - M]$ must be equal to $x[j] = x[j - M]$.

   (f) Shift the checkpoints found in step 5e.

      i. If $x[j - M] < x[j]$
      Shift each pointer **checks**$[k]$, $k \in [q, p]$, to point to the next right node (i.e., higher in sorted order).

      ii. Else
      shift to the next left node.

   (g) Delete node containing $x[j - M]$ from the linked list (repair the links between the nodes adjacent to it).

   (h) Rewrite the data in this node by $x[j]$.

   (i) Insert this node before the node containing h.

   (j) Follow **Next Sorted** link from **checks**$[n_0]$ to get the median value.

6. END DO

Several special cases may arise such as $x[j]$ smaller or larger than any of the sample from the previous block. A lot of the code is devoted to handling such special cases. However, for the sake of clarity, these special cases are not shown in the outline above. Interested readers must consult the documentation of the C code (c.f., Appendix B) to understand how these special cases are handled.

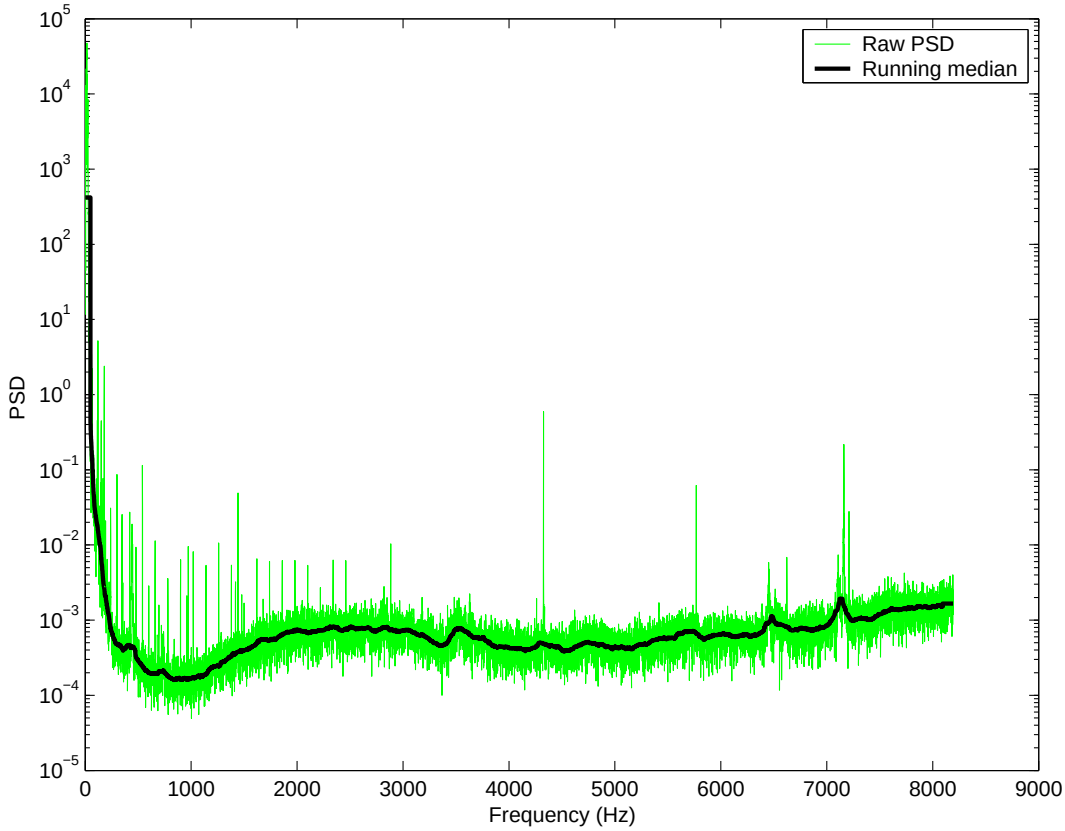

FIG. 1:

III. APPLICATION EXAMPLE

We show an example of noise floor estimation using running median in Fig. 1. The Power Spectral Density (PSD) of H1:LSC-AS_Q for an arbitrary epoch during the LIGO S2 run is plotted along with a running median of the PSD using a blocksize that corresponds to a bandwidth of 100 Hz. The frequency resolution of the raw PSD is 0.5 Hz. In contrast to the running median estimate, a running average with the same blocksize will get significantly distorted because of the presence of the line features (i.e., outliers) in the PSD. Fig. 2 illustrates the difference in performance of the running median and running average estimate. The running median noise floor estimate can be used directly in Fourier domain analyses [4] or in constructing a time domain whitening filter [3].

---

[1] A. Stuart, J. K. Ord, *Kendall's advanced theory of Statistics*, Vol. 1 (Edward Arnold, 1994).
[2] S. D. Mohanty, *Class. Quantum Grav.*, **19** (2002).
[3] S. Mukherjee, *Class. Quantum Grav.*, In press (2003).
[4] http://www.lsc-group.phys.uwm.edu/pulgroup/, Pulgroup S2 investigations page.
[5] Y. Itoh *et al*, Ligo internal note LIGO-T030175-00-0-Z (2003); GEO600 technical note (2002).
[6] Y. Langsam *et al*, *Data Structures using C and C++*, (Prentice Hall, 1996).
[7] R. Kumar, R. Agrawal, *Programming in ANSI C*, (West Publishing Company, 1992).
[8] http://www.doxygen.org

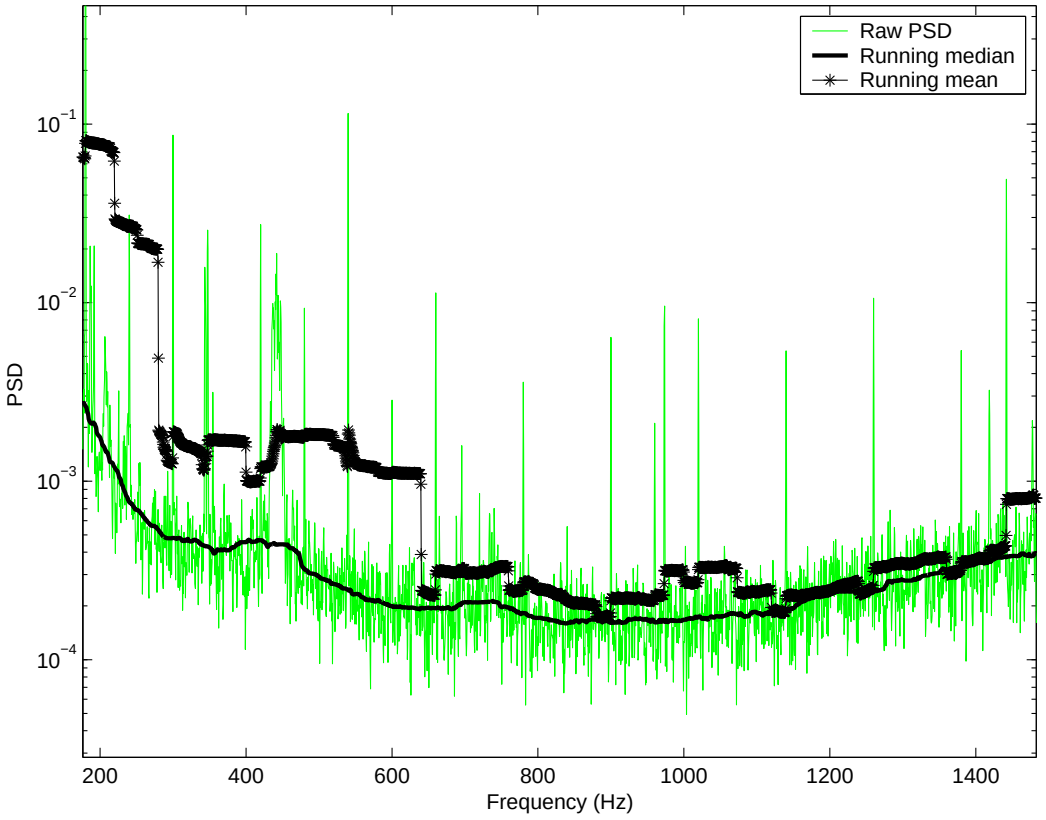

FIG. 2:

APPENDIX A: RUNNING MEDIAN C CODE

1. Function Documentation

*a. void rngmed (const double * data, unsigned int lendata, unsigned int nblocks, double * medians)*

Computes running median in an efficient manner.
Parameters:

data Pointer to input data array

lendata Length of input data array

nblocks Block size for computing running median

medians Pointer to output array. Number of elements is lendata - nblocks+1. Must be allocated outside this function.

*b. int rngmed_sortindex (const void * elem1, const void * elem2)*

This function is passed to the *qsort* function defined in *<stdlib.h>*. It is used internally by the *rngmed* function.
Parameters:

elem1 element of a **rngmed_val_index** (p. 7) array

elem2 another element of same **rngmed_val_index** (p. 7) array

2. Structure Documentation

a. struct node

This structure is used to make a linked list. The list holds the samples in one block in the running median algorithm.
Parameters:

data Holds a single number.

next_sorted Points to the next node in the sorted list.

prev_sorted Points to the previous node in the sorted list.

next_sequence point to the next node in the sequential list.

b. struct rngmed_val_index

A structure to store values and indices of elements in an array
Parameters:

data Stores a single number

index Stores the original position of the number

This structure is used to track the indices of elements after sorting by qsort. An array of rngmed_val_index is passed to qsort which rearranges the array according to the values in the data member. The indices of these elements in the original unsorted array can then be read off from the index member.

APPENDIX B: THE *RNGMED* FUNCTION BODY

Here we provide documentation of the *rngmed* function body.

1. Variables

```
double *sorted_indices;
struct rngmed_val_index *index_block;
struct node **checks,**node_addresses;
struct node *first_sequence,*last_sequence;
struct node *currentnode,*previousnode;
struct node *leftnode, *rightnode;
struct node *reuse_next_sorted,*reuse_prev_sorted;
struct node *dummy_node,*dummy_node1,*dummy_node2;
int ncheckpts,stepchkpts;
int nextchkptindx,*checks4shift;
int nearestchk,midpoint,offset,numberoffsets;
int samplecount,k,counter_chkpt,chkcount,shiftcounter;
double nextsample,deletesample,dummy,*dummy_array;
int shift,dummy_int;
```

2. Sort the first block

Allocate storage for an array of **rngmed_val_index**.

```
index_block =(struct rngmed_val_index *)calloc(nblocks, sizeof(struct rngmed_val_index));
```

Store the samples in the **data** member of each array element. Store the index of the sample in the **index** member of each array element.

```
for(k=0;k<nblocks;k++){
     index_block[k].data=data[k];
     index_block[k].index=k;
}
```

Pass the array to *qsort* along with pointer to function *rngmed_sortindex*.

```
qsort(index_block, nblocks, sizeof(struct rngmed_val_index),rngmed_sortindex);
```

Get the original indices of the samples in the sorted list. This list of indices is used at the start of the core part of the code.

```
sorted_indices=(double *)calloc(nblocks,sizeof(double));
for(k=0;k<nblocks;k++){
    sorted_indices[k]=index_block[k].index;
}
```

3. Set up checkpoints

Checkpoints are special nodes in the linked list containing the block of **nblocks** samples. A new sample is first compared against the values stored in these special nodes.

Get the number of nodes between consecutive checkpoints.

```
stepchkpts=sqrt(nblocks);
```

Get the number of checkpoints to use.

```
ncheckpts=nblocks/stepchkpts;
```

Allocate array to hold the pointers to checkpoints.

```
checks=(struct node **)calloc(ncheckpts,sizeof(struct node*));
```

The insertion of a new sample and the deletion of an old sample from the linked list containing the block of **nblocks** samples leads to the shifting of the checkpoints. The indices of elements in **checks** that need to be changed is stored in **checks4shift**.

```
if(!(checks4shift=(int*)calloc(ncheckpts,sizeof(int)))){
      printf("Could not allocate storage for checks4shift\n");
      return;
}
```

4. Get the nearest checkpoint

Get the nearest checkpoint to the node(s) containing the median of the block of **nblocks** samples. For *odd* **nblocks**, the median is the datum in the node that lies at the **midpoint** of the sorted list. For *even* **nblocks**, the median is the average of the data in the two nodes at the middle. The two cases are distinguished by the flag **numberoffsets**.

```
if((int)fmod(nblocks,2.0)){
      midpoint=(nblocks+1)/2-1;
      numberoffsets=1;
}
else{
      midpoint=nblocks/2-1;
      numberoffsets=2;
}
```

Get the nearest checkpoint to the median. This is used for fast access to the node(s) containing the median value. The usual method of access in an elementary linked list is *sequential*, which is slower.

```
nearestchk=floor(midpoint/stepchkpts);
offset=midpoint-nearestchk*stepchkpts;
```

5. Set up the linked list

The linked list containing **nblocks** samples is set up and initialized. This list has three types of links from one node to another. See the documentation for **struct node**. First, the sequential ordering is between nodes is set up. But the addresses of the nodes are needed to set up later the links representing the sorted order. The linked list has bidirectional links going in both the *ascending* and *descending* order.

Allocate storage to hold the node addresses.

```
node_addresses=(struct node **)calloc(nblocks,sizeof(struct node *));
```

Create a node. This stores the sequentially first sample in the block of **nblocks** samples.

```
first_sequence=(struct node *)calloc(1,sizeof(struct node));
```

Store its address.

```
node_addresses[0]=first_sequence;
```

Initialize this node.

```
first_sequence->next_sequence=NULL;
first_sequence->next_sorted=NULL;
first_sequence->prev_sorted=NULL;
first_sequence->data=data[0];
```

Start loop to setup links between nodes and load the sample values. Only the links representing the *sequential* order are setup. Links representing the sorted order are setup later.

BEGIN FOR LOOP.

```
previousnode=first_sequence;
for(samplecount=1;samplecount<nblocks;samplecount++){
      currentnode=(struct node *)calloc(1,sizeof(struct node));
      if(!currentnode){
           printf("Could not create node ");
           return;
      }
```

Store the address of the node.

```
      node_addresses[samplecount]=currentnode;
```

Link from already allocated node **previousnode** to the new one, **currentnode**. Load current data sample into **currentnode**.

```
      previousnode->next_sequence=currentnode;
      currentnode->next_sequence=NULL;
      currentnode->prev_sorted=NULL;
      currentnode->next_sorted=NULL;
      currentnode->data=data[samplecount];
```

Set **currentnode** as **previousnode** for next iteration of the loop.

```
      previousnode=currentnode;
}
```

END FOR LOOP.

This stores the sequentially last sample of the block of **nblocks** samples.

```
last_sequence=currentnode;
```

6. Set up sorted list

The links between nodes representing sorted order are set up. For this the previously stored indices, after sorting using *qsort*, are used.

Node containing the lowest value.

```
currentnode=node_addresses[(int)sorted_indices[0]];
```

This is also the first checkpoint.

```
checks[0]=currentnode;
```

BEGIN FOR LOOP.

```
previousnode=NULL;
nextchkptindx=stepchkpts;
counter_chkpt=1;
for(samplecount=1;samplecount<nblocks;samplecount++){
```

Get the address of the node containing the next highest value.

```
    dummy_node=node_addresses[(int)sorted_indices[samplecount]];
```

Make a link from **current_node** to this address. A second link is made to the previous lower value also. Thus, the linked list has bidirectional links going in both the ascending and descending order.

```
    currentnode->next_sorted=dummy_node;
    currentnode->prev_sorted=previousnode;
    previousnode=currentnode;
    currentnode=dummy_node;
```

If the node is also a checkpoint then record its address in the **checks** array.

```
    if(samplecount==nextchkptindx && counter_chkpt<ncheckpts){
          checks[counter_chkpt]=currentnode;
          nextchkptindx+=stepchkpts;
          counter_chkpt++;
    }
}
```

END FOR LOOP.

Set up links for the last node in sorted order.

```
currentnode->prev_sorted=previousnode;
currentnode->next_sorted=NULL;
```

7. The core part

This is the core engine of the code.

The output is stored in the array **medians** which should be allocated outside the code.

First, get the median of the first block of samples. Go to the nearest checkpoint.

```
currentnode=checks[nearestchk];
```

Follow link from this node to the node containing the median.

```
for(k=1;k<=offset;k++){
      currentnode=currentnode->next_sorted;
}
```

Depending on odd or even **nblocks**, get the value from the **data** member of **currentnode** or calculate the average of this value and the value stored in the nest node.

```
dummy=0;
for(k=1;k<=numberoffsets;k++){
      dummy+=currentnode->data;
      currentnode=currentnode->next_sorted;
}
medians[0]=dummy/numberoffsets;
```

Now move to calculation of medians for successive blocks.

BEGIN FOR LOOP.

```
for(samplecount=nblocks;samplecount<lendata;samplecount++){
     nextsample=data[samplecount];
```

a. Locate point of insertion

First the point where the nextsample must be inserted in the linked list constructed above must be located.

Compare the new sample, **nextsample**, with checkpoints. There are two cases to be considered.

a. Case 1

```
if(nextsample>checks[0]->data){
```

Find a checkpoint that is greater than the new sample.

```
for(chkcount=1;chkcount<ncheckpts;chkcount++){
        if(nextsample>checks[chkcount]->data){
        }
        else{
             break;
        }
}
```

Back up to previous checkpoint.

```
chkcount-=1;
```

rightnode is the node that lies immediately to the right of the new sample in ascending order and **leftnode** is the node immediately on the left.

Follow the link in ascending order, starting from the checkpoint to the left of **nextsample**, until the bracketing nodes are found.

```
rightnode=checks[chkcount];
while(rightnode){
        if(nextsample<=rightnode->data){
              break;
        }
        leftnode=rightnode;
        rightnode=rightnode->next_sorted;
}
```

The new sample must be inserted as a node between **leftnode** and **rightnode**.

The node containing the sequentially first sample of the block, **first_sequence**, must be removed from the list and the same node is reused to store the new sample. This node then becomes the sequentially last, i.e., **last_sequence**.

Special care is needed if the node to be removed also happens to be either **rightnode** or **leftnode**. In this case, the checkpoints need not be shifted (**shift** = 0).

```
if(rightnode==first_sequence){
        rightnode->data=nextsample;
        first_sequence=first_sequence->next_sequence;
        rightnode->next_sequence=NULL;
        last_sequence->next_sequence=rightnode;
        last_sequence=rightnode;
        shift=0;
}
else{
        if(leftnode==first_sequence){
               leftnode->data=nextsample;
```

```
                        first_sequence=first_sequence->next_sequence;
                        leftnode->next_sequence=NULL;
                        last_sequence->next_sequence=leftnode;
                        last_sequence=leftnode;
                        shift=0;
                }
```

Otherwise the checkpoints may need to be shifted (**shift** = 1).

```
                else {
                        reuse_next_sorted=rightnode;
                        reuse_prev_sorted=leftnode;
                        shift=1;
                }
        }
```

The nodes to the right and left of the node that will be recycled and inserted are **rightnode** and **leftnode**. So, store the addresses of these nodes in **reuse_next(prev)_sorted**. The links between **rightnode** and **leftnode** must be broken and pointed to the new node. Conversely the new node must form links to **rightnode** and **leftnode**.

b. Case 2 This is the case where **nextsample** $\leq$ **checks[0]** $\rightarrow$ **data**. Recall that the first checkpoint **checks[0]** also holds the lowest value in sorted order. But this need not be the sequentially first value. Hence, distinguish the two cases below.

```
else{
        chkcount=0;
        dummy_node=checks[0];
        if(dummy_node==first_sequence){
              dummy_node->data=nextsample;
              first_sequence=first_sequence->next_sequence;
              dummy_node->next_sequence=NULL;
              last_sequence->next_sequence=dummy_node;
              last_sequence=dummy_node;
              shift=0;
        }
        else{
              reuse_next_sorted=checks[0];
              reuse_prev_sorted=NULL;
              shift=1;
        }
        rightnode=checks[0];
        leftnode=NULL;
}
```

b. Find checkpoints to shift

If the sequentially first sample is not immediately adjacent to **nextsample** then deleting the node containing the first sample and reinserting the node elsewhere in the list (as located in the above code) requires that the intermediate checkpoints be shifted by one node.

```
if(shift){
        deletesample=first_sequence->data;
```

The direction of the shift depends on whether the sequentially first sample, **deletesample**, is less than, greater than or equal to **nextsample**.

a. Sequentially first greater than sequentially last The checkpoints must be shifted to point to the nodes with immediately lower values in the ordered list.

```
        if(deletesample>nextsample){
             shiftcounter=0;
             for(k=chkcount;k<ncheckpts;k++){
                   dummy_node=checks[k];
                   dummy=dummy_node->data;
                   if(dummy>=nextsample){
                         if(dummy<=deletesample){
```

This checkpoint falls between **deletesample** and **nextsample**. So, it must be shifted. Store its index in **checks4shift**.

```
                        checks4shift[shiftcounter]=k;
                        shiftcounter++;
                    }
                    else{
                        break;
                    }
                }
            }
            shift=-1;
        }
```

b. Sequentially first less than sequentially last The checkpoints must be shifted to point to the nodes with immediately higher values in the ordered list.

```
        else
            if(deletesample<nextsample){
                shiftcounter=0;
                for(k=chkcount;k>=0;k--){
                    dummy_node=checks[k];
                    dummy=dummy_node->data;
```

This checkpoint falls between **deletesample** and **nextsample**. So, it must be shifted. Store its index in **checks4shift**.

```
                    if(dummy>=deletesample){
                        checks4shift[shiftcounter]=k;
                        shiftcounter++;
                    }
                    else{
                        break;
                    }
                }
                shift=1;
            }
```

c. Sequentially first equal to sequentially last The sequentially first and last samples are equal but they are separated in the ordered list. This implies that all values in between in the ordered list must be equal. So, only the sequential list links need be changed.

```
            else{
                dummy_node=first_sequence;
                dummy_node->data=nextsample;
                last_sequence->next_sequence=dummy_node;
                first_sequence=dummy_node->next_sequence;
                dummy_node->next_sequence=0;
                last_sequence=dummy_node;
                shift=0;
            }
    }
```

c. Implementing the link changes

Now the node containing the sequentially first sample, **first_sequence**, must be recycled. This means its old links (both sequential and ordered) must be severed. The nodes linking into **first_sequence** must be relinked. The links between the nodes immediately adjacent to the insertion point must be broken and relinked to the recycled **first_sequence** node.

```
    if(shift){
```

Some cases did not require any shift and in those cases **first_sequence** has already been recycled. Do the following only if **shift** $\neq 0$.

First reset the sequential links and load **nextsample** into **first_sequence**. The **first_sequence** node becomes **dummy_node**.

```
dummy_node=first_sequence;
first_sequence=dummy_node->next_sequence;
dummy_node->next_sequence=NULL;
last_sequence->next_sequence=dummy_node;
last_sequence=dummy_node;
dummy_node->data=nextsample;
```

Store the sorted order links.

```
dummy_node1=dummy_node->prev_sorted;
dummy_node2=dummy_node->next_sorted;
```

a. Repair the deletion point The nodes adjacent to **first_sequence** in the sorted list are relinked. Two cases must be considered.

Case 1 If **first_sequence** was also the first checkpoint then its **prev_sorted** link (=**dummy_node1** above) must be NULL. This may be a redundant check since this case has been addressed earlier in the code (**shift** = 0). But there is no harm in repeating it.

```
if(!dummy_node1){
       dummy_node2->prev_sorted=dummy_node1;
}
```

Case 2 If **first_sequence** was the last node in the ascending order list then **next_sorted** must be NULL.

```
else {
       if(!dummy_node2){
              dummy_node1->next_sorted=dummy_node2;
       }
```

Case 3 The normal case where there are nodes on either side.

```
       else{
              dummy_node1->next_sorted=dummy_node2;
              dummy_node2->prev_sorted=dummy_node1;
       }
}
```

b. Relink nodes at insertion point Recall that the adjacent nodes were **rightnode** and **leftnode**. Link them into the recycled node.

If the insertion point is at the end of the ascending order sorted list, **rightnode** must be NULL.

```
if(!rightnode){
       leftnode->next_sorted=dummy_node;
 }
```

If the insertion point is at the beginning of the ascending order sorted list, **leftnode** must be NULL.

```
else {
       if(!leftnode){
              rightnode->prev_sorted=dummy_node;
       }
```

Else both **leftnode** and **rightnode** exist.

```
       else{
              leftnode->next_sorted=dummy_node;
              rightnode->prev_sorted=dummy_node;
       }
}
```

d. Implementing checkpoint shifts

Two cases here: shift to left or right.

```
if(shift==-1){
       for(k=0;k<shiftcounter;k++){
              dummy_int=checks4shift[k];
              checks[dummy_int]=checks[dummy_int]->prev_sorted;
       }
}
else
       if(shift==1){
              for(k=0;k<shiftcounter;k++){
                     dummy_int=checks4shift[k];
                     checks[dummy_int]=checks[dummy_int]->next_sorted;
              }
       }
```

Link the recycled node to **leftnode** and **rightnode**.

```
     dummy_node->next_sorted=reuse_next_sorted;
     dummy_node->prev_sorted=reuse_prev_sorted;
}
```

8. Get the median

Go to the nearest check point.

```
currentnode=checks[nearestchk];
```

Follow link through **offset** number of nodes.

```
for(k=1;k<=offset;k++){
     currentnode=currentnode->next_sorted;
}
dummy=0;
```

Even and odd **nblocks** cases.

```
for(k=1;k<=numberoffsets;k++){
     dummy+=currentnode->data;
     currentnode=currentnode->next_sorted;
}
medians[samplecount-nblocks+1]=dummy/numberoffsets;
```

END FOR LOOP.

```
}
```

9. Clean up

Deallocate memory that was dynamically allocated. Not shown in this document are some sections of the code where additional cleaning up is done during the course of execution.

```
free(node_addresses);
currentnode=first_sequence;
while(currentnode){
        previousnode=currentnode;
        currentnode=currentnode->next_sequence;
        free(previousnode);
}
free(checks);
```