Content

1 Heap

2 Hash

3 Sorting

4 Tree

1. Heap

* 1. **Keep the heap property(assuming it is all max heap)**

Function: heapfy()

1. **Algorithm:**

1. If left > root, swap(left, root), or if right > root, swap(right, root)
2. Recursively call heapfy() on the swap the child node till end so the heap property is reserved.
3. **Code**

// n size of array, i is start index.

Void heapfy(int[] arr, int n, int i)

{

int largest = i;

int l = 2i+1; // binary tree index relations

int r = 2i+2;

if(l<n && arr[l] > arr[largest])

largest = i;

if(r<n && arr[r]> arr[largest])

largest = r;

//swap

If(largest != i)

{

swap(arr[i], arr[largest]);

heapfy(arr, n, largest);

}

}

1. **Time complexity**

h is the height of the tree

O(h) = O(log\_2 n)

* 1. Build Heap

1. **Algorithm**

do heapfy() on all the nodes that have children, these nodes have index(n/2-1, 0) (based on binary tree property)

1. **Code**

void buildHeap(int[] arr, int n)

{

for(int i=n/2 -1; i>=0; i--)

{

heapfy(arr, n, i);

}

}

1. **Time Complexity**

Time = \sum \_{h=0} ^ {log\_2n} [\frac{n}{2^{h+1}}] O(h)

O(h) : time to call one heapfy()

[\frac{n}{2^{h+1}}]: number of nodes on the same tree level

\sum \_{h=0} ^ {log\_2^n} : sum over all the levels.

So time = O(n \sum\_{h=0}^(lgn) h/(2^h))

= O(2n)

=O(n)

**1.3 Heapsort**

1. **Algorithm**
2. Build the heap
3. Start from the last element(index n-1),
4. swap it with the first element. Then the last element becomes sorted
5. Do heapfy on the unsorted elements
6. Move to the next unsorted element, in this case it is the last but two(index n-2), repeat 3) and 4) until all elements become sorted.
7. **Code**

Void heapsort(int arr[], int n)

// build heap

void buildHeap(arr, n);

for(int i = n-1; i>=0; i--)

{

swap(arr[0], arr[i]);

heapfy(arr, i, 0);

}

1. **Time complexity**
2. Build heap O(n)
3. Sort nlog\_2 n
4. **Comparison**
5. Compare to quicksort

|  |  |  |  |
| --- | --- | --- | --- |
|  | QS | HS | comment |
| Average Speed | Faster |  | 12nlogn vs 16 nlog n |
| Worst Speed |  | Faster | n^2 vs nlogn |
|  |  |  |  |

1. Compare to mergesort

|  |  |  |  |
| --- | --- | --- | --- |
|  | MS | HS | comment |
| Space | O(n) | O(1) | MS needs additional space for merge |
| Stable **s**ort | Yes |  | MS keeps order for same element |
| Better Cache performance | Yes |  | MS accesses the caches that are near to each other. |
|  |  |  |  |

* 1. Binary heap time complexity analysis

1. **Time complexity table**

|  |  |  |
| --- | --- | --- |
|  | Average | Worst |
| Search | O(n) | O(n) |
| Insert | O(1) | O(h) = O(long(n)) |
| Delete | O(log(n)) | O(long(n)) |
| Peek | O(1) | O(1) |

1. **Insert**
2. Add the element to the bottom level of the heap
3. Compare the added element with its parent

If they are in correct order, stop

1. If not swap the element with its parent and return to the 2) by keeping comparing the parent
2. **Insert average time complexity**

Assuming a uniform distribution of numbers, which means for any element in the heap, it has a one-half chance of being greater than its parent. And it has one-fourth chance of being greater than its grandparent. So the expected number of swap during the insertion is

Probability of swapping with 1st parent \* number of swap + Probability of swapping with 2st parent \* number of swap + …+ Probability of swapping with mst parent

= ½ \* 1 + ¼ \* 2 + 1/8\* 3 + ½^m \* m

= \sum \frac{m}{2^m}

= 2 when m goes to \inf

Therefore the averaged time complexity is O(1)

1. Hash
2. **Hash functions**:

Suppose we need to have M number of key-value pairs

1. Keys are positive integers

Modular hashing

hashIndex = key % M;

1. Floating point numbers

Turn the floating number to binary representation and use it as the key, then do modular hashing as for positive integers

1. Turn string to positive integer code, for example:

String s;

int hashIndex = 0;

for(int i=0; i< s.length; i++)

{

hashIndex = (R\*hashIndex + s.charAt(i)) % M

}

(java uses R = 31)

1. Java conventions

Every data type must implement a method called hashCode,

The JVM provides a default one (this can be understood as using address)

Or users are able to override it.

Convert hashcode to array index,

Private int hashFunction(Key key)

{

Return (key.hashCode() & 0x7fffffff ) %M

}

This mask turns 32 bit Integer to 31 bit non negative integer

1. **Collision resolving using chaining**
2. Idea

For each cell in the hashtable, use a list to store <key, value> pair that has the same hash index

Hash Index list<key, value>

0

1 <15, v1> <8, v2>

2

3

4 <11, v3>

5

6 <27, v4>

1. Example in c++

class HashNode {

private:

int key;

int value;

HashNode \*next;

public:

// constructor

//HashNodeint key, int value)

// implement getters and setters

// getKey() getValue() setValue() getNext()setNext()

};

const int TABLE\_SIZE = 128;

class HashMap

{

private:

HashNode \*\*table;

Public:

HashMap() {

table = new HashNode\*[TABLE\_SIZE];

for (int i = 0; i < TABLE\_SIZE; i++)

table[i] = NULL;

}

int get(int key) {

int hash = (key % TABLE\_SIZE);

if (table[hash] == NULL)

return -1;

else {

HashNode \*entry = table[hash];

while (entry != NULL && entry->getKey() != key)

entry = entry->getNext();

return entry == NULL? -1 : entry->getValue();

}

}

void put(int key, int value) {

int hash = (key % TABLE\_SIZE);

if (table[hash] == NULL)

table[hash] = new HashNode(key, value);

else {

HashNode \*entry = table[hash];

while (entry->getNext() != NULL)

entry = entry->getNext();

if (entry->getKey() == key)

entry->setValue(value);

else

entry->setNext(new HashNode(key, value));

}

}

void remove(int key) {

int hash = (key % TABLE\_SIZE);

if (table[hash] != NULL) {

HashNode \*prevEntry = NULL;

HashNode \*entry = table[hash];

while (entry->getNext() != NULL && entry->getKey() != key) {

prevEntry = entry;

entry = entry->getNext();

}

if (entry->getKey() == key) {

HashNode \*nextEntry = entry->getNext();

if (prevEntry == NULL) {

table[hash] = nextEntry;

} else {

prevEntry->setNext(next);

}

delete entry;

}

}

}

};

1. **Collision resolving using open addressing**
2. Idea

Open addressing solves the hash collision as follows,

If collision, find the next empty cell

If full, increase the size of the table

1. Example

Here, to mark a node deleted we have used **DeletedNode** with key and value -1. This can differenciate if table cell is empty by default or it is empty due to deletion.  
Insert can insert an item in a deleted slot, but search doesn’t stop at a deleted slot.

class DeletedEntry: public HashEntry {

private:

      static DeletedEntry \*entry;

      DeletedEntry() :

            HashEntry(-1, -1) {

      }

public:

      static DeletedEntry \*getUniqueDeletedEntry() {

            if (entry == NULL)

                  entry = new DeletedEntry();

            return entry;

      }

};

DeletedEntry \*DeletedEntry::entry = NULL;

const int TABLE\_SIZE = 128;

class HashMap {

private:

      HashEntry \*\*table;

public:

      HashMap() {

            table = new HashEntry\*[TABLE\_SIZE];

            for (int i = 0; i < TABLE\_SIZE; i++)

                  table[i] = NULL;

      }

      int get(int key) {

            int hash = (key % TABLE\_SIZE);

            int initialHash = -1;

            while (hash != initialHash && (table[hash]

                        == DeletedEntry::getUniqueDeletedEntry() ||

table[hash] != NULL && table[hash]->getKey() != key)) {

                  if (initialHash == -1)

                        initialHash = hash;

                  hash = (hash + 1) % TABLE\_SIZE;

            }

            if (table[hash] == NULL || hash == initialHash)

                  return -1;

            else

                  return table[hash]->getValue();

      }

      void put(int key, int value) {

            int hash = (key % TABLE\_SIZE);

            int initialHash = -1;

            int indexOfDeletedEntry = -1;

            while (hash != initialHash && (table[hash]

                        == DeletedEntry::getUniqueDeletedEntry() ||

table[hash] != NULL  && table[hash]->getKey() != key)) {

                  if (initialHash == -1)

                        initialHash = hash;

                  if (table[hash] == DeletedEntry::getUniqueDeletedEntry())

                        indexOfDeletedEntry = hash;

                  hash = (hash + 1) % TABLE\_SIZE;

            }

            if ((table[hash] == NULL || hash == initialHash)

&& indexOfDeletedEntry!= -1)

                  table[indexOfDeletedEntry] = new HashEntry(key, value);

            else if (initialHash != hash)

                  if (table[hash] != DeletedEntry::getUniqueDeletedEntry()

                             && table[hash] != NULL && table[hash]->getKey() == key)

                        table[hash]->setValue(value);

                  else

                        table[hash] = new HashEntry(key, value);

      }

      void remove(int key) {

            int hash = (key % TABLE\_SIZE);

            int initialHash = -1;

            while (hash != initialHash && (table[hash]

                        == DeletedEntry::getUniqueDeletedEntry() ||

table[hash] != NULL && table[hash]->getKey() != key)) {

                  if (initialHash == -1)

                        initialHash = hash;

                  hash = (hash + 1) % TABLE\_SIZE;

            }

            if (hash != initialHash && table[hash] != NULL) {

                  delete table[hash];

                  table[hash] = DeletedEntry::getUniqueDeletedEntry();

            }

      }

};

**Open addressing vs. chaining**

|  |  |  |
| --- | --- | --- |
|  | **Chaining** | **Open addressing** |
| **Collision resolution** | Using external data structure | Using hash table itself |
| **Memory waste** | Pointer size overhead per entry (storing list heads in the table) | No overhead 1 |
| **Performance dependence on table's load factor** | Directly proportional | Proportional to (loadFactor) / (1 - loadFactor) |
| **Allow to store more items, than hash table size** | Yes | No. Moreover, it's recommended to keep table's load factor below 0.7 |
| **Hash function requirements** | Uniform distribution | Uniform distribution, should avoid clustering |
| **Handle removals** | Removals are ok | Removals clog the hash table with "DELETED" entries |
| **Implementation** | Simple | Correct implementation of open addressing based hash table is quite tricky |

1. **Sorting**
   1. **Soring Basic Concept**
2. **In place sorting**

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

1. **Internal and external sorting**

When all data that needs to be sorted cannot be placed in-memory at a time, the sorting is called [external sorting](http://en.wikipedia.org/wiki/External_sorting). External Sorting is used for massive amount of data. Merge Sort and its variations are typically used for external sorting. Some extrenal storage like hard-disk, CD, etc is used for external storage.  
When all data is placed in-memory, then sorting is called internal sorting.

1. **Stable sorting**

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

* 1. **Selection Sort**

1. **Algorithm**

The algorithm maintains two subarrays in a given array. One subarray is already sorted, the remaining subarray is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is **SELECTED** and moved to the sorted subarray.

1. **C++ implementation**

void selectionSort(int arr[], int n)

{

    int i, j, min\_idx;

// One by one move boundary of unsorted subarray

for (i = 0; i < n-1; i++)

    {

        // Find the minimum element in unsorted array

        min\_idx = i;

        for (j = i+1; j < n; j++)

        {  if (arr[j] < arr[min\_idx])

            min\_idx = j;

   }

        // Swap the found minimum element with the first element

        swap(&arr[min\_idx], &arr[i]);

    }

}

1. **Complexity**

**Time complexity:**

Average: O(n^2) comparisons, O(n) swaps

Best: O(n^2) comparisons, O(n) swaps

Worst: O(n^2) comparisons, O(n) swaps

**Space complexity:**

O(1)

The good thing about selection sort is it never makes more than O(n) swaps/write and can be useful when memory write is a costly operation.

* 1. **Insertion Sort**

1. **Algorithm**  
   // Sort an arr[] of size n  
   insertionSort(arr, n)  
   Loop from i = 1 to n-1.

Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

1. C++ implementation

void insertionSort(int arr[], int n)

{

    int i, key, j;

    for (i = 1; i < n; i++)

    {

        Key = arr[i];

        j = i-1;

        /\* Move elements of arr[0..i-1], that are

           greater than key, to one position ahead

           of their current position \*/

        while (j >= 0 && arr[j] > key)

        {

            arr[j+1] = arr[j];

            j = j-1;

        }

        arr[j+1] = key;

    }

}

1. **Complexity**

**Time complexity:**

Worst O(n^2) comparisons and swaps: Elements are reversely sorted.

Best O(n) comparisons and swaps: Elements are already sorted.

**Space complexity:**

O(1)

* 1. **Bubble sort**

1. **Algorithm**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**b. c++ implementation**

void bubbleSort(int arr[], int n)

{

int i, j;

bool swapped;

for (i = 0; i < n-1; i++)

{

swapped = false;

for (j = 0; j < n-i-1; j++)

{

if (arr[j] > arr[j+1])

{

swap(&arr[j], &arr[j+1]);

swapped = true;

}

}

// IF no two elements were swapped by inner loop, then break

if (swapped == false)

break;

}

}

* 1. **QuickSort**

1. **Algorithm**
2. Randomly choose a pivot element
3. partition

Given an array and pivot element, put x at its correct position in sorted array such that all elements smaller than x before x, and put all elements greater than x after x.

This is done by keep track of an index i which rightmost elements that are smaller than the pivot. For each jth element, if arr[j] < pivot

Then swap a[j] with a[i+1], i++

1. keep doing 1) and 2) in the partitioned array
2. implement in c++

int partition (int arr[], int low, int high)

{

    int pivot = arr[high];    // pivot

    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)

    {

        // If current element is smaller than or

        // equal to pivot

        if (arr[j] <= pivot)

        {

            i++;    // increment index of smaller element

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

void quickSort(int arr[], int low, int high)

{

     if (low < high)

     {

        int pi = partition(arr, low, high);

         quickSort(arr, low, pi - 1);

         quickSort(arr, pi + 1, high);

     }

}

1. **Time complexity**

Average case: nlogn

Worst case(with a sorted array and bad pivot): n^2, the algorithm goes to insertion sort

1. **Space complexity: log n**

Since it is a recursion call and it needs a stack frame to store function address. The length of the stack frame is log n

1. **Quick sort is in-place but not stable.**
2. **Comparison with heap sort**
3. worst time: heap sort is better as it guarantees n logn
4. average time: quicksort is better as the prefactor of n logn is smaller
5. **Comparison with merge sort**
6. Merge sort is a stable sort, quick sort is not.
7. Space. Merge sort requires O(n) space.
8. Example

Java.Arrays.sort(), the reason to use quick sort in this case is java array consists of primitive types only. For two primitive type variables, as long as the values are the same, they are the same. Just like identical particles in quantum mechanics. Therefore, we do not require stable sort for primitive types. So quick sort becomes a good choice.

* 1. **Merge Sort**

1. **Algorithm**

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

middle m = (l+r)/2

1. Call mergeSort for first half:

Call mergeSort(arr, l, m)

1. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

1. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

1. **Implementation in c++**

// Merges two subarrays of arr[]. First subarray is arr[l..m] Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1+ j];

/\* Merge the temp arrays back into arr[l..r]\*/

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = l; // Initial index of merged subarray

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

/\* Copy the remaining elements of L[], if there

are any \*/

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

/\* Copy the remaining elements of R[], if there

are any \*/

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

/\* l is for left index and r is right index of the

sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

// Same as (l+r)/2, but avoids overflow for

// large l and h

int m = l+(r-l)/2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

1. **Complexity:**

**Time complexity:**

Average: nlog(n)

Best: nlog(n)

Worst: nlog(n)

**Space complexity:**

Depending on the give data structure.

If it is an array then it needs n additional spaces, however if it is linkedlist, then it does not need additional space

1. **Not in-place but stable**
2. **Application**
3. **Sort linkedlist.**

Because merge two linkedlists can be done by inserting element from one list to the other, and inserting element in the linkedlist needs O(1) in space, and O(1) in time. Therefore merge sort is useful for sorting linked list.

1. **External sorting**
2. **Java collection sort**

The reason to choose merge sort is collection are made of objects. So the sorting has to be stable. And a natural assumption is when using objects, one does not care the usage of space too much. Therefore merge sort is optimal.

* 1. **External Merge Sorting for large amount of data example**

Since merge sort is a divide and conquer algorithm, it can be used to sort large amount of data. The algorithm first sorts *M* items at a time and puts the sorted lists back into external memory. It then [recursively](https://en.wikipedia.org/wiki/Recursion) does a M/B merge {\displaystyle {\tfrac {M}{B}}}Mdddddon those sorted lists. To do this merge, *B* elements from each sorted list are loaded into internal memory, and the minimum is repeatedly outputted.

Example, sort 900 [megabytes](https://en.wikipedia.org/wiki/Megabyte) of data using only 100 megabytes of RAM:

1. Read 100 MB of the data in main memory and sort by some conventional method, like [quicksort](https://en.wikipedia.org/wiki/Quicksort).
2. Write the sorted data to disk.
3. Repeat steps 1 and 2 until all of the data is in sorted 100 MB chunks (there are 900MB / 100MB = 9 chunks), which now need to be merged into one single output file.
4. Read the first 10 MB (= 100MB / (9 chunks + 1)) of each sorted chunk into input buffers in main memory and allocate the remaining 10 MB for an output buffer. (In practice, it might provide better performance to make the output buffer larger and the input buffers slightly smaller.)
5. Perform a [9-way merge](https://en.wikipedia.org/wiki/K-way_merging) and store the result in the output buffer. Whenever the output buffer fills, write it to the final sorted file and empty it. Whenever any of the 9 input buffers empties, fill it with the next 10 MB of its associated 100 MB sorted chunk until no more data from the chunk is available. This is the key step that makes external merge sort work externally -- because the merge algorithm only makes one pass sequentially through each of the chunks, each chunk does not have to be loaded completely; rather, sequential parts of the chunk can be loaded as needed.
6. **Tree**
   1. **Binary Tree**
7. **Properties**
8. **How many different Unlabeled Binary Trees can be there with n nodes?**

For n = 1, there is only one tree

o

For n = 2, there are two trees

o o

/ \

o o

For n = 3, there are five trees

o o o o o

/ \ / \ / \

o o o o o o

/ \ \ /

o o o o

We use T(n) to denote the number of trees with n nodes. We consider base case

T(0) = 0, because there is only one empty tree.

T(1) = 1, because if we have only one node, the tree is the node itself.

T(2) = 2, after we put the first node, which is also the root node, the rest one can be put either on the left or on the right. So there are two different trees.

Starting from n=3, we first place the root node, then the number of the rest nodes is n-1 =2. There are 3 possibilities,

Possibility number 1:

First, we can place two rest nodes on the left subtree, 0 nodes on the right subtree. The number of trees is the number of tree with 2 nodes times the number of tree with 0 nodes. In our notation, it is T(2)\*T(0).

Possibility number 2:

Second, we can place one node on the left subtree, the other on the right subtree. The number is T(1)\*T(1).

Possibility number 3:

First, we can place 0 nodes on the left subtree, 2 nodes on the right subtree. The number of trees is the number of tree with 0 nodes times the number of tree with 2 nodes. In our notation, it is T(0)\*T(2).

So the total number of trees is

T(2)\*T(0) + T(1)\*T(1) + T(0)\*T(2) = 2\*1 + 1\*1 +1\*2 = 5.

Similarly,

T(4) = T(0)\*T(3) + T(1)\*T(2) + T(2)\*T(1) + T(3)\*T(0)

= 1\*5 + 1\*2 + 2\*1 + 5\*1 = 14.

In general,

T(n) = \sum\_{i=0}^{n-1} T(i) T(n-i-1)

* 1. **Red and Black Tree**

1. **Def**

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.

1. Every node has a color either red or black.
2. Root of tree is always black.
3. There are no two adjacent red nodes
4. Every path from root to a NULL node has same number of black nodes
5. **Usage**

Most of the BST operations (e.g., search, max, min, insert, delete) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The self-balancing BST are implement as set and map in c++ and TreeSet and TreeMap in java.

1. Properties:
2. **A node of height h has black-height >= h/2**. Since if a node is red, then its two child must be black, so in the path from the node to root, red nodes cannot be more than one half of the height. Therefore, **black-height >= h/2.**
3. **Number of nodes >=2^(h/2) -1**
4. **Every Red Black Tree with n nodes has height <=**2Log2(n+1), which is strictly bounded by Log\_2 n
5. **Operation: Rotation**
6. **Operation: Insertion**
7. **Operation: Deletion**
8. Graph
9. Summary BFS vs DFS

b: Branching factor for example for binary tree b=2

d: depth of the tree

|  |  |  |
| --- | --- | --- |
|  | DFS | BFS |
| Worst time | O(b^{d+1}) | O(b^{d+1}) |
| Worst space | O(bd)\* | O(b^d) |
| Fewest node to starting node | No | Yes |

Worst space of DFS

\*For graph that contains circle, we need a set to store all visited node then the space complexity is O(b^d). For graph like a tree in which we do not need to store visited node, then the space complexity is O(bd)

1. **In place matrix transpose**
2. **Index mapping**

or: old row

oc: old col

nr: new row = oc

nc: new col = or

ol: old location

nl: new location

C: number of cols

R: number of rows

N: size of matrix N = C\*R

1. For an element in old matrix A[or][oc]

ol = or \* C + oc

1. The same element in new matrix

nl = nr \* R + nc

1. ol \* R = or \* C \* R + oc \* R

= or \* N + nr\*R

= or \* N +nl – nc

= or \* N + nl – or

= or \* ( N -1 ) + nl

1. nl = ol \* R – or \* (N-1)
2. nl = nl mod (N-1) (since nl <= N-1)

= (ol \* R) mod (N-1)

1. **c++ implementation**

void MatrixInplaceTranspose(int \*A, int r, int c)

{

    int size = r\*c - 1;

    int t; // element hold to move to the next

    int next; // location where t is going to move to

    int cycleBegin; // start location of cycle

    int i; // iterator

    bitset<HASH\_SIZE> b; // hash to mark moved elements

    b.reset();

    b[0] = b[size] = 1;

    i = 1; // Note that A[0] and A[size-1] won't move

    while (i < size)

    {

        cycleBegin = i;

        t = A[i]; hold the element in t, prepare to move it to the new location

        do

        {

            // i\_new = (i\*r)%(N-1)

            next = (i\*r)%size;

            swap(A[next], t);

            b[i] = 1;

            i = next;

        }

        while (i != cycleBegin);

        // Get Next Move (what about querying random location?)

        for (i = 1; i < size && b[i]; i++)

            ;

        cout << endl;

    }

}