

Homework Set # 2

Data Structure Engineering for Algorithm Design

Name: Zizhen Chen

SMU ID: 37366827

Data Structures deals with the **representation** and **organization** of data to support efficient:

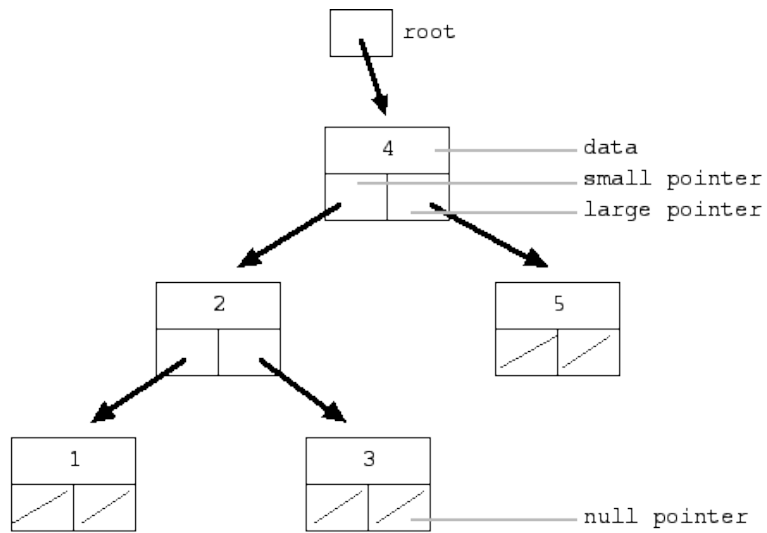
- **storage**
- **access**
- **execution of operations**
- **display of results.**

1. **[Data structure conversion](15 points)**. Design and analyze an on-the-fly algorithm for converting a binary search tree to an ordered doubly linked list following the inorder traversal of the data. Utilize a loop invariant (text p.17) to argue the correctness of your algorithm. Have your algorithm use as little extra space as possible, with "in place" the goal. Perform a walkthrough for the sequence S where the elements of S are first processed left-to-right to build the search tree. Then apply your algorithm to convert the binary search tree to a doubly linked list, showing the status of the conversion at each node of the search as it is consumed and moved into the doubly linked list.

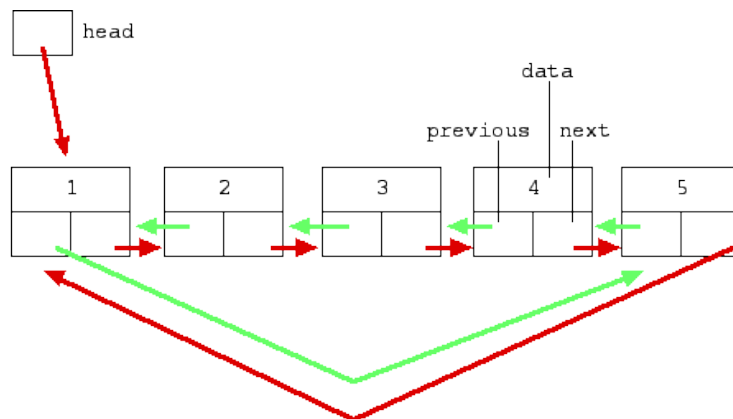
$S = 63, 35, 77, 49, 83, 28, 14, 45, 70, 12, 54, 78$

Answer:

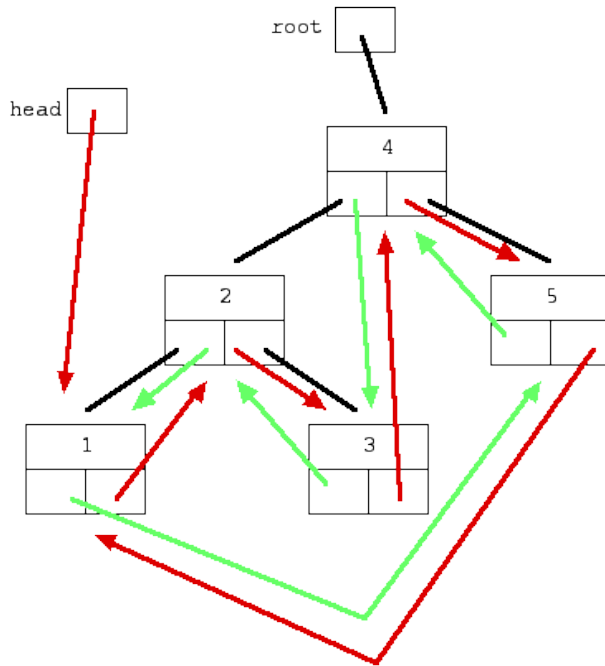
A binary search tree is like this:



A doubly linked circular list is like this.



Then we have to convert the binary search tree to doubly linked circular list which should be like this.



Since the binary search tree is made up of its left subtree, current node and its right subtree, we can use Divide and Conquer Method to recursively resolve the converting problem. Actually, this process is really like the inorder traversal of the binary search tree. First convert left subtree, then relink the current node's pointer, finally convert the right subtree. Then we can get such algorithm code in C++:

```
void convert(Node* treeNode, Node* &prevNode, Node* &head)
{
    //prevNode is used to keep track of previously traversed node
    if(treeNode==NULL)
        return;
    convert(treeNode->left, prevNode, head);
    treeNode->left=prevNode; //Current node's left pointer points to previous node
    if(prevNode!=NULL)
        prevNode->right=treeNode; //Previous node's right pointer points to current
node
    else
        head=treeNode;
    Node *right=treeNode->right; //Record the right pointed node (right subtree)
    head->left=treeNode;
    treeNode->right=head; //The currently end node's right pointer points to head node
    (This made a circularly double linked list!)
    prevNode=treeNode; //Update previous node
    convert(right, prevNode, head);
}
```

To verify it works correctly, I built a whole program (using C++ language) to test the S sets of numbers in this question. And the run result screenshot is listed below.

```
C:\Windows\system32\cmd.exe

Please enter 12 value to create a binary search tree.
63 35 77 49 83 28 14 45 70 12 54 78
Binary search tree created!

Now begin to in-order traverse the tree.
12 14 28 35 45 49 54 63 70 77 78 83

Now begin to convert the binary search tree to double linked list.
Converting process complete!

In order to show it is really converted:
First we traverse it from head node to end node.
12 14 28 35 45 49 54 63 70 77 78 83
Second, we traverse it from end node to head node.
83 78 77 70 63 54 49 45 35 28 14 12
```

From the C++ code above, we can see this algorithm won't need extra space depended on input size. That means the space complexity of this algorithm is $O(1)$ and we can say it is "in-place".

I talked with Professor Matula, in the case of my algorithm, it doesn't have any loops, only recursively called functions. It can't use loop invariant to analyze the correctness. To show I know what loop invariant is, I list the concept of it below.

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Here is a full program code attached.

```
// BinarySearchTreeToDoubleLinkedList.cpp : Defines the entry point for the console application.
//
```

```
#include "stdafx.h"
```

```
using std::cin;
using std::cout;
using std::endl;
```

```
struct Node
{
    int key;
    Node *left;
    Node *right;
};
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    void insertNode(Node*&,Node*);
    Node *root=NULL;
    int key=0;
    cout<<"Please enter 12 value to create a binary search tree."<<endl;
    for(int i=0;i<12;i++)
    {
        cin>>key;
        Node *newNode=new Node;
        newNode->key=key;
```

```

        newNode->left=NULL;
        newNode->right=NULL;
        insertNode(root,newNode);
    }
    cout<<"Binary search tree created!"<<endl<<endl<<"Now begin to in-order traverse
the tree."<<endl;
    void inorderTraversal(Node*);
    inorderTraversal(root);
    cout<<endl<<endl<<"Now begin to convert the binary search tree to double linked
list."<<endl;
    Node *prevNode=NULL,*head=NULL;
    void convert(Node*, Node* &, Node* &);
    convert(root,prevNode,head);
    cout<<"Converting process complete!"<<endl<<endl<<"In order to show it is really
converted:"<<endl<<"First we traverse it from head node to end node."<<endl;
    for(int i=0;i<12;i++){
        cout<<head->key<<' ';
        head=head->right;
    }
    cout<<endl<<"Second, we traverse it from end node to head node."<<endl;
    for(int i=0;i<12;i++){
        head=head->left;
        cout<<head->key<<' ';
    }
    cout<<endl;
    for(int i=0;i<12;i++){
        Node *node=NULL;
        node=head;
        head=head->right;
        delete node;
    }
    return 0;
}

void insertNode(Node* &treeNode,Node *newNode)
{//this the insert node function can be used to create a new binary search tree
    if(treeNode==NULL)
        treeNode=newNode;
    else if(newNode->key<treeNode->key)
        insertNode(treeNode->left,newNode);
    else
        insertNode(treeNode->right,newNode);
}

void inorderTraversal(Node* treeNode)
{
    if(treeNode!=NULL)
    {
        inorderTraversal(treeNode->left);
        cout<<treeNode->key<<' ';
        inorderTraversal(treeNode->right);
    }
}

void convert(Node* treeNode, Node* &prevNode,Node* &head)
{//prevNode is used to keep track of previously traversed node
    if(treeNode==NULL)
        return;

```

```

convert(treeNode->left,prevNode,head);
treeNode->left=prevNode;//Current node's left pointer points to previous node
if(prevNode!=NULL)
    prevNode->right=treeNode;//Previous node's right pointer points to current
node
else
    head=treeNode;
Node *right=treeNode->right;//Record the right pointed node (right subtree)
head->left=treeNode;
treeNode->right=head;//The currently end node's right pointer points to head node
(This made a circularly double linked list!)
prevNode=treeNode;//Update previous node
convert(right,prevNode,head);
}

```

2. **[Heapify] (15 points).** The array form of a heap (i.e. a binary heap array) has the children of the element at position i in positions (left @ $2i$, right @ $2i+1$). For a balanced ternary heap array the three children of the element at position i are at positions (left @ $3i-1$, middle @ $3i$, right @ $3i+1$). The term "Heapify" refers to the algorithm that builds a binary heap array from right-to-left (leaf-to-root in binary tree form) given the size of the array to be built. The term "Ternary Heapify" here refers to the analogous algorithm that builds a balanced ternary heap array in right-to-left order.

- Heapify the sequence S of Problem 1, and give the number of comparisons required.
- Do the same as (a) for Ternary Heapify.
- Build the decision tree determined by the comparisons employed to Heapify a six element list: $a_1, a_2, a_3, a_4, a_5, a_6$. What is then the worst case and average case number of comparisons to Heapify six elements? Is this optimal in either case?
- Do the same as (c) for Ternary Heapify.

Answer:

- Since the question hasn't pointed out whether the heap is a max heap or min heap, I assume it is a max heap. Then the algorithm (in C++ code) has two main function listed below.

```

void maxHeapify(int array[],int length,int i)
{//move i to the appropriate place in the max heap stored in array
    int leftChild=2*i+1,rightChild=2*i+2,largest=-1;
    if((1<length)&&(array[leftChild]>array[i]))
        largest=1;
    else
        largest=i;
    if((r<length)&&(array[rightChild]>array[largest]))
        largest=r;
    if(largest!=i)
    {
        array[i]^=array[largest]^=array[i]^=array[largest];//swap array[i] and
array[largest]
        maxHeapify(array,length,largest);
    }
}

void buildMaxHeap(int array[],int length)

```

```

{ //repeatedly use maxHeapify to build a max heap
  for(int i=length/2-1;i>=0;i--)
    maxHeapify(array,length,i);
}

```

The first one is maxHeapify() function which is to move i to the appropriate place in the max heap stored in array. The second is buildMaxHeap() function which is to use maxHeapify() function repeatedly to build a max heap.

There are several places which need to pay attention.

First, is in the first line of maxHeapify() function which is

```
int leftChild=2*i+1,rightChild=2*i+2,largest=-1;
```

leftChild stores array index of left child of i , rightChild stores array index of right child of i and i itself is also the index of i . The point needs to pay attention is normally, the array form of a heap (i.e. a binary heap array) has the children of the element at position i in positions (left @ $2i$, right @ $2i+1$). However, since I use C++ code to build the algorithm and C++ code's array index starts from 0, so the left child is at $2i+1$ and the right child is at $2i+2$.

Second, is the swap operation in the maxHeapify() function which is

```
array[i]^=array[largest]^=array[i]^=array[largest];
```

Here I use “xor” method to swap two values and it can save a temp variable space.

Third, is in the buildMaxHeap() function which is to heapify all non-leaf nodes, then a max heap will be built. Normally, in a heap, the non-leaf nodes indices end at floor of $(\text{length}/2)$, however, since C++ array indices start from 0 so the non-leaf nodes indices end at $\text{length}/2-1$. Therefore, the for loop in buildMaxHeap() function starts from index of $(\text{length}/2-1)$ to 0.

Then we can use the S sets of numbers in question 1 to call the buildMaxHeap() function and the run result screenshot is like below:

```

C:\Windows\system32\cmd.exe
The inputed numbers are:
63 35 77 49 83 28 14 45 70 12 54 78
The max heap of these numbers are stored in such order in an array:
83 70 78 63 54 77 14 45 49 12 35 28

```

Finally, the number of comparison required is 18.

- b. Just like the binary heapify above, I also built a similar C++ program to finish the ternary heapify (also assume it is a max heap) process with maxHeapify() and buildMaxHeap() function. It also has some similar points to pay attention.

First, for a balanced ternary heap array the three children of the element at position i are at positions (left @ $3i-1$, middle @ $3i$, right @ $3i+1$). Since C++ array indices start from 0, the positions change to (left @ $3i+1$, middle @ $3i+2$, right @ $3i+3$).

Second, normally, in a ternary heap, the non-leaf nodes indices end at ceil of $((\text{length}-1)/3)$, however, since C++ array indices start from 0 so the non-leaf nodes indices end at $(\text{ceil of } ((\text{length}-1)/3))-1$. Therefore, the for loop in buildMaxHeap() function starts from index of $(\text{ceil of } ((\text{length}-1)/3))-1$ to 0.

Below is the C++ code of these two functions.

```

void maxHeapify(int array[],int length,int i)
{//move i to the appropriate place in the max heap stored in array
    int leftChild=3*i+1,middleChild=3*i+2,rightChild=3*i+3,largest=-1;
    if((leftChild<length)&&(array[leftChild]>array[i]))
        largest=leftChild;
    else
        largest=i;
    if((middleChild<length)&&(array[middleChild]>array[largest]))
        largest=middleChild;
    if((rightChild<length)&&(array[rightChild]>array[largest]))
        largest=rightChild;
    if(largest!=i)
    {
        array[i]^=array[largest]^=array[i]^=array[largest];{//swap array[i] and
array[largest]
        maxHeapify(array,length,largest);
    }
}


void buildMaxHeap(int array[],int length)
{//repeatedly use maxHeapify to build a max heap
    for(double i=ceil((length-1)/3.0)-1;i>=0;i--)
        maxHeapify(array,length,int(i));
}

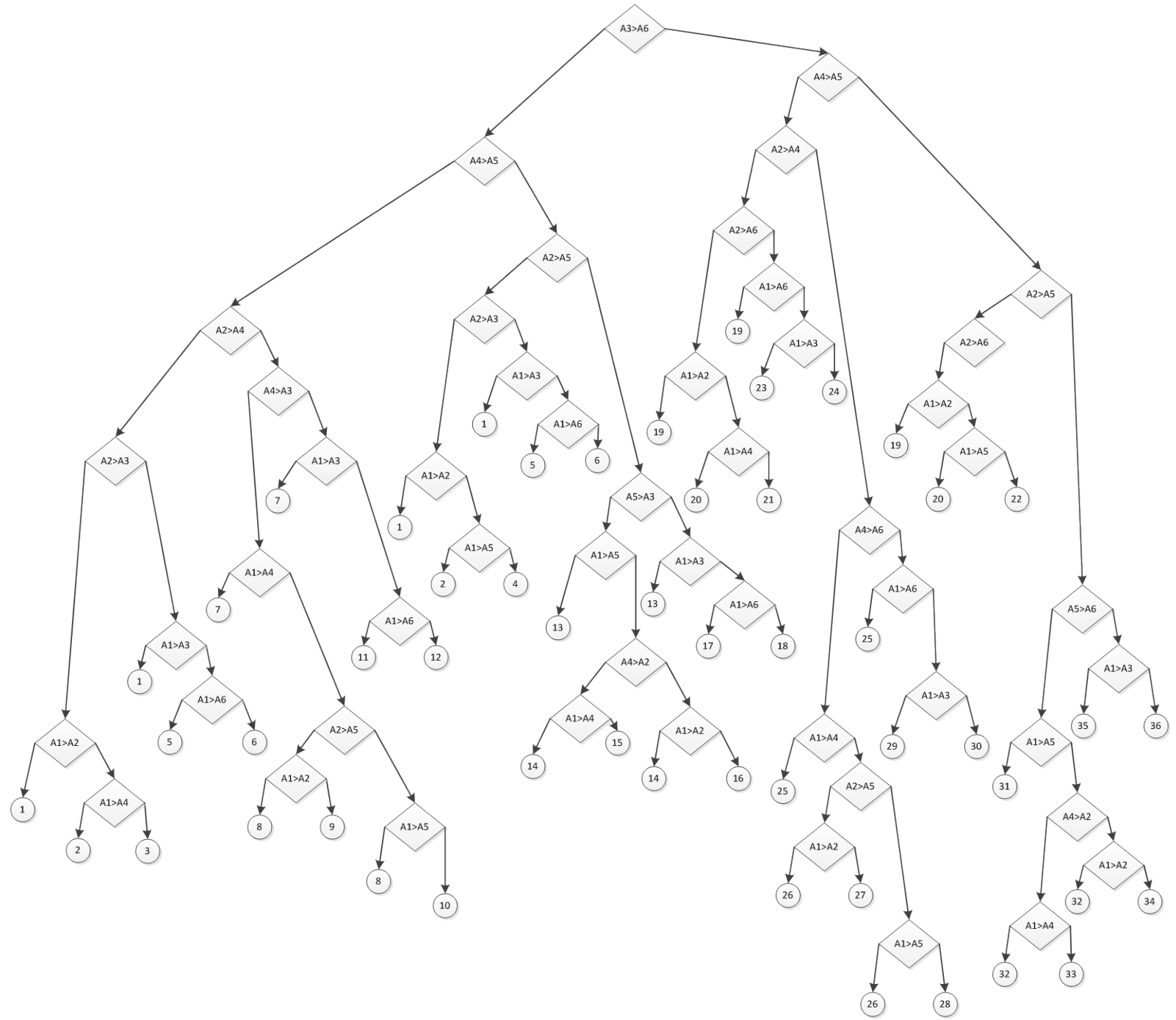
```

The run result screenshot is listed below.

Finally, the number of comparison required is 14.

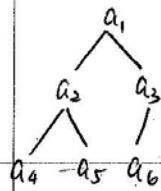
Actually, both binary heapify or ternary heapify algorithm time complexity is $O(n)$.

- c. The decision tree is listed below. In this decision tree, each leaf like  is one state of the heap, there are 36 heap states at all.
 Left child is the result of Yes, right child is the result of No.
 I listed all the states of heap in the next page of this decision tree.
 The worst case number of comparisons to Binary Heapify six elements is 7.
 The average case number of comparisons to Binary Heapify six elements is 6.
 I listed all comparison numbers data of every state in each case in the next pages of this decision tree, too. The data format is "Index State Comparisons".

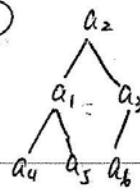


All states of binary heaps:

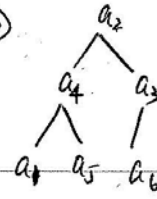
①



②

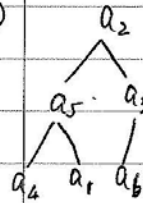


③

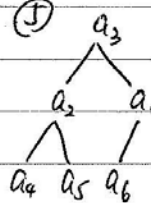


Binary
Heapify
States.

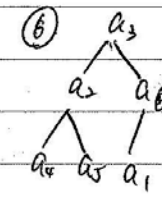
④



⑤

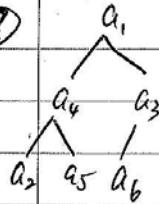


⑥

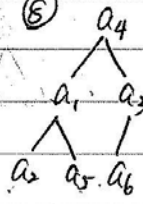


(Both Sides)

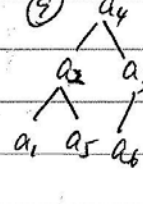
⑦



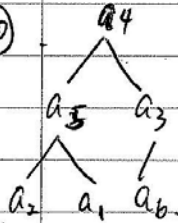
⑧



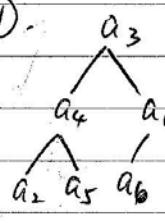
⑨



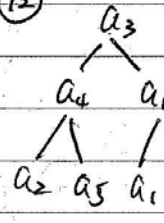
⑩



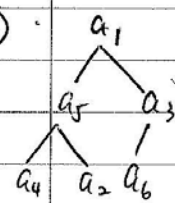
⑪



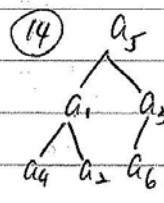
⑫



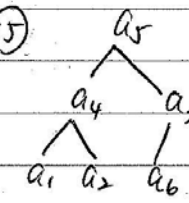
⑬



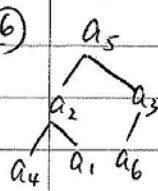
⑭



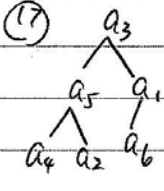
⑮



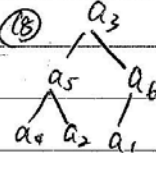
⑯

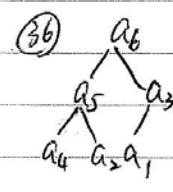
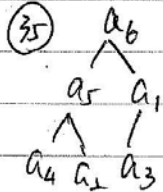
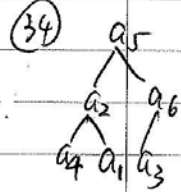
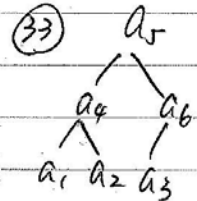
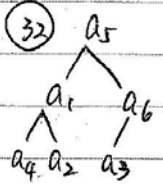
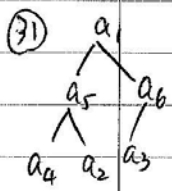
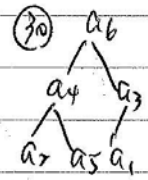
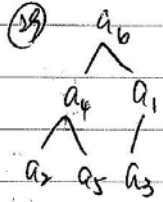
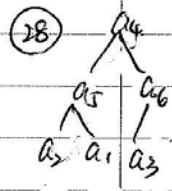
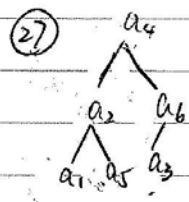
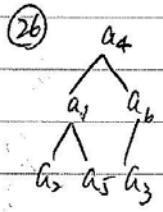
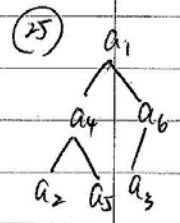
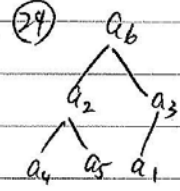
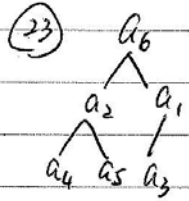
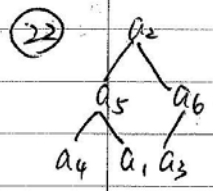
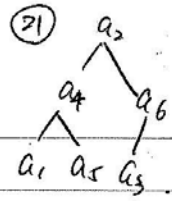
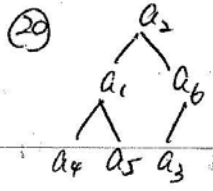
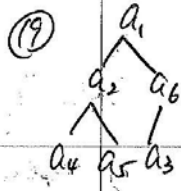


⑰




⑱





All comparison numbers data of every state:

Binary Heapify Record		
1. ① : 5	16. ② 6	34. ②④ 6
2. ② : 6	17. ④ 6	35. ②⑤ 5
3. ③ : 6	18. ① 5	36. ②⑥ 7
4. ① : 5	19. ⑤ 6	37. ②⑦ 7
5. ⑤ : 6	20. ⑥ 6	38. ②⑧ 7
6. ⑥ : 6	21. ⑬ 5	39. ②⑧ 7
7. ⑦ : 5	22. ⑭ 7	40. ②⑤ 5
8. ⑧ : 7	23. ⑮ 7	41. ②⑨ 6
9. ⑨ : 7	24. ⑭ 7	42. ③⑩ 6
10. ⑧ : 7	25. ⑮ 7	43. ①⑨ 5
11. ⑩ : 7	26. ⑬ 5	44. ②⑩ 6
12. ⑦ : 5	27. ⑰ 6	45. ②② 6
13. ⑪ : 6	28. ⑬ 5	46. ③① 5
14. ⑫ : 6	29. ⑱ 5	47. ③② 7
15. ① : 5	30. ⑳ 6	48. ③③ 7
	31. ②① 6	49. ③④ 7
	32. ⑰ 5	50. ③④ 7
	33. ③③ 6	51. ③⑤ 5
		52. ③⑥ 5

- d. The decision tree is listed below. In this decision tree, each leaf like  is one state of the heap, there are 18 heap states at all.

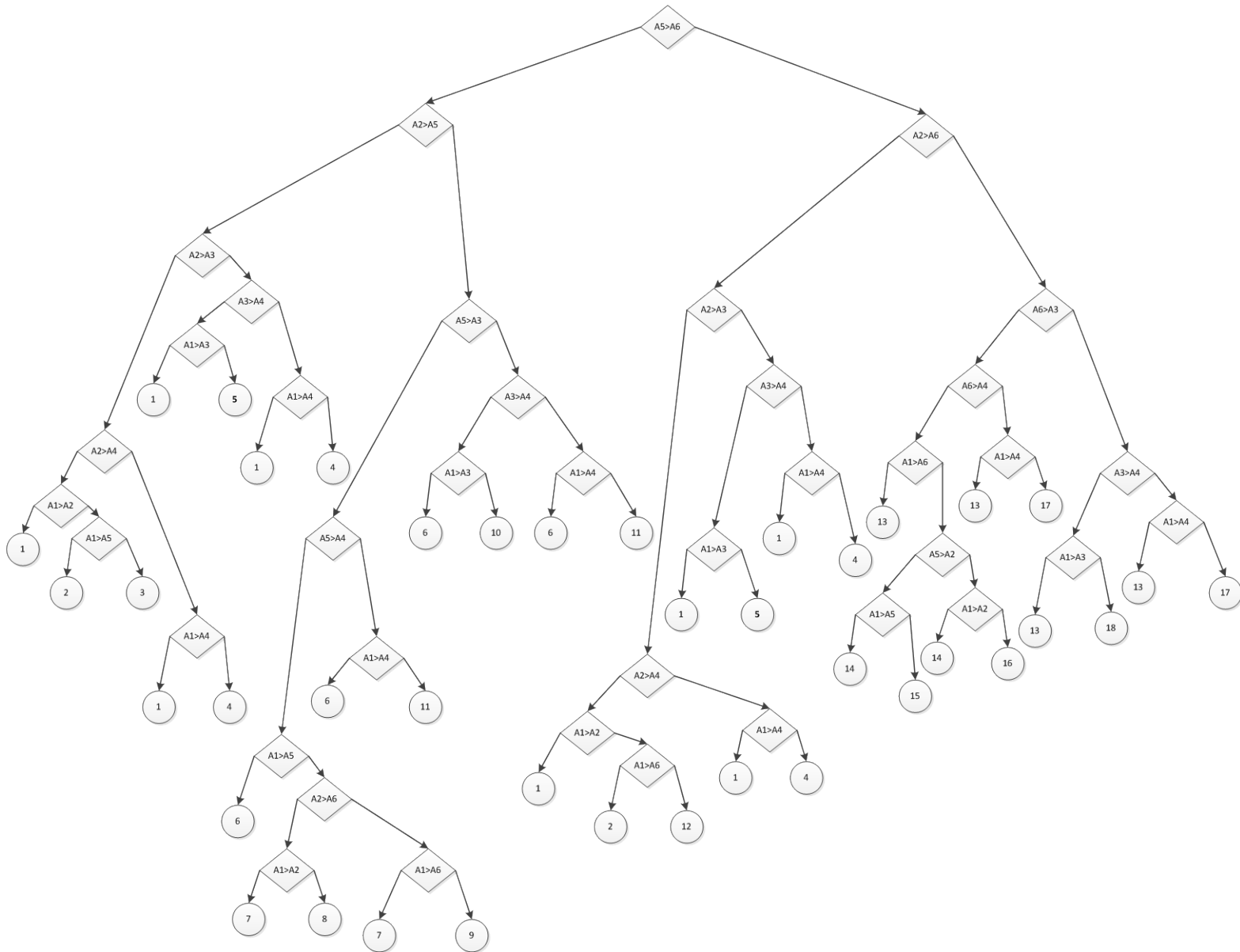
Left child is the result of Yes, right child is the result of No.

I listed all the states of heap in the next page of this decision tree.

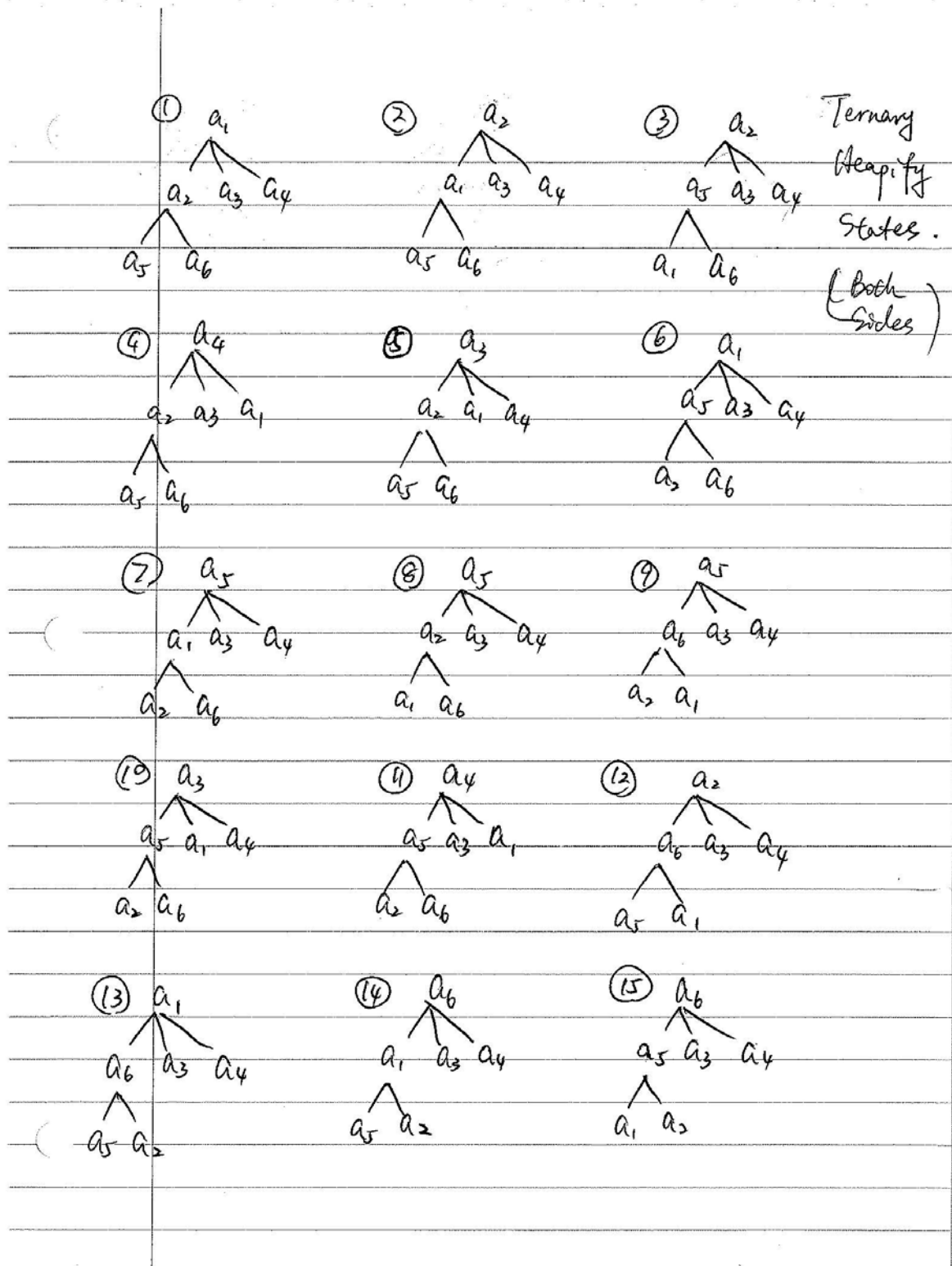
The worst case number of comparisons to Ternary Heapify six elements is 7.

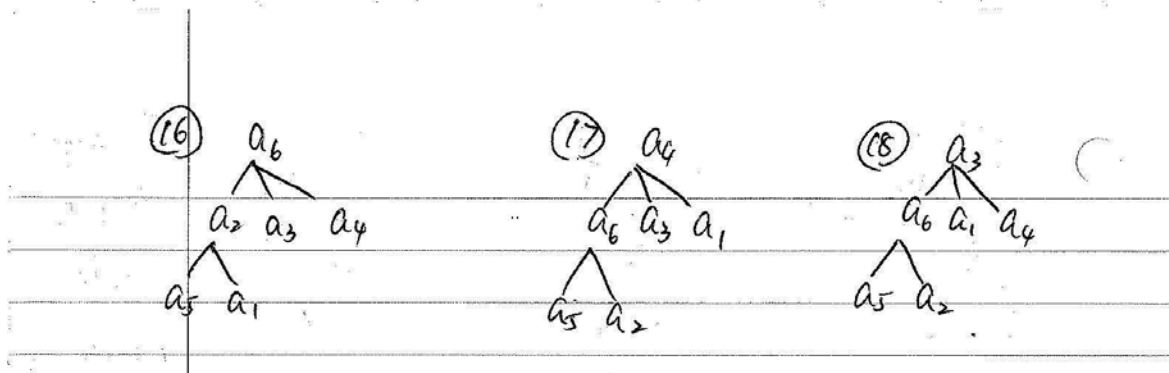
The average case number of comparisons to Ternary Heapify six elements is 5.5.

I listed all comparison numbers data of every state in each case in the next pages of this decision tree, too. The data format is "Index State Comparisons".



All states of ternary heaps:





All comparison numbers data of every state:

Ternary Heapify Record -

1. ① 5	19. ⑥ 5	38. ⑱ 5.
2. ② 6	20. ⑪ 5.	39. ⑲ 5
3. ③ 6.	21. ① 5	40. ⑰ 5.
4. ① 5	22. ② 6	
5. ④ 5.	23. ⑫ 6	
6. ① 5	24. ① 5	
7. ⑤ 5	25. ④ 5.	
8. ① 5	26. ① 5	
9. ④ 5	27. ⑤ 5	
10. ⑥ 5	28. ① 5	
11. ⑦ 7	29. ④ 5	
12. ⑧ 7	30. ⑬ 5	
13. ⑦ 7	31. ⑭ 7	
14. ⑧ 7	32. ⑮ 7	
15. ⑥ 5	33. ⑭ 7	
16. ⑪ 5	34. ⑯ 7	
17. ⑥ 5	35. ⑬ 5	
18. ⑩ 5	36. ⑰ 5	
	37. ⑬ 5	

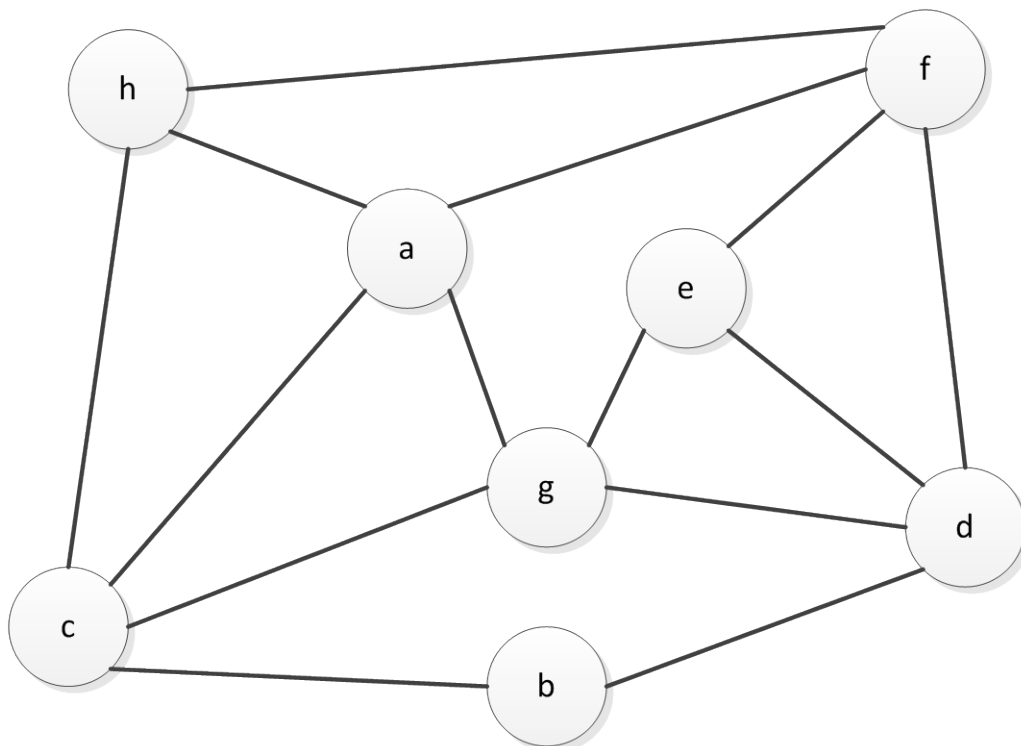
3. **[Graph (Symmetric Matrix) Reordering]** Let the adjacency list representation of a graph (0, 1 symmetric matrix) be given by a packed edge list. Describe an algorithm for reordering the edge list in place so all adjacency lists follow a new vertex ordering given by a permutation $p(1)$, $p(2)$, ..., $p(n)$. Apply your algorithm to the graph given by the packed edge list stored as example HW2 - 3 graph. Reorder by determining a smallest last search order with "ties" broken lexicographically.

HW2 - 3 graph in adjacency list form:

- a: c, f, g, h;
- b: c, d;
- c: a, b, g, h;
- d: b, e, f, g;
- e: d, f, g;
- f: a, d, e, h;
- g: a, c, d, e;
- h: a, c, f;

Answer:

From the graph in adjacency list above, we can establish graph like below:



And we can also get degree of each node. I listed them below:

- a: 4;
- b: 2;

- c: 4;
- d: 4;
- e: 3;
- f: 4;
- g: 4;
- h: 3;

So we can rename node a~e with numbers according to their degrees(from large to small) like below:

- a: 1;
- b: 8;
- c: 2;
- d: 3;
- e: 6;
- f: 4;
- g: 5;
- h: 7;

From the original adjacency list, we can get

a				b		c				d				e		f				g				h				
c	f	g	h	c	d	a	b	g	h	b	e	f	g	d	f	g	a	d	e	h	a	c	d	e	a	c	f	

After renamed them:

1				8		2				3				6		4				5				7				
2	4	5	7	2	3	1	8	5	7	8	6	4	5	3	4	5	1	3	6	7	1	2	3	6	1	2	4	

Then we need to compress it, the compress rule is like this, if the node number is larger than the according list numbers, then the we omit the list numbers. For example:

4			
1	3	6	7

After compress, it becomes

4	
6	7

Therefore, after compressed the list becomes and radix sort the lists nodes (List A)

1				2				3				4		5
2	4	5	7	5	7	8	4	5	6	8	6	7	6	

Then we use radix sort the list nodes and ready for decompress it. (List B)

2	4		5			6			7			8		
1	1	3	1	2	3	3	4	5	1	2	4	2	3	

Now, we combine the compressed list (List A) and the sorted list (List B) to get the reordered whole list!

1				2				3				4			5				6			7			8		
2	4	5	7	1	5	7	8	4	5	6	8	1	3	6	7	1	2	3	6	3	4	5	1	2	4	2	3

Since we renamed node with numbers, then we rename back the result list and get

a				c				d				f			g				e			h				b	
c	f	g	h	a	g	h	b	f	g	e	b	a	d	e	h	a	c	d	e	d	f	g	a	c	f	c	d

The pseudocode of this algorithm

//Compress Process

int i = 0;

int j;

LOOP: if the adjacency list is not empty

pointer = adjacency list[i++].List point;

j = 0;

LOOP: if point[j] is not equal NULL

if adjacency list[i].vertex > point[j++].V

Delete this V and free space;

LOOP END

insert NULL at the end of V list

LOOP END

//Ready for Decompress

i = 0;

LOOP: if the adjacency list is not empty

find the smallest vertex in adjacency list[s] from i to size of
adjacency list;

exchange position between adjacency list[i++] and adjacency
list[s];

LOOP END

//Dcompress

i = size of adjacency list

LOOP: if i > 0

j = i-1;

Loop: if j >= 0

if there is a vertex i in adjacency list[j];

insert j into adjacency list[i];

j--;

LOOP END

i--;

LOOP END

4. **[Sparse Matrix Multiplication]** Describe an efficient sparse matrix multiplication procedure for two $n \times n$ matrices stored in "adjacency list" form. Assume matrix A has m_A non zero entries and matrix B has m_B non zero entries, with $n \ll m_A \ll m_B \ll n^2$. Describe why your algorithm will generate the $n \times n$ matrix product in adjacency list form in time $O(n m_A)$.

Apply your algorithm for multiplying the two 8×8 0,1 matrices A_1, B_1 where A_1 is given by the following adjacency list of unit entries and B_1 is given by the adjacency list of unit entries of Problem 3. Count the actual number of multiplications used in this matrix product and explain why it is considerably less than the bound $n m_A$ (which is $8 \times 19 = 152$) in this case.

Adjacency List For 0,1 Matrix A_1

- 1: 3, 5, 7
- 2: 4, 8
- 3: 5, 8
- 4: 2, 6
- 5: 1, 2, 7, 8
- 6: 3
- 7: 1, 5, 6
- 8: 6, 7

Answer:

We are going to calculate $n \times n$ sparse matrix multiplication and store them in adjacency list. Assume we have such two adjacency list represents matrix A and B. First, in order to keep it simply, we assume matrix A and B are both “0,1” matrices.

Matrix A:

	1	2	3	4	5	6	7	8
1	0	0	1	0	1	0	1	0
2	0	0	0	1	0	0	0	1
3	0	0	0	0	1	0	0	1
4	0	1	0	0	0	1	0	0
5	1	1	0	0	0	0	1	1
6	0	0	1	0	0	0	0	0
7	1	0	0	0	1	1	0	0
8	0	0	0	0	0	1	1	0

Matrix B:

	1	2	3	4	5	6	7	8
1	0	0	1	0	0	1	1	1
2	0	0	1	1	0	0	0	0
3	1	1	0	0	0	0	1	1
4	0	1	0	0	1	1	1	0
5	0	0	0	1	0	1	1	0
6	1	0	0	1	1	0	0	1
7	1	0	1	1	1	0	0	0
8	1	0	1	0	0	1	0	0

And the according adjacencies are:

Adjacency A:

1	3	5	7	
2	4	8		
3	5	8		
4	2	6		
5	1	2	7	8
6	3			
7	1	5	6	
8	6	7		

Adjacency B:

1	3	6	7	8
2	3	4		
3	1	2	7	8
4	2	5	6	7
5	4	6	7	
6	1	4	5	8
7	1	3	4	5
8	1	3	6	

We imagine the answer matrix is like this:

	1	2	3	4	5	6	7	8
1	c1	c2	c3	c4	c5	c6	c7	c8
2	c9	c10	c11	c12	c13	c14	c15	c16
3	c17	c18	c19	c20	c21	c22	c23	c24
4	c25	c26	c27	c28	c29	c30	c31	c32
5	c33	c34	c35	c36	c37	c38	c39	c40
6	c41	c42	c43	c44	c45	c46	c47	c48
7	c49	c50	c51	c52	c53	c54	c55	c56
8	c57	c58	c59	c60	c61	c62	c63	c64

Then the multiplication of the two matrices can be like this:

1) List 1 of Adjacency A is 3, 5 and 7

2) Find List 3, 5 and 7 in Adjacency B whether there are "1" in these three lists. Count how many "1" in these three lists. If there are n_1 "1"s, then the value of c1 in then answer matrix is n_1 .

3) Find List 3, 5 and 7 in Adjacency B whether there are "2" in these three lists. Count how many "2" in these three lists. If there are n_2 "2"s, then the value of c2 in then answer matrix is n_2 .

4) Find List 3, 5 and 7 in Adjacency B whether there are "3" in these three lists. Count how many "3" in these three lists. If there are n_3 "3"s, then the value of c3 in then answer matrix is n_3 .

...

Repeat the steps...until

9) Find List 3, 5 and 7 in Adjacency B whether there are "8" in these three lists. Count how many "8" in these three lists. If there are n_8 "8"s, then the value of c8 in then answer matrix is n_8 .

Up to now, we finished first row of the answer Matrix. Actually, we don't always need to find from c1 to c8 for 8 times and do them for 9 steps at all. We can do from an opposite way.

We can just search from list 3 and if encountered "1", the value of c1 plus 1, if encountered "2" the value of c2 plus 1, ..., if encountered "8", the value of c8 plus 1. Then we begin to search list 5 and list 7 like this way. Finally, when we searched all the three lists, we finished the first row of the answer Matrix. Then we only need to do the number of times which is just the whole sum of length of these three lists. If the sum of length of these three lists is really small, then we do small times.

Then we need to finish all the lists in Adjacency A like we did in the first list above and we will finish all the rows of the answer Matrix.

From the above description, we can easily find out the time complexity of this algorithm.

Here we have two sparse matrixes A and B. A has m_A non-zero entries and B has m_B non-zero entries.

For each list in adjacency lists A, we have A_i entries. And $A_1 + A_2 + \dots + A_n = m_A$

Same as Adjacency lists B. Let us set C is the answer matrix, then $C = A * B$.

For first entry in row one of C, the max times of addition we need do is A_1

Then we get the operations for the first row of C in worst case:

$$A_1 + A_1 + \dots + A_1 = A_1 * n$$

Then we got:

row 1: $A_1 * n$

row 2: $A_2 * n$

.....

row n: $A_n * n$

$$\text{Total} = \text{row1} + \text{row2} + \dots + \text{row n} = (A_1 + A_2 + \dots + A_n) * n = m_A * n$$

So, we get the upper bound $O(m_A * n)$

For the algorithm code in C++, first, we define a Node struct.

```
struct Node
{
    int key;
    Node *next;
};
```

It is one node in adjacency list.

Then we can create adjacency lists for the matrix like this:

```
void createAdjacency(int matrix[][8], Node lists[])
{
    for(int i=0; i<8; i++)
    {
        lists[i].key=i+1;
        lists[i].next=NULL;
        Node *pointer=&lists[i];
        for(int j=0; j<8; j++)
            if(matrix[i][j]!=0)
            {
                Node *node=new Node;
                node->key=j+1;
                node->next=NULL;
                pointer->next=node;
            }
    }
}
```

```

        pointer=pointer->next;
    }
}

```

Now we can interview the main multiply algorithm C++ code:

```

void multiply(Node listsA[],Node listsB[], int matrix[][8])
{
    for(int i=0;i<8;i++)//this part is to initialize answer matrix
        for(int j=0;j<8;j++)//it should not be calculated in algorithm
            matrix[i][j]=0;
    Node *pointerA=NULL,*pointerB=NULL;
    for(int i=0;i<8;i++)
    {
        pointerA=listsA[i].next;
        while(pointerA!=NULL)
        {
            pointerB=listsB[pointerA->key-1].next;
            while(pointerB!=NULL)
            {
                matrix[i][pointerB->key-1]++; //since they are 0,1 matrices
                pointerB=pointerB->next;
            }
            pointerA=pointerA->next;
        }
    }
}

```

The run result screenshot of the program is listed below:

```

C:\Windows\system32\cmd.exe
Please inputer values of matrix A:
0 0 1 0 1 0 1 0
0 0 0 1 0 0 0 1
0 0 0 0 1 0 0 1
0 1 0 0 0 1 0 0
1 1 0 0 0 0 1 1
0 0 1 0 0 0 0 0
1 0 0 0 1 1 0 0
0 0 0 0 0 1 1 0

Please inputer values of matrix B:
0 0 1 0 0 1 1 1
0 0 1 1 0 0 0 0
1 1 0 0 0 0 1 1
0 1 0 0 1 1 1 0
0 0 0 1 0 1 1 0
1 0 0 1 1 0 0 1
1 0 1 1 1 0 0 0
1 0 1 0 0 1 0 0

The adjacency list A is listed below:
1: 3 5 7
2: 4 8
3: 5 8
4: 2 6
5: 1 2 7 8
6: 3
7: 1 5 6
8: 6 7

The adjacency list B is listed below:
1: 3 6 7 8
2: 3 4
3: 1 2 7 8
4: 2 5 6 7
5: 4 6 7
6: 1 4 5 8
7: 1 3 4 5
8: 1 3 6

The result Matrix is:
2 1 1 2 1 1 2 1
1 1 1 0 1 2 1 0
1 0 1 1 0 2 1 0
1 0 1 2 1 0 0 1
2 0 4 2 1 2 1 1
1 1 0 0 0 0 1 1
1 0 1 2 1 2 2 2
2 0 1 2 2 0 0 1

The actual multiplication number is: 66

```

Since I built a program for this algorithm, it is really easy to count how many times the algorithm really calculated (real time complexity). We only need to add a global variable “count” and initialize it to 0; Then add count++ in the inner while loop like below.

```

while(pointerB!=NULL)
{
    matrix[i][pointerB->key-1]++;
    pointerB=pointerB->next;
    count++;
}

```

Then the actual multiplication number is showed like below which is 66.

```
The result Matrix is:
2 1 1 2 1 1 2 1
1 1 1 0 1 2 1 0
1 0 1 1 0 2 1 0
1 0 1 2 1 0 0 1
2 0 4 2 1 2 1 1
1 1 0 0 0 0 1 1
1 0 1 2 1 2 2 2
2 0 1 2 2 0 0 1

The actual multiplication number is: 66
```

If we calculate $n \cdot m_A$ which is $8 \times 19 = 152$, we can see the actual multiplication is considerably less than the upper bound. The reason is simple, we calculate the upper bound for worst case which is

For first entry in row one of C, the max times of addition we need do is A_1

Then we get the operations for the first row of C in worst case:

$$A_1 + A_1 + \dots + A_1 = A_1 * n$$

But actually how many A_1 addition for the first row of matrix C is depend on the length of each list in adjacency lists B, we can found that the longest length of the lists is 4 which is only half of n . Therefore, in actual multiplication, the most worst cast of operations for one row of C matrix is $A_1 * n/2$. So the actual number should be less than $152/2=76$ which is 66.

Above is a simple situation which assume matrices are “0,1” matrices. That is totally for simply thinking of the algorithm. If we are dealing with sparse normal matrices which the values can be any numbers, then we only need to mainly change two places of the algorithm in C++ code.

The first change is the Node struct, we should add a value part in it which stores the according value in the matrix.

```
struct Node
{
    int key,value; //added a value part
    Node *next;
};
```

Second, since it is not “0,1” matrix, we couldn’t simply using plus 1 to calculate the result entries in the answer matrix. We should get the product of the value of one node in one list of adjacency lists A and the value of one node in one list of adjacency lists B, then accumulate adding them to the entry of the answer matrix. The multiplication code only need to be changed as follow:

```
void multiply(Node listsA[],Node listsB[], int matrix[][8])
{
    for(int i=0;i<8;i++)//this part is to initialize answer matrix
        for(int j=0;j<8;j++)//it should not be calculated in algorithm
```

```

        matrix[i][j]=0;
Node *pointerA=NULL,*pointerB=NULL;
for(int i=0;i<8;i++)
{
    pointerA=listsA[i].next;
    while(pointerA!=NULL)
    {
        pointerB=listsB[pointerA->key-1].next;
        while(pointerB!=NULL)
        {//below is the changing place!
            matrix[i][pointerB->key-1]+=pointerA->value*pointerB->value;
            pointerB=pointerB->next;
        }
        pointerA=pointerA->next;
    }
}
}

```

In order to verify it is right, we calculate another two sparse matrices

	1	2	3	4	5	6	7	8			1	2	3	4	5	6	7	8
1	0	0	4	0	1	0	1	0		1	0	0	7	0	0	1	1	1
2	0	0	0	1	0	0	0	1		2	0	0	1	5	0	0	0	0
3	0	0	0	0	6	0	0	1		3	1	1	0	0	0	0	1	1
4	0	1	0	0	0	1	0	0		4	0	1	0	0	1	4	1	0
5	1	1	0	0	0	0	1	8		5	0	0	0	1	0	1	1	0
6	0	0	1	0	0	0	0	0		6	1	0	3	2	1	0	0	1
7	1	0	0	2	1	1	0	0		7	1	0	1	1	1	0	0	0
8	0	0	0	0	0	1	1	0		8	1	0	1	0	0	6	0	0

The result matrix should be:

	1	2	3	4	5	6	7	8
1	5	4	1	2	1	1	5	4
2	1	1	1	0	1	10	1	0
3	1	0	1	6	0	12	6	0
4	1	0	4	7	1	0	0	1
5	9	0	17	6	1	49	1	1
6	1	1	0	0	0	0	1	1
7	1	1	10	4	2	7	4	2
8	2	0	4	3	2	0	0	1

The run result screenshot is:

```

C:\Windows\system32\cmd.exe
Please inputer values of matrix A:
0 0 4 0 1 0 1 0
0 0 0 1 0 0 0 1
0 0 0 0 6 0 0 1
0 1 0 0 0 1 0 0
1 1 0 0 0 0 1 8
0 0 1 0 0 0 0 0
1 0 0 1 2 1 0 0
0 0 0 0 0 1 1 0

Please inputer values of matrix B:
0 0 7 0 0 1 1 1
0 0 1 5 0 0 0 0
1 1 0 0 0 0 1 1
0 1 0 0 1 4 1 0
0 0 0 1 0 1 1 0
1 0 3 2 1 0 0 1
1 0 1 1 1 0 0 0
1 0 1 0 0 6 0 0

The adjacency list A is listed below:
1: 3 5 7
2: 4 8
3: 5 8
4: 2 6
5: 1 2 7 8
6: 3
7: 1 4 5 6
8: 6 7

The adjacency list B is listed below:
1: 3 6 7 8
2: 3 4
3: 1 2 7 8
4: 2 5 6 7
5: 4 6 7
6: 1 3 4 5 8
7: 1 3 4 5
8: 1 3 6

The result Matrix is:
5 4 1 2 1 1 5 4
1 1 1 0 1 10 1 0
1 0 1 6 0 12 6 0
1 0 4 7 1 0 0 1
9 0 17 6 1 49 1 1
1 1 0 0 0 0 1 1
1 1 10 4 2 7 4 2
2 0 4 3 2 0 0 1

The actual multiplication number is: 73

```

5. [Random Geometric Graphs: Generation and Display]

- i. Generate and display a random geometric graph (RGG) with $n = 200$ vertices and $r = 0.16$.
- ii. Generate an RGG with $n = 1600$ vertices and $r = 0.06$. Display all the vertices and only the edges incident to either a particular vertex of the highest degree or a particular vertex of lowest degree (give the degrees of these two vertices).
- iii. [5 bonus points] Generate an RGG with 400 vertices on the unit sphere with $r = 0.25$, where distance is measured between the (x, y, z) coordinates. Then display the graph for the hemisphere with $z \geq 0$, and separately for the hemisphere with $z \leq 0$.

Answer:

I use Processing Language to draw the graph. Processing is an open source programming language based on Java and environment for people who want to create images, animations, and interactions. Since it's based on Java, its grammar is as same as Java's. It doesn't need to be installed in Windows and also supports other platforms like Mac OS and Linux. The download link is: <http://processing.org/download/>.

1. It just to display a graph with 200 vertices and the edges which are with $r=0.16$ property. So I just randomly generate vertices and use a distance calculating formula to determine whether to draw the edge. The formula is: If you have two vertices of coordinates

$a(x_a, y_a)$ and $b(x_b, y_b)$ then the distance of $ab = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$ the code is listed below:

```
void setup(){  
    Point[] points;  
  
    size(600,600); //square size is 600 px *600 px  
  
    background(0,0,0);  
  
    stroke(255,255,0);  
  
    points=new Point[200];  
  
    for(int i=0;i<200;i++)  
        points[i]=new Point();  
  
    for(int i=0;i<200;i++){ //randomly generate vertices' coordinates.  
        float xCoordinate=random(600),yCoordinate=random(600);  
        point(xCoordinate,yCoordinate); //draw point  
        points[i].xCoordinate=xCoordinate;  
        points[i].yCoordinate=yCoordinate;  
    }  
  
    for(int i=0;i<200;i++)  
        for(int j=0;j<200;j++){  
            if(i==j)
```

```

        continue;

        if(sqrt(sq(points[i].xCoordinate-points[j].xCoordinate)+sq(points[i].yCoordinate-
points[j].yCoordinate))<600*0.16)//r=0.16 for a unit square, so in reality r should be 600*0.16

        line(points[i].xCoordinate,points[i].yCoordinate,points[j].xCoordinate,points[j].yCoordinate);//dr
        aw line

    }

}

class Point{

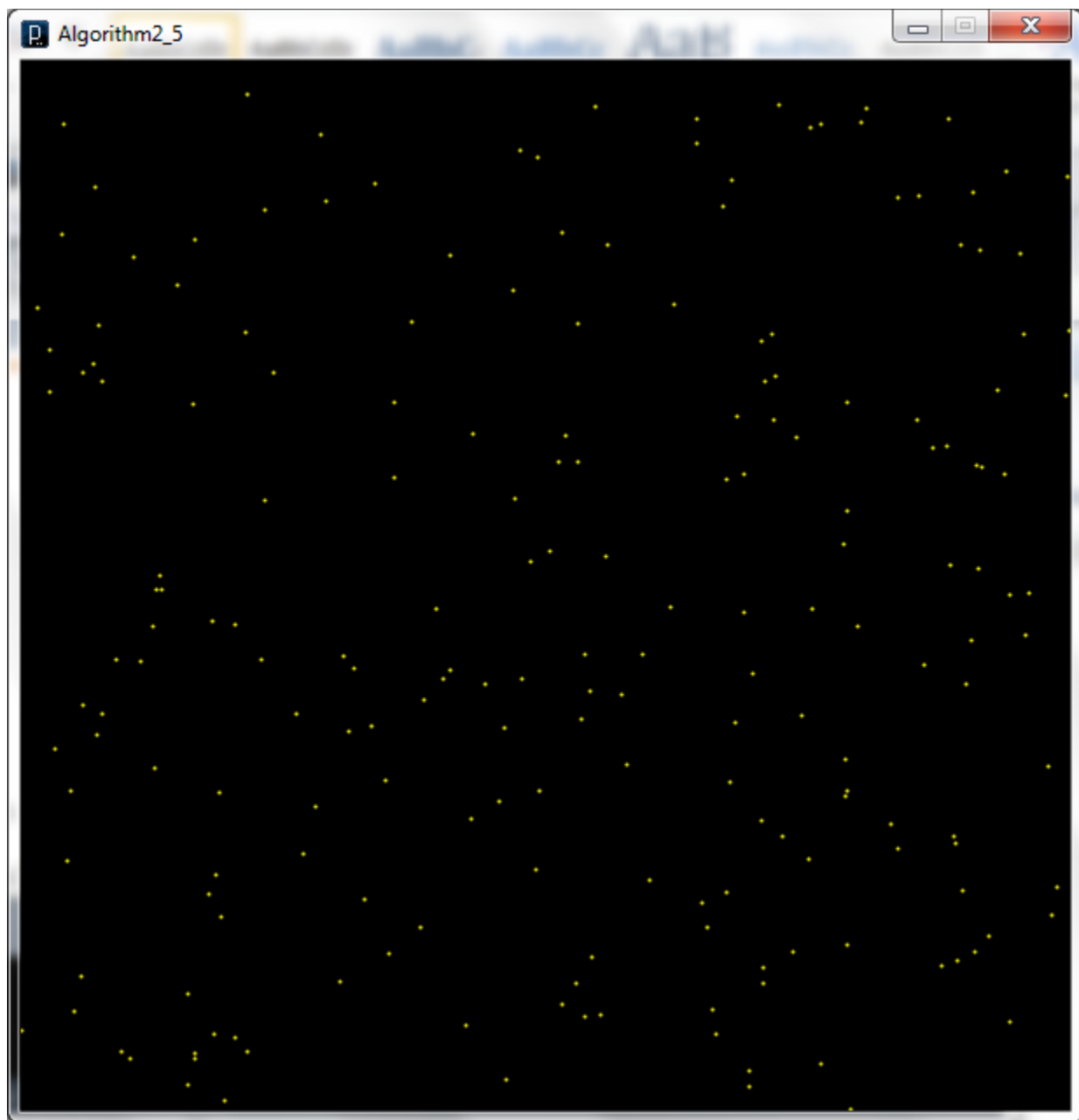
    float xCoordinate,yCoordinate;

}

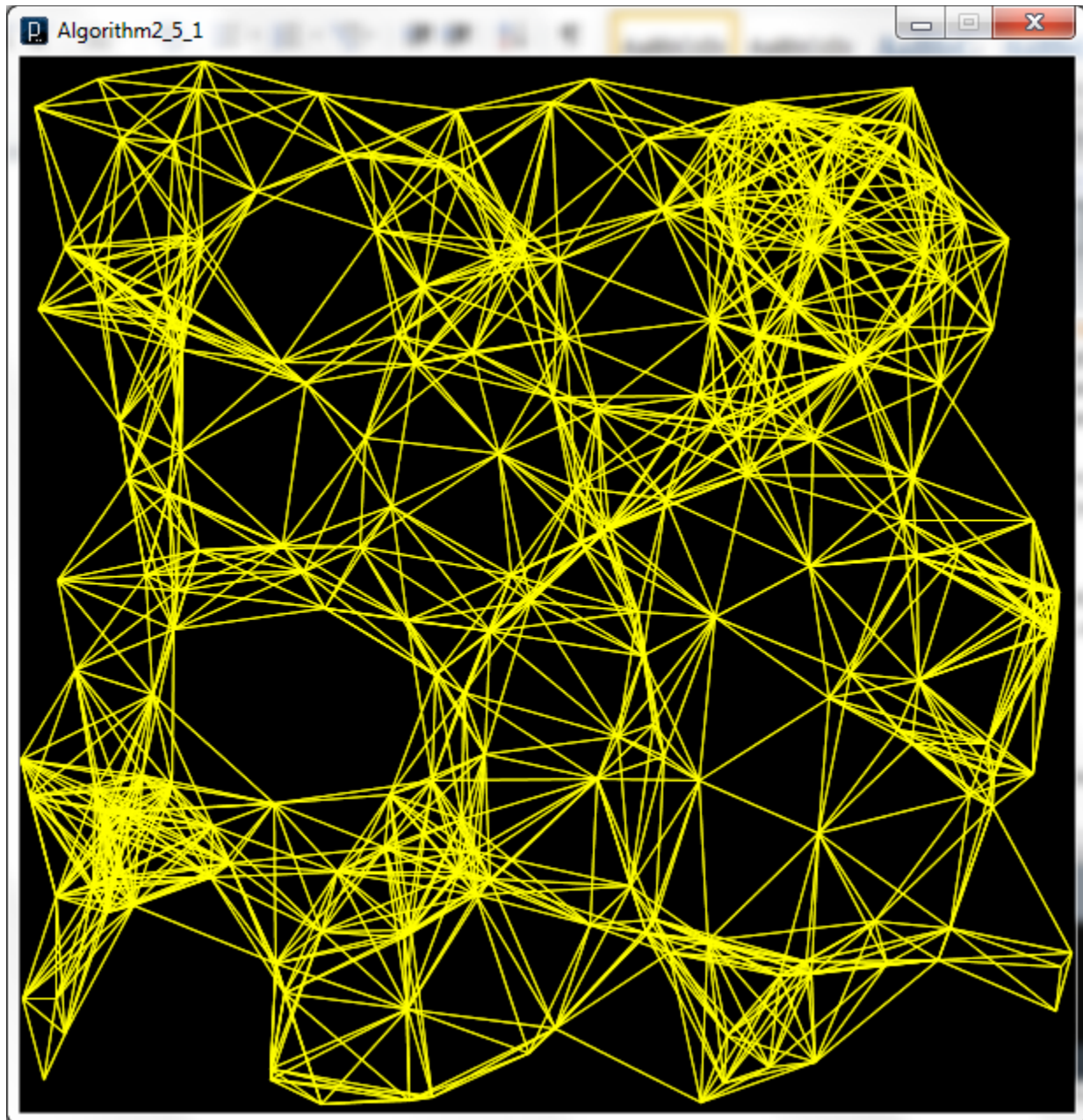
```

The result screen shot is listed below.

First, show the random 200 vertices...screenshot.



Then, show the screenshot after edges linked....



2. From the second question, I won't show the whole code since it is a little long. I will show the important part.

Since this question needs to know the linked vertices' degrees and show two vertices of highest and lowest degrees (I consulted with Professor Matula, he said only need to show one of each, otherwise, we have to use sort algorithm), we need to know which vertices are linked together and their degree. That's why I use adjacency lists to store the 1600 vertices. Each list size is just the degree the according vertex (In real programming, it is the size of list minus 1). Since

Processing is based on Java, it also has ArrayList class which can easily store as a list. And I use an array of ArrayLists to build the adjacency lists. The code is listed below:

```
ArrayList<Point>[] graph=new ArrayList[1600];

for(int i=0;i<1600;i++){

    float xCoordinate=random(600),yCoordinate=random(600);

    point(xCoordinate,yCoordinate);//draw point

    graph[i]=new ArrayList<Point>();

    graph[i].add(new Point(xCoordinate,yCoordinate));

}

for(int i=0;i<1600;i++){

    for(int j=0;j<1600;j++){

        if(i==j)

            continue;

        if(sqrt(sq(graph[i].get(0).xCoordinate-
graph[j].get(0).xCoordinate)+sq(graph[i].get(0).yCoordinate-
graph[j].get(0).yCoordinate))<600*0.06){

            graph[i].add(new Point(graph[j].get(0).xCoordinate,graph[j].get(0).yCoordinate));

        }

    }

}
```

Since it only needs to show one highest degree vertex and one lowest degree vertex, then I only need to find out one maximum degree vertex and one minimum degree vertex. The according code is listed below:

```
int max=graph[0].size(),min=graph[0].size(),maxIndex=0,minIndex=0;

for(int i=0;i<1600;i++){

    if(graph[i].size()>max){

        max=graph[i].size();

        maxIndex=i;

    }

}
```

```

    }
    if(graph[i].size()<min){
        min=graph[i].size();
        minIndex=i;
    }
}

```

Finally, only need to draw edges of all the according adjacency list vertices to the highest degree vertex and to draw edges of all the according adjacency list vertices to the lowest degree vertex. The code is listed below.

```

    for(int i=1;i<graph[maxIndex].size();i++)//draw edges of all the adjacency list vertices of the
    vertex with highest degree

```

```

line(graph[maxIndex].get(0).xCoordinate,graph[maxIndex].get(0).yCoordinate,graph[maxIndex]
.get(i).xCoordinate,graph[maxIndex].get(i).yCoordinate);

```

```

    for(int i=1;i<graph[minIndex].size();i++)//draw edges of all the adjacency list vertices of the
    vertex with lowest degree

```

```

line(graph[minIndex].get(0).xCoordinate,graph[minIndex].get(0).yCoordinate,graph[minIndex].
get(i).xCoordinate,graph[minIndex].get(i).yCoordinate);

```

Now, show the result screenshot:

3. For the third question, first, we need to generate random distributed points on a sphere surface.

For the random distributed characteristics, we can pick four numbers a,b,c, and d from a uniform distribution on (-1,1), and reject pairs with $k \geq 1$ where $k = a^2 + b^2 + c^2 + d^2$.

Then we can transform the points on the sphere surface to x, y, z coordinate with such formulas,

$$x = 2 * (b * d + a * c) / k$$

$$y = 2 * (c * d - a * b) / k$$

$$z = (a^2 + d^2 - b^2 - c^2) / k$$

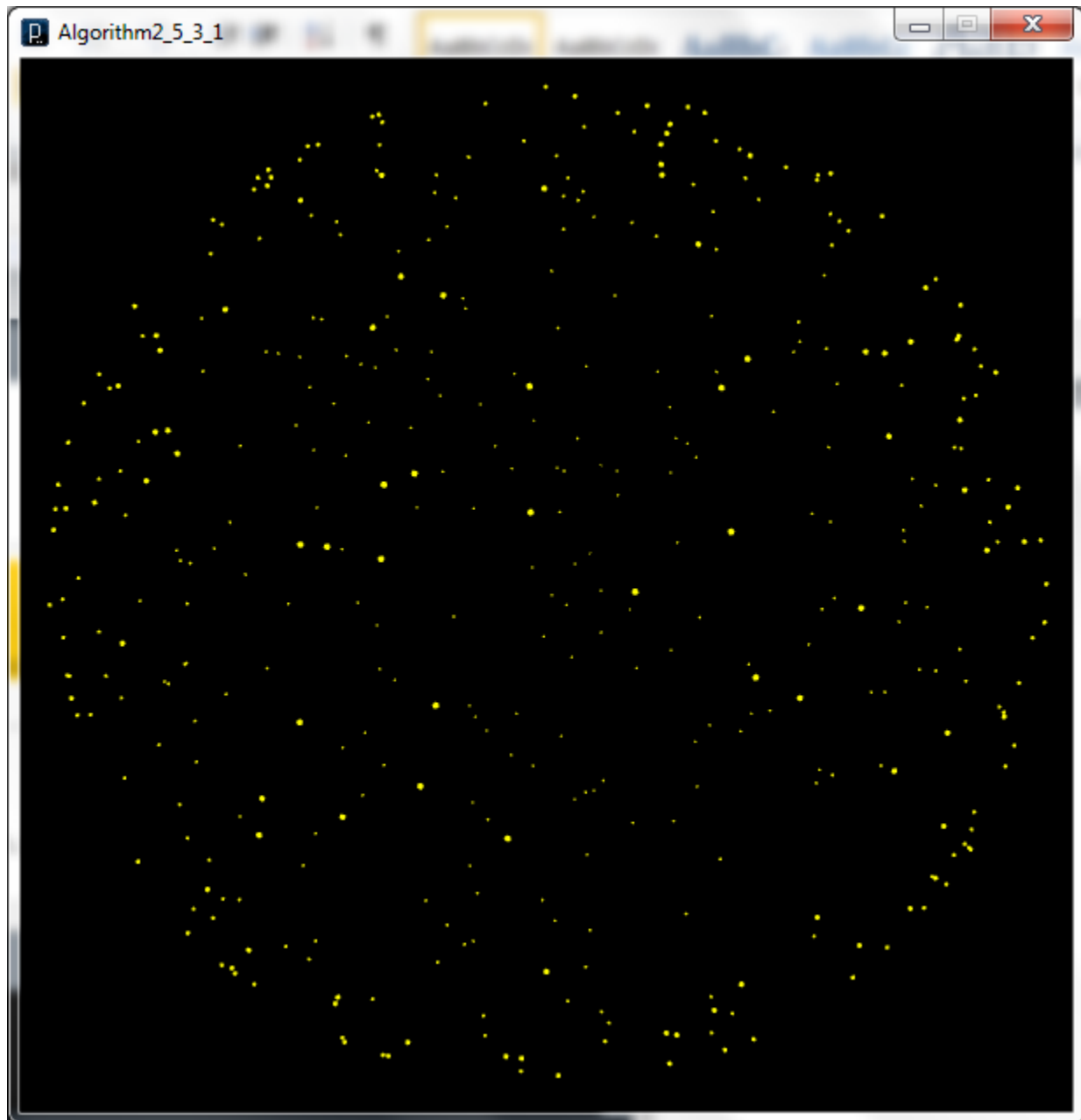
In the programming code, generating one vertex code is listed below:

```
PVector randomSpherePoint(float radius)
{
    float a=0, b=0, c=0, d=0, k=99;
    while (k >= 1.0)
    {
        a = random (-1.0, 1.0);
        b = random (-1.0, 1.0);
        c = random (-1.0, 1.0);
        d = random (-1.0, 1.0);
        k = a*a + b*b + c*c + d*d;
    }
    k = k/radius;
    return new PVector
        (2*(b*d + a*c)/k, 2*(c*d - a*b)/k, (a*a+d*d-b*b-c*c)/k);
}
```

Then call this method 400 times... to draw 400 vertices.

```
void createSpherePoints()
{
    for (int i=0;i<400;i++)
        points[i]=randomSpherePoint(250);
}
```

The result screenshot is listed below:



The calculate the distance among these points. The calculating formula is similar to the one in the 2nd question which is $a(x_a, y_a, z_a)$ and $b(x_b, y_b, z_b)$ then the distance of $ab = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2}$. In programming code is shown as this.

```
for(int i=0;i<400;i++)  
  
    for(int j=0;j<400;j++){  
  
        if(i==j)  
  
            continue;  
  
        if(sqrt(sq(points[i].x-points[j].x)+sq(points[i].y-points[j].y)+sq(points[i].z-  
points[j].z))<250*0.25)  
  
            line(points[i].x,points[i].y,points[i].z,points[j].x,points[j].y,points[j].z);//draw line  
  
    }
```

Then, the result screenshot is as below:

