INSERTSORT(OLD)

```
oid insertinorder(int b, int a[], int n){
    if (n==0){
        return;}
    int i,temp;
    for (i=0;i<=n;i++){
        if(b<a[i]){
            temp=a[i];
            a[i]=b;
            b=temp;
            a[n]=b;
        }
    }
    for (i=0;i<=4;i++){
        printf("%d",a[i]);
    }
    printf("\n");
    return;
}
```

QUICKSORT(OLD)

```
int pivot(int A[],int p, int q){
    int i=p-1;
    int j=p;
    int x=A[q];
    int temp;
    for(j;j<q;j++){
        if (A[j]<=x){
            i++;
            temp=A[i];
            A[i]=A[j];
            A[j]=temp;
        }
    }
    temp=A[i+1];
    A[i+1]=A[q];
    A[q]=temp;
    return i+1;
}
```

HASCYCLE

```
void destroy(struct list * head){
    if (head->size==0){
        printf("-3\n");
        return;
    }
    struct Node *ptr = head->first;
    while (ptr!=NULL){
        ptr=ptr->next;
        free(head->first);
        head->first=ptr;
        head->size --;
    }
    traverse(head);
}
void insertcycle(struct list * head){
    struct Node *ptr = head->first;
    int n;
    scanf("%d",&n);
    int count=1;
    while(count!=n){
        ptr=ptr->next;
        count++;
    }
    struct Node *ptr2 = head->first;
    while (ptr2->next!=NULL){
        ptr2=ptr2->next;
    }
    ptr2->next = ptr;
}
void hascycle(struct list * head){
    if (head->size==0){
    printf("0\n");
    return;
    }
    struct Node *hare = head->first->next;
    struct Node *tortoise = head->first->next->next;
    while (hare->next !=NULL && tortoise->next !=NULL){
        if(hare == tortoise){
            printf("1\n");
            return;
        }
        hare=hare->next;
        tortoise = tortoise->next->next;
    }
    printf("0\n");
```

V-HASHTABLE

```
typedef struct Student{
    char name[9];
    long int id;
}Student;
typedef struct node{
    Student *st;
    struct node *next;
}node;
typedef struct head{
    node *first;
}head;
typedef struct Hashtable{
    int elementCount;
    float loadFactor;
    int insertionTime;
    int queryingTime;
    int length;
    head *ha;
}Hashtable;
int insert(head *h, Student *s){
    node* n=(node*)malloc(sizeof(node));
    node *t;
    int i=0;
    n->st=s;
    n->next=NULL;
    t=h->first;
    if(t==NULL){
        h->first=n;
        return i;
    }
    while(t->next!=NULL){
        i++;
        t=t->next;
    }
    i++;
    t->next=n;
    return i;
}
int sum(char name[]){
    int s=0,i;
    for(i=0;i<8;i++){
        s+=name[i];
    }
    return s;
}
```

```
    /* if (flag==1){
        hare=hare->next;
        int count = 1 ;
        while(hare!=tortoise){
            hare = hare->next;
            count++;
        }
        printf("%d\n",count);
        return;
    }
    else if (flag==0){
        printf("0\n");
        return;
    }*/

    return;
}
void traversegeneric(struct list *head){
    if (head->size==0){
    printf("-2\n");
    return;
    }
    struct Node *hare = head->first->next;
    struct Node *tortoise = head->first->next->next;
    int flag=0;
    while (hare->next !=NULL && tortoise->next !=NULL){
        if(hare == tortoise){
            flag=1;
            break;
        }
        hare=hare->next;
        tortoise = tortoise->next->next;
    }
    if (flag==1){
        struct Node *hare2 = head->first;
        while(hare2!=hare){
            printf("%d\t",hare2->ele);
            hare = hare->next;
            hare2 = hare2->next;
        }
        printf("%d\t",hare2->ele);
        hare2=hare2->next;
        while(hare2!=hare){
            printf("%d\t",hare2->ele);
            hare2 = hare2->next;
        }
        printf("-2\n");
        return;
    }
    else {
        traverse(head);
    }
}
void destroygeneric(struct list *head){
    head->first = NULL;
    head->size=0;
    traverse(head);
}
```

V-SORT Sparse and Dense

```
void SortSparseLists(int **a,int size,int xLo,int xHi,int yLo,int yHi){
    head *b=(head *)malloc(sizeof(head)*(xHi-xLo+1));
    int i,j;
    for(int j=0;j<(xHi-xLo+1);j++){
        b[j].first=NULL;
```

```
int Hashfunction(int in,char name[],long int id){
    if(in==1){
        return ((sum(name)%89)%20);
    }
    else if(in==2){
        return ((sum(name)%105943)%20);
    }
    else if(in==3){
        return ((sum(name)%89)%200);
    }
    else if(in==4){
        return ((sum(name)%105943)%200);
    }
    else if(in==5){
        return ((id%89)%20);
    }
    else if(in==6){
        return ((id%105943)%20);
    }
    else if(in==7){
        return ((id%89)%200);
    }
    else if(in==8){
        return ((id%105943)%200);
    }
}
void readRecords(Student* s,int n, Hashtable *h[]){
    int i,j;
    for(i=0;i<n;i++){
        //printf("fin");
        scanf("%s%ld",s[i].name,&s[i].id);
        for(j=0;j<8;j++){
            h[j]->insertionTime+=insert(&((h[j]->ha)[Hashfunction(j+1,s[i].name,s[i].id)]),&s[i]);
        }
        //printf("%s\t%ld\n",s[i].name,s[i].id);
    }
//printf("fllosdok");

}
Student *find(Hashtable *h[],int in,char n[],long int id){
    head l=(h[in]->ha)[Hashfunction(in+1,n,id)];
    int i=0;
    node *t=l.first;
    while(t!=NULL){
        i++;
        if(strcmp(((t->st)->name),n)==0 && id==(t->st)->id){
            (h[in]->queryingTime)+=i;
            //printf("finish");
            return t->st;
        }
        t=t->next;
    }
}
void readQueries(int k,Hashtable *h[]){
    int i,j;
    Student *s=(Student *)malloc(sizeof(Student)*k);
    for(i=0;i<k;i++){
        scanf("%s%ld",s[i].name,&s[i].id);
        for(int j=0;j<8;j++){
            find(h,j,s[i].name,s[i].id);
        }
    }
}
void findInsertionComplexity(Hashtable *h[]){
        }
        for(i=0;i<size;i++){
                insert(&b[a[i][0]-xLo],a[i][1]);
        }
        i=0;
                for(j=0;j<(xHi-xLo+1);j++){
                        if(b[j].first==NULL){
                                continue;

                        }
                        else{
                                node *t;
                                t=b[j].first;
                                while(t!=NULL){
                                        a[i][0]=xLo+j;
                                        a[i][1]=t->ele;
                                        i++;
                                        t=t->next;
                                }

                        }
                }
        print(a,size);
}
void SortDenseLists(int **a,int size,int xLo,int xHi,int yLo,int yHi){
        head **b =(head**)malloc(sizeof(head*)*(xHi-xLo+1));
        int i,j;
        for(i=0;i<(xHi-xLo+1);i++){
                b[i]=(head*)malloc(sizeof(head)*(yHi-yLo+1));
                for(j=0;j<(yHi-yLo+1);j++){
                        b[i][j].first=NULL;

                        b[i][j].size=0;
                }
        }
        for(int i=0;i<size;i++){
                b[a[i][0]-xLo][a[i][1]-yLo].size++;
        }
        int k=0;
        for(i=0;i<(xHi-xLo+1);i++){
                for(j=0;j<(yHi-yLo+1);j++){
                        while(b[i][j].size){
                                a[k][0]=i+xLo;
                                a[k][1]=j+yLo;
                                b[i][j].size--;
                                k++;
                        }
                }
        }
        print(a,size);
}
```

V- Student Sort

```
int part(Student st[],int lo,int hi,int p){
        swap(st,lo,p);
        int f=lo+1;
        int h=hi;
        while(f<=h){
                while(st[f].marks<=st[lo].marks && f<=hi){
                        f++;
                }
                while(st[h].marks>st[lo].marks && h>=lo){
                        h--;
                }
                if(f<h){
                        swap(st,f,h);
```

```
                for(int j=0;j<8;j++){
                        printf("%d,%d\t",j+1,h[j]->insertionTime);
                }
}
void findQueryComplexity(Hashtable *h[]){
                for(int j=0;j<8;j++){
                        printf("%d,%d\t",j+1,h[j]->queryingTime);
                }
}

int main(){
        Student* records;
        Hashtable* h[8];
        int i,n,j;
        for(i=0;i<8;i++){
                h[i]=(Hashtable *)malloc(sizeof(Hashtable));
                if((i>=0 && i<2)|| (i>=4 &&i<6)){
                        h[i]->length=20;
                        h[i]->insertionTime=0;
                        h[i]->queryingTime=0;
                        h[i]->ha=(head*)malloc(sizeof(head)*(h[i]-
>length));
                        for(j=0;j<(h[i]->length);j++){
                                (h[i]->ha)[j].first=NULL;
                        }
                }
                else{
                        h[i]->length=200;
                        h[i]->insertionTime=0;
                        h[i]->queryingTime=0;
                        h[i]->ha=(head*)malloc(sizeof(head)*(h[i]-
>length));
                        for(j=0;j<(h[i]->length);j++){
                                (h[i]->ha)[j].first=NULL;
                        }
                }
        }
        scanf("%d",&i);
        while(i!=-1){
                if(i==1){
//printf("fin");

                        scanf("%d",&n);

        records=(Student*)malloc(sizeof(Student)*n);
                        readRecords(records,n,h);
                        //printf("finisj");

                }
                if(i==2){
//printf("fin");
                        int k;
                        scanf("%d",&k);
                        readQueries(k,h);
//printf("fin");
                }
                if(i==3){
                        findInsertionComplexity(h);
                }
                if(i==4){
                        findQueryComplexity(h);
                }
                scanf("%d",&i);
        }
}
```

```
            f++;
            h--;
        }
    }
    swap(st,f-1,lo);
    return f-1;
}
void quicksort(Student st[], int m, int lo, int hi ){
    if(m==3){
        ///rintf("sf");
        while(lo<hi){
            int p=part(st,lo,hi,pivot(st,lo,hi));
            //int size1=p-lo;
            //int size2=hi-p;
            if(p-lo<=2){
                if(lo==p || lo==p-1){
                    //return ;
                }
                else if(st[lo].marks>st[p-
1].marks){
                    swap(st,lo,p-1);
                }
                //return ;
            }
            else{
                quicksort(st,m,lo,p-1);
            }
            if(hi-p<=2){
                if(hi==p || hi==p+1){
                    return ;
                }
                else
if(st[hi].marks<st[p+1].marks){
                    swap(st,hi,p+1);
                }
                return ;
            }
            else{
                //quicksort(st,m,p+1,hi);
                lo=p+1;
            }

        }
        return;
    }
    if(lo<hi){
        int p=part(st,lo,hi,pivot(st,lo,hi));
        if(m==1){
            quicksort(st,m,lo,p-1);
            quicksort(st,m,p+1,hi);
        }
        if(m==2){
            if(p-lo<=2){
                if(lo ==p || lo==p-1){
                    //return ;
                }
                else if(st[lo].marks>st[p-
1].marks){
                    swap(st,lo,p-1);
                }
            }
            else{
                quicksort(st,m,lo,p-1);
            }
            if(hi-p<=2){
```

<u>V-HASHTABLE 2</u>

```
typedef struct symbol{
    char name[20];
    char type[20];
}symbol;
typedef struct node{
    symbol *s;
    struct node* next;
}node;
typedef struct head{
    node* first;
}head;
typedef struct HashTable{
    int entries;
    int size;
    float loadFactor;
    int freeSlots;
    int insertionTime;
    int queryingTime;
    head *he;
}HashTable;
HashTable createEmptyHashTable(int s){
    HashTable h;
    h.size=s;
    h.entries=0;
    h.freeSlots=s;
    h.insertionTime=0;
    h.queryingTime=0;
    h.he=(head*)malloc(sizeof(head)*s);
    int i=0;
    for(i=0;i<s;i++){
        ((h.he)[i]).first=NULL;
    }
    return h;
}
int Hashfunction(HashTable h,char key[]){
    int i=0;
    int s=0;
    for(i=0;i<strlen(key);i++){
        s+=key[i];
    }
    int index=((s)%(1<<16))%(h.size);
    return index;
}
HashTable insertlink(head* head, symbol *sy, HashTable h){
    node  *n,*t;
    n=(node*)malloc(sizeof(node));
    n->s=sy;
    n->next=NULL;
    t=head->first;
    if(t==NULL){
        head->first=n;
        h.freeSlots--;
    }
    else{
        int i=1;
        while(t->next!=NULL){
            t=t->next;
            i++;
        }
        t->next=n;
        h.insertionTime+=i;
    }
    return h;
}
```

```
                if(hi==p || hi==p+1){
                    //return ;
                }
                else
if(st[hi].marks<st[p+1].marks){
                    swap(st,hi,p+1);
                }
                //return;
            }
            else{
                quicksort(st,m,p+1,hi);
            }
        }
    }
}
void qs4(Student st[], int lo, int hi ){
    struct stack *s=(struct stack *)malloc(sizeof(struct
stack));
    push(s,lo,hi);
    struct node *e;
    while(top(s)!=NULL){
        e=top(s);
        lo=e->lo;
        hi=e->hi;
        pop(s);
        while(lo<hi){
            int p=part(st,lo,hi,pivot(st,lo,hi));
            //int size1=p-lo;
            //int size2=hi-p;
            if(p-lo<=2){
                if(lo==p || lo==p-1){
                    //return ;
                }
                else if(st[lo].marks>st[p-
1].marks){
                    swap(st,lo,p-1);
                }
                //return ;
            }
            else{
                //quicksort(st,m,lo,p-1);
                push(s,lo,p-1);
            }
            if(hi-p<=2){
                if(hi==p || hi==p+1){
                    //return ;
                }
                else
if(st[hi].marks<st[p+1].marks){
                    swap(st,hi,p+1);
                }
                break ;
            }
            else{
                //quicksort(st,m,p+1,hi);
                lo=p+1;
            }
        }
    }
}
void pa(Student st[], int m, int lo, int hi ){

    if(lo<hi){
        int p=part(st,lo,hi,pivot(st,lo,hi));
  int f=p-lo,s=hi-p;
```

```
HashTable insert(HashTable h,symbol *sy){
    h.entries++;
    h.loadFactor=((float)h.entries)/h.size;
    h=insertlink(&((h.he)[Hashfunction(h,sy->name)]),sy,h);
    return h;
}

HashTable reinsert(HashTable hn,HashTable h){
    int i;
    node *n;
    for(i=0;i<h.size;i++){
        if((h.he)[i].first==NULL){
            continue;
        }
        n=(h.he)[i].first;
        while(n!=NULL){
            hn=insert(hn,n->s);
            n=n->next;
        }
    }
    return hn;
}
void printht(HashTable H){
    printf("%d,\t%d,\t%f,\t%d,\t%d\n",H.entries,H.size,H.loadFact
or,H.freeSlots, H.insertionTime);
}
HashTable createHashTable(int size,float minLoad,float maxLoad,int
resizeFactor, symbol *list,int q){
    HashTable h=createEmptyHashTable(size);
    int i;
    for(i=0;i<q;i++){
        h=insert(h,&list[i]);
        if(h.loadFactor>maxLoad){
            int nsi=h.size*resizeFactor;
            HashTable hn=createEmptyHashTable(nsi);
            hn.insertionTime=h.insertionTime;
            hn=reinsert(hn,h);
            //delete(h);
            h=hn;
        }
        if(h.loadFactor<minLoad){
            int nsi=h.size/resizeFactor;
            HashTable hn=createEmptyHashTable(nsi);
            hn.insertionTime=h.insertionTime;
            hn=reinsert(hn,h);
            //delete(h);
            h=hn;
        }

    }
    printht(h);
    return h;
}
void readSymbols(symbol *list,int n){
    int i;
    for(i=0;i<n;i++){
        scanf("%s%s",list[i].name,list[i].type);
    }
}
HashTable findlink(head* head, symbol *sy, HashTable h){
    node  *n,*t;
    t=head->first;

        int i=0;
        while(t!=NULL){
```

```
    if(f>=s && f<m){
      return ;
    }
    if(s>=f && s<m){
      return ;
    }
                           pa(st,m,lo,p-1);
                           pa(st,m,p+1,hi);
  }
}
```

```
                           if(t->s==sy){
        break;
                           }
                           i++;
                           t=t->next;
                  }
                  h.queryingTime+=i;
         return h;
}

HashTable find(HashTable h,symbol *sy){
         h=findlink(&((h.he)[Hashfunction(h,sy->name)]),sy,h);
         return h;
}

HashTable lookupQueries(HashTable h,symbol *list,int q){
  int i;
  for(i=0;i<q;i++){
    h=find(h,&list[i]);

  }
}
```

N- COUNTSORT OF STRINGS

```
void countSort(char arr[])
{
  // The output character array that will have sorted arr
  char output[strlen(arr)];

  // Create a count array to store count of inidividul
  // characters and initialize count array as 0
  int count[RANGE + 1], i;
  memset(count, 0, sizeof(count));

  // Store count of each character
  for(i = 0; arr[i]; ++i)
  {
    ++count[arr[i]];
    //printf("%d ", i); 12 prints
  }

  // Change count[i] so that count[i] now contains actual
  // position of this character in output array
  for (i = 1; i <= RANGE; ++i)
    count[i] += count[i-1];

  // Build the output character array
  for (i = 0; arr[i]; ++i)
  {
    output[count[arr[i]]-1] = arr[i];
    --count[arr[i]];
  }

  // Copy the output array to arr, so that arr now
  // contains sorted characters
  for (i = 0; arr[i]; ++i)
    arr[i] = output[i];
}
```