```
--------------------------------------------------------------------------------
----------------------------------------
//ARRAYLIST IMPLEMENTATION



#include<iostream>
using namespace std;
class ArrayList {
  private:
    int SIZE;
    int length;
    int pos;
    int * Array;
    int * curr;
  public:
    ArrayList() {
      SIZE=10;
      Array= new int[SIZE];
      length=0;
      pos=0;
      curr= Array;
    }
    ~ArrayList() {
      delete []Array;
      delete curr;
    }
    void printArray() {
      if(length>0) {
        head();
        for(int x=0; x<length; x++)
          cout<<*curr++<<"\t";
      } else cout<<"Array is Empty"<<endl;
    }
    void InsertElement(int val) {
      if(!IsFull()) {
        head();
        curr= curr +length;
        *curr= val;
        length++;
      } else {
        cout<<"Array is Full"<<endl;
      }
    }
    void InsertAtPos(int val, int pos) {
      if (!IsFull())
      if (pos<=length&&pos>0) {
        tail();
        for (int i=length; i>=pos; i-- ) {
          *(curr+1)= *curr;
          back(); //curr= curr-1;
        }
        next();//curr= curr+1;
        *(curr)= val;
        length++;
      } else if (pos>length && pos<=SIZE) {
        head();
        curr= curr+pos-1;
```

```
58          *curr= val;
59          length++;
60        } else
61          cout<<"Invalid Position"<<endl;
62      }
63      void reverseArray() {
64        int *p1, *pn, temp;
65        p1= Array;
66        pn= Array+length-1;
67        int val= length/2;
68        for (int i=0; i<val; i++) {
69          temp= *p1;
70          *p1= *pn;
71          *pn= temp;
72          p1++;
73          pn--;
74        }
75      }
76      void deleteElement(int n) {
77        if (!IsEmpty()) {
78          int *ptr= Array;
79          for (int x=0; x<length; x++) {
80            if(*ptr==n) {
81              int *ptr2= ptr;
82              for (int j=x; j<length; j++) {
83                ptr2++;
84                *ptr= *ptr2;
85                ptr++;
86              }
87              length--;
88              break;
89            }
90            ptr++;
91          }
92        } else cout<<"Array is Empty, Delete operation failed"<<endl;
93      }
94      void deleteElementAtPos(int pos) {
95        if (!IsEmpty()) {
96          if (pos<=SIZE && pos>0){
97            head();  //curr= &Array[0]
98            curr = curr+pos-1;
99            for (int x=0; x<=length-pos; x++){
100             *(curr)= *(curr+1);
101             next(); //curr= curr+1;
102           }
103             length--;
104         }
105       } else cout<<"Array is Empty, Delete operation failed"<<endl;
106     }
107     bool IsFull() {
108       if (length==SIZE)
109         return true;
110       else return false;
111     }
112     bool IsEmpty() {
113       if (length==0)
114         return true;
115       else return false;
```

```
116     }
117     void head() {
118       curr= Array;
119     }
120     void tail() {
121       curr= Array+length-1;
122     }
123
124     void back() {
125       curr= curr-1;
126     }
127     void next() {
128       curr= curr+1;
129     }
130     int Length() {
131       return length;
132     }
133     void emptylist() {
134       head();
135       for (int x=0; x<SIZE; x++) {
136         *curr++=0;
137       }
138     }
139     void sortArray() {
140       int *p1;
141       int *p2, *temp;
142       //sorting - ASCENDING ORDER
143       for(int i=0; i<SIZE; i++) {
144         p1 = Array+i;
145         for(int j=i+1; j<SIZE; j++) {
146           p2 = Array+j;
147           if(*p1>*p2) {
148             *temp  =*p1;
149             *p1=*p2;
150             *p2=*temp;
151           }
152         }
153
154       }
155
156     }
157     //    void reverseArray() {
158 //      if(length>0) {
159 //
160 //        int * temp= Array+length-1;
161 //        int * tempA= new int [length-1];
162 //        int *ptr= tempA;
163 //
164 //        for(int x=0; x<length; x++) {
165 //          *ptr= *temp;
166 //          ptr++;
167 //          temp--;
168 //        }
169 //        ptr = tempA;
170 //        temp= Array;
171 //        for(int x=0; x<length; x++) {
172 //          *temp= *ptr;
173 //          ptr++;
```

```
174 //           temp++;
175 //         }
176 //       }
177 //     }
178 };
179
180 int main () {
181   ArrayList *obj= new ArrayList();
182
183   obj->emptylist();
184   obj->InsertElement(1);
185   obj->InsertElement(2);
186   obj->InsertElement(3);
187   obj->InsertElement(4);
188
189   obj->printArray();cout<<endl;
190   obj->InsertAtPos(99,2);
191
192   obj->printArray();cout<<endl;
193    obj->deleteElementAtPos(2);
194    obj->reverseArray();
195
196   obj->printArray();cout<<endl;
197 //  obj->InsertElement(1);
198 //  obj->InsertElement(2);
199 //  obj->InsertElement(3);
200 //  obj->InsertElement(4);
201 //  obj->InsertElement(5);
202 //  obj->InsertElement(6);
203 //  obj->InsertElement(7);
204 //  obj->InsertAtPos(23,1);
205 //  obj->InsertElement(8);
206 //  obj->InsertElement(9);
207 //  obj->InsertElement(10);
208 //  obj->InsertElement(11);
209 //  obj->InsertElement(12);
210 //  obj->InsertElement(13);
211   //obj->printArray();
212 //  obj->deleteElement(1);
213 //  obj->deleteElement(2);
214   cout<<endl;
215
216   //obj->deleteElementAtPos(4);
217
218   cout<<endl;
219   //obj->emptylist();
220    // obj->reverseArrayAdvanced();
221   cout<<endl;
222   return 0;
223 }
224
225
226 ------------------------------------------------------------------------
    ------------------------------------
227
228 //Single Linklist implementation
229
230
```

```
231
232
233 #include<iostream>
234 using namespace std;
235 class node {
236   public:
237     int data;
238     node *next;
239 };
240
241 node *head= new node();
242 node *curr= new node();
243 int length=0;
244 void GoToHead() { // set curr pointer to head node;
245   curr= head;
246 }
247
248 void insertNodeAtEnd(int val) { // This function will insert new node at the
    end.
249   GoToHead();
250   node *t= new node();
251   while(curr->next!=NULL)
252     curr= curr->next;
253   t->data= val;
254   t->next= NULL;
255   curr->next= t;
256   length++;
257 }
258 void AddNodeBeforeHead( int val) { // This function will insert new node as
    a head.
259   GoToHead();
260   node *t= new node();
261   t->data= val;
262   t->next= curr;
263   head= t;
264   length++;
265 }
266 void InsertAfterSpecificKey(int val, int key ) {
267   node *t= new node();
268   GoToHead();
269   while (curr!=NULL) {
270     if (curr->data==key) {
271       t->data= val;
272       t->next= NULL;
273       t->next= curr->next;
274       curr->next= t;
275       length++;
276       break;
277     }
278     curr= curr->next;
279   }
280 }
281 void InsertBeforeSpecificKey(int val, int key ) {
282   node *ptr=NULL;
283   GoToHead();
284   while (curr!=NULL) {
285     if (curr->data==key) {
286       node *t= new node();
```

```
287       t->data= val;
288       t->next= NULL;
289       t->next= curr;
290       ptr->next= t;
291       length++;
292       break;
293     }
294     ptr= curr;
295     curr= curr->next;
296   }
297 }
298 void printLinklist() {
299   GoToHead();
300   while(curr!=NULL) {
301     cout<<curr->data<<"\t";
302     curr= curr->next;
303   }
304 }
305
306 void DeleteNodeUsingKey(int key) {
307   GoToHead();
308   node *prenode= new node();
309   if(curr->data== key) {
310     head= curr->next;
311     delete curr;
312     length--;
313     return;
314   } else
315     while(curr!=NULL) {
316       if(curr->data==key) {
317         prenode->next= curr->next;
318         delete curr;
319         length--;
320         break;
321       }
322       prenode= curr;
323       curr=curr->next;
324     }
325
326 }
327 void DeleteNodeUsingPos(int pos) {
328   GoToHead();
329   node *prenode= new node();
330   if(pos>length) {
331     cout<<"This Position dosenot exist"<<endl;
332     return;
333   } else if (pos==1 ) { // if we want to delet head node
334     prenode= curr;
335     head= curr->next;
336     delete prenode;
337     length--;
338   } else {
339     for (int x=1; x<pos; x++) {
340       prenode= curr;
341       curr= curr->next;
342     }
343     prenode->next= curr->next;
344     delete curr;
```

```
345     length--;
346 
347   }
348 }
349 
350 void InsertNodeUsingKey(int val, int key, bool isBefore) {
351   if (isBefore)
352     InsertBeforeSpecificKey( val, key);
353   else
354     InsertAfterSpecificKey( val, key);
355 
356 }
357 void InsertNodeUsingPos(int val, int pos, bool isBefore) {
358   GoToHead();
359   if(pos>length) {
360     cout<<"This Position dosenot exist"<<endl;
361     return;
362   } else if (pos==1 && isBefore ) { // if we want to insert before head
363     AddNodeBeforeHead(val);
364   } else {
365     node *prenode= new node();
366     for (int x=1; x<pos; x++) {
367       prenode= curr;
368       curr= curr->next;
369     }
370     if (isBefore) {
371       node *t= new node();
372       t->data= val;
373       t->next= NULL;
374       t->next= curr;
375       prenode->next= t;
376 
377     } else {
378       node *t= new node();
379       t->data= val;
380       t->next= NULL;
381       t->next= curr->next;
382       curr->next= t;
383     }
384   }
385 
386 }
387 int main () {
388   head->data= 1;
389   head->next=NULL;
390 
391   insertNodeAtEnd(2);
392   insertNodeAtEnd(3);
393   insertNodeAtEnd(4);
394   printLinklist();
395   cout<<endl;
396 
397   InsertAfterSpecificKey(99, 2);
398   printLinklist();
399   cout<<endl;
400 
401   DeleteNodeUsingKey(99);
402   printLinklist();
```

```
403   cout<<endl;
404
405   InsertBeforeSpecificKey(99, 2);
406   printLinklist();
407   cout<<endl;
408
409   InsertNodeUsingPos(88,1,true);
410   printLinklist();
411   cout<<endl;
412
413   DeleteNodeUsingPos(1);
414   DeleteNodeUsingPos(2);
415
416   printLinklist();
417   cout<<endl;
418   return 0;
419 }
420
421
422 ------------------------------------------------------------------------------
    ------------------------------------
423 //DOUBLY LINK
424
425
426
427 #include<iostream>
428 using namespace std;
429 class Node{
430     public:
431     int data;
432     Node*next;
433     Node*prev;
434     Node(int s){data=s;
435     next=prev=NULL;}
436 };
437
438 class DLinkList{
439     private:
440     Node*head;
441     int length;
442     public:
443     DLinkList(){head=NULL;
444     length=0;}
445     void insertHead(int valve){Node*t=new Node(valve);
446     if(head==NULL){head=t;
447     return;}
448     t->next=head;
449     head->prev=t;
450     head=t;
451     length++;}
452     // void insertEnd(int valve){//}
453     void insertSpecific(int valve,int pos){
454         if(pos<1||pos>length+1){cout<<"Invalid Position"<<endl;
455         return;}
456         Node*temp=head;
457         Node*p=new Node(valve);
458         if(pos==1){
459             insertHead(valve);}
```

```
460         else{for(int i=1;i<pos;i++){
461         temp=temp->next;}
462         p->next=temp->next;
463         p->prev=temp;
464         temp->next->prev=p;
465         temp->next=p;
466         length++;
467         }}
468     void deletion(int valve){Node*temp;
469     if(valve>length){
470         cout<<"Invalid Pos"<<endl;
471         return;
472     }
473     temp=head;
474     if(valve==1){
475         head=head->next;
476         temp=head;
477     }
478     while(temp->next->data!=valve){
479         temp=temp->next;}
480         temp->next->next->prev=temp;
481         temp->next=temp->next->next;}
482     void print(){bool flag;
483     cout<<"Press 0 to print in Ascending and 1 to print in Descending ";
484     cin>>flag;
485     if(flag==1){
486     Node*curr=head;
487     while(curr!=NULL){
488     cout<<curr->data<<endl;
489     curr=curr->next;}}
490     if(flag==0){Node*curr=head;
491     while(curr->next!=NULL){curr=curr->next;}
492     while(curr!=NULL){
493         cout<<curr->data<<endl;
494         curr=curr->prev;}
495     }}
496 };
497 int main(){DLinkList List1;
498 // List1.insertHead(2);
499 // List1.insertHead(3);
500 // List1.insertHead(9);
501 // List1.insertHead(10);
502 // List1.insertHead(12);
503 List1.insertSpecific(1,1);
504 List1.insertSpecific(2,1);
505 List1.insertSpecific(3,1);
506 List1.insertSpecific(4,1);
507 List1.insertSpecific(5,1);
508 List1.insertSpecific(6,1);
509 List1.print();
510 List1.print();
511 //cout<<endl;
512 //cout<<"To insert at end, give position 1 in the perimeter:"<<endl;
513 List1.deletion(5);
514 List1.print();}
515
516
517
```

```
518
519 ------------------------------------------------------------------------------
    ------------------------------------
520 //CIRCULAR LINK
521
522
523
524
525 #include<iostream>
526 using namespace std;
527 class node{
528     public:
529     int data;
530     node*next;
531     node(int valve){
532         data=valve;
533         next=NULL;
534     }
535 };
536 class circular{
537     public:
538     node*head;
539     int length;
540     circular(){
541         head=NULL;
542         length=0;
543     }
544     void insert(int value){
545         if(head==NULL){
546             node*n=new node(value);
547             head=n;
548             head->next=head;
549             return;
550         }
551         node*n=new node(value);
552         node*temp=head;
553         while(temp->next!=head){
554             temp=temp->next;
555         }
556         n->next=head;
557         head=n;
558         temp->next=head;
559         return;
560     }
561     // void deletion(int )
562     void print(){
563         node*temp;
564
565         temp=head;
566         while(temp->next!=head){
567             cout<<temp->data;
568             temp=temp->next;
569         }
570     }
571     void deletion(){
572         if(head==NULL){
573             cout<<"nothing to delete";
574             return;
```

```
        }
        node*temp=head;
        while(temp->next!=head){
            temp=temp->next;
        }
        head=head->next;
        temp->next=head;
    }
};
int main(){
    circular obj1;
    obj1.insert(5);
    obj1.insert(5);
    obj1.insert(5);
    obj1.insert(5);
    obj1.insert(5);
    obj1.deletion();
    obj1.deletion();
    obj1.deletion();
    // obj1.insert(5);
    obj1.print();
};



-------------------------------------------------------------------------------------
//STACK USING ARRAY



#include<iostream>
using namespace std;
#define SIZE 100
class StackArr{
    private:
    int top;
    public:
    int arr[SIZE];
    StackArr(){
        top = -1;
        int arr[SIZE];
    }
    void pop(){
        if(top==-1){
            cout<<"Stack Underflows";
            return;
        }
        cout<<arr[top]<<endl;
        top--;
    }
    void push(int valve){
        if(top>SIZE){
            cout<<"Stack Overflows";
            return;
        }
        top++;
```

```
632         arr[top]=valve;
633     }
634     void display(){
635         for(int i=top;top>=0;i--){
636             cout<<arr[top]<<endl;
637             top--;
638         }
639     }
640     int peek(){
641         if(top==-1){
642             cout<<"Stack is empty"<<endl;
643             return 0;
644         }
645         return arr[top];
646     }
647     void isEmpty(){
648         if(top==-1){
649             cout<<"Stack is empty"<<endl;
650         }
651         return;
652     }
653 };
654 int main(){
655     StackArr obj1;
656     //obj1.isEmpty();
657     //obj1.display();
658     //obj1.peek();
659     //obj1.push(2);
660     //cout<<obj1.peek();
661     obj1.push(4);
662     obj1.push(7);
663     obj1.push(8);
664     //obj1.display();
665     obj1.pop();
666     obj1.pop();
667     cout<<obj1.peek()<<endl;
668     //obj1.pop();
669     //obj1.pop()
670     obj1.isEmpty();
671     obj1.display();
672 }
673 ------------------------------------------------------------------------
    ------------------------------------
674 //STACK USING LINKLIST
675
676
677
678 #include<iostream>
679 using namespace std;
680 class Node{
681     public:
682     int data;
683     Node*next;
684     Node(int valve){
685         data=valve;
686         next=NULL;
687     }
688 };
```

```
689 class Stack{
690     private:
691     Node*head;
692     int length;
693     public:
694     Stack(){
695         head=NULL;
696         length=0;
697     }
698     void push(int vault){
699         Node*n=new Node(vault);
700             n->next=head;
701             n->data = vault;
702             head=n;
703         }
704     void pop(){
705         Node*temp=head;
706         cout<<head->data<<endl;
707         head=head->next;
708         delete temp;
709     }
710     void peek(){
711         cout<<head->data;
712     }
713     void IsEmpty(){
714         if(head==NULL){
715             cout<<"Empty";
716         }
717         else{
718             cout<<"It is not empty"<<endl;
719         }
720     }
721     void display(){
722         if(head==NULL){
723             cout<<"Stack is empty";
724         }
725         else{
726         Node*temp=head;
727         while(temp!=NULL){
728             cout<<temp->data<<endl;
729             temp=temp->next;
730         }
731     }
732     }
733 };
734 int main(){
735     Stack obj1;
736     //obj1.push(5);
737     //obj1.push(6);
738     obj1.push(9);
739     obj1.push(15);
740     obj1.push(19);
741     obj1.peek();
742     cout<<endl;
743     obj1.IsEmpty();
744     obj1.display();
745     obj1.pop();
746     obj1.pop();
```

```
747     //obj1.pop();
748     //obj1.pop();
749 }
750 ------------------------------------------------------------------------------
    -------------------------------------
751 //QUEUE USING ARRAY
752
753
754
755 #include<iostream>
756 using namespace std;
757 class Queue{
758     private:
759     int *arr;
760     int front;
761     int rear;
762     int size;
763     int noofelements;
764     public:
765     Queue(int s){
766         arr=new int[s];
767         size=s;
768         front=0;
769         rear=-1;
770         noofelements=0;
771     }
772     void enqueue(int val){
773         if(isFull()){
774             cout<<"Queue overflow"<<endl;
775             return;
776         }
777         if(rear==(size-1))
778         rear=0;
779         else
780         rear++;
781         arr[rear]=val;
782         noofelements++;
783     }
784     bool isFull(){
785         if(noofelements==size)
786         return true;
787         else
788         return false;
789     }
790     int dequeue(){
791         if(isEmpty()){
792             cout<<"Queue Underflow"<<endl;
793             return 0;
794         }
795         int val=arr[front];
796         if(front==(size-1))
797         front=0;
798         else
799         front++;
800         noofelements--;
801         return val;
802     }
803     bool isEmpty(){
```

```
804         if(noofelements==0)
805         return true;
806         else
807         return false;
808     }
809     void definition(){
810     }
811 };
812 int main(){
813     Queue obj1(100);
814     obj1.enqueue(4);
815     obj1.enqueue(8);
816     cout<<obj1.dequeue();
817     cout<<obj1.dequeue();
818     //cout<<obj1.dequeue();
819 }
820 ------------------------------------------------------------------------------
    ------------------------------------
821 //QUEUE USING LINK LIST
822
823
824
825
826 #include<iostream>
827 using namespace std;
828 class Node{
829     public:
830     int data;
831     Node*next;
832     Node(int valve){
833         data=valve;
834         next=NULL;
835     }
836 };
837 class QueueL{
838     private:
839     Node*head;
840     Node*front;
841     Node*rear;
842     int length;
843     public:
844     QueueL(){
845         head=NULL;
846         length=0;
847     }
848     void Enqueue(int vault){
849         /*if(isFull()){
850             cout<<"Queue overflows"<<endl;
851             return 0;
852         }*/
853         Node *n=new Node(vault);
854         if(head==NULL){
855             head=n;
856             front=head;
857             rear=head;
858             length++;
859         }
860         else{
```

```
            rear->next=n;
            rear=n;
            length++;
        }
    }
    bool isEmpty(){
        if(head==NULL)
        return true;
        else
        return false;
    }
    void Dequeue(){
        if(isEmpty()){
            cout<<"Queue Underflows";
            return;
        }
        Node*vamp;
        vamp=front;
        front=front->next;
        cout<<vamp->data;
        delete vamp;
    }
};
int main(){
    QueueL obj1;
    obj1.Enqueue(2);
    obj1.Enqueue(4);
    obj1.Dequeue();
    cout<<endl;
    obj1.Dequeue();
    cout<<endl;
}
```

```
-------------------------------------------------------------------------------
    ------------------------------------
```

```
//BST IMPLEMENTATION
```

```
#include<iostream>
#include <bits/stdc++.h>
using namespace std;
class Node{
    public:
    int data;
    Node*left;
    Node*right;
    Node(int data){
        this->data=data;
        left=right=NULL;
    }
};
class BinarySearchTree{
    public:
    Node*root;
```

```
    BinarySearchTree(){
        root=NULL;
    }
    bool searchNode(int num);
    Node*insert(Node*root,int val);
    void remove(Node*root,int val);
    void inOrderTraversal(Node*root);
    void preOrderTraversal(Node*root);
    void postOrderTraversal(Node*root);
    void makeDeletion(Node*&nodePtr);
    int getLeafCount(Node* node);
    void Merging(BinarySearchTree tree);
};
int main(){
    BinarySearchTree tree;
    BinarySearchTree Stree;
    tree.insert(tree.root,10);
    tree.insert(tree.root,8);
    tree.insert(tree.root,6);
    tree.insert(tree.root,9);
    tree.insert(tree.root,15);
    tree.insert(tree.root,14);
    tree.insert(tree.root,20);

    //tree.insert(tree.root,5);
    //tree.insert(tree.root,17);
    //tree.insert(tree.root,25);
    //tree.insert(tree.root,14);
    //tree.insert(tree.root,20);
    //Node*Anroot=tree.root->left->left;
    //tree.makeDeletion(tree.root);

    /*
    Stree.insert(tree.root,11);
    Stree.insert(tree.root,22);
    Stree.insert(tree.root,7);
    Stree.insert(tree.root,25);
    */

    Stree.Merging(tree);
    tree.makeDeletion(tree.root->left->left);
    cout<<"\n In-Order"<<endl;
    cout<<"Left---Root---Right"<<endl;
    tree.inOrderTraversal(tree.root);

    cout<<"\n Pre-Order"<<endl;
    cout<<"Root---Left---Right"<<endl;
    tree.preOrderTraversal(tree.root);

    cout<<"\n Post-Order"<<endl;
    cout<<"Left---Right---Root"<<endl;
    tree.postOrderTraversal(tree.root);
    cout<<"\n\nThe Tree Leaf Count Is: ";
    cout<<tree.getLeafCount(tree.root)<<"\t";
    cout<<endl;
    if(tree.searchNode(29)){
        cout<<"Value Found";
    }
```

```
    else{
        cout<<"Not found";
    }
    return 0;
}
bool BinarySearchTree::searchNode(int num){
  Node *nodePtr = root;
  while (nodePtr)
  {
    if (nodePtr->data == num)
      return true;
    else if (num < nodePtr->data)
      nodePtr = nodePtr->left;
    else
      nodePtr = nodePtr->right;
  }
  return false;
}
Node*BinarySearchTree::insert(Node*r,int val){
    if(r==NULL){
        Node*t=new Node(val);
        if(r==root){
            root=r=t;
        }
        else{
        r=t;}
        return r;
    }
    else if(val==r->data){
        cout<<"Duplicate Data: "<<val<<endl;
    }
    else if(val<r->data){
        r->left=insert(r->left,val);
    }
    else if(val>r->data){
        r->right=insert(r->right,val);
    }
    return r;
}
void BinarySearchTree::inOrderTraversal(Node*r){
    if(r==NULL){
        return;
    }
    inOrderTraversal(r->left);
    cout<<" "<<r->data<<" ->";
    inOrderTraversal(r->right);
}
void BinarySearchTree::preOrderTraversal(Node*r){
    if(r==NULL){
        return;
    }
    cout<<" "<<r->data<<" ->";
    inOrderTraversal(r->left);
    inOrderTraversal(r->right);
}
void BinarySearchTree::postOrderTraversal(Node*r){
    if(r==NULL){
        return;
```

```
1034     }
1035     inOrderTraversal(r->left);
1036     inOrderTraversal(r->right);
1037     cout<<" "<<r->data<<" ->";
1038 }
1039 void BinarySearchTree::makeDeletion(Node*&nodePtr)
1040 {
1041   Node*tempNodePtr;
1042   if (nodePtr == NULL)
1043     cout << "Cannot delete empty node.\n";
1044   else if (nodePtr->right == NULL)
1045   {
1046     tempNodePtr = nodePtr;
1047     nodePtr = nodePtr->left;
1048     delete tempNodePtr;
1049   }
1050     else if (nodePtr->left == NULL)
1051   {
1052     tempNodePtr = nodePtr;
1053     nodePtr = nodePtr->right;
1054     delete tempNodePtr;
1055   }
1056   else
1057   {
1058     tempNodePtr = nodePtr->right;
1059     while (tempNodePtr->left)
1060     tempNodePtr = tempNodePtr->left;
1061     tempNodePtr->left = nodePtr->left;
1062     tempNodePtr = nodePtr;
1063     nodePtr = nodePtr->right;
1064     delete tempNodePtr;
1065   }
1066 }
1067 int BinarySearchTree::getLeafCount(Node* root)
1068 {
1069   if(root == NULL)
1070     return 0;
1071   if(root->left == NULL && root->right == NULL)
1072     return 1;
1073   else
1074     return getLeafCount(root->left)+getLeafCount(root->right);
1075 }
1076 void BinarySearchTree::Merging(BinarySearchTree tree){
1077     if(root==NULL){
1078         return;
1079     }
1080     else{
1081         tree.inOrderTraversal(tree.root->left);
1082         insert(root,root->data);
1083         tree.inOrderTraversal(tree.root->right);}
1084     }
1085
1086
1087 ------------------------------------------------------------------------
    ------------------------------------
1088 //BST ADEEL IMPLEMENTATION
1089
1090
```

```
#include <iostream>
using namespace std;

class IntBinaryTree
{
private:
  struct TreeNode{
    int value;
    TreeNode *left;
    TreeNode *right;
  };
  TreeNode *root;

    // void tree_clear(TreeNode* nodeptr)
    // {
  // if (nodeptr != NULL) {
  //     tree_clear( nodeptr->left );
  //     tree_clear( nodeptr->right );
  //     delete nodeptr;
  // }
    // }
  void tree_clear(TreeNode *&);
  void deleteNode(int, TreeNode *&);
  void makeDeletion(TreeNode *&);
  void displayInOrder(TreeNode *);
public:
        IntBinaryTree()   // Constructor
    { root = NULL; }
  // ~IntBinaryTree() // Destructor
  //  { tree_clear(root); }
        // void tree_clear(TreeNode* nodeptr);
  void insertNode(int);
  bool searchNode(int);
  void remove(int);
  void showNodesInOrder(void)
    { displayInOrder(root); }
};
bool IntBinaryTree::searchNode(int num)
{
  TreeNode *nodePtr = root;

  while (nodePtr)
  {
    if (nodePtr->value == num)
      return true;
    else if (num < nodePtr->value)
      nodePtr = nodePtr->left;
    else
      nodePtr = nodePtr->right;
  }
  return false;
}
void IntBinaryTree::makeDeletion(TreeNode *&nodePtr)
{
  TreeNode *tempNodePtr;  // Temporary pointer, used in
                          // reattaching the left subtree.

  if (nodePtr == NULL)
```

```
1149     cout << "Cannot delete empty node.\n";
1150   else if (nodePtr->right == NULL)
1151   {
1152     tempNodePtr = nodePtr;
1153     nodePtr = nodePtr->left; // Reattach the left child
1154     delete tempNodePtr;
1155   }
1156       else if (nodePtr->left == NULL)
1157   {
1158     tempNodePtr = nodePtr;
1159     nodePtr = nodePtr->right; // Reattach the right child
1160     delete tempNodePtr;
1161   }
1162   // If the node has two children.
1163   else
1164   {
1165     // Move one node the right.
1166     tempNodePtr = nodePtr->right;
1167     // Go to the end left node.
1168     while (tempNodePtr->left)
1169       tempNodePtr = tempNodePtr->left;
1170     // Reattach the left subtree.
1171     tempNodePtr->left = nodePtr->left;
1172     tempNodePtr = nodePtr;
1173     // Reattach the right subtree.
1174     nodePtr = nodePtr->right;
1175     delete tempNodePtr;
1176   }
1177 }
1178
1179
1180
1181 void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr)
1182 {
1183   if (num < nodePtr->value)
1184     deleteNode(num, nodePtr->left);
1185   else if (num > nodePtr->value)
1186     deleteNode(num, nodePtr->right);
1187   else
1188     makeDeletion(nodePtr);
1189 }
1190
1191 void IntBinaryTree::displayInOrder(TreeNode *nodePtr)
1192 {
1193   if (nodePtr)
1194   {
1195     displayInOrder(nodePtr->left);
1196     cout<< nodePtr->value << endl;
1197     displayInOrder(nodePtr->right);
1198   }
1199 }
1200
1201 void IntBinaryTree::insertNode(int num)
1202 {
1203   TreeNode *newNode,  // Pointer to a new node
1204            *nodePtr;  // Pointer to traverse the tree
1205
1206   // Create a new node
```

```
1207   newNode = new TreeNode;
1208   newNode->value = num;
1209   newNode->left = newNode->right = NULL;
1210
1211   if (!root)  // Is the tree empty?
1212     root = newNode;
1213   else
1214   {
1215     nodePtr = root;
1216             while (nodePtr != NULL)
1217     {      if (num < nodePtr->value)
1218       {      if (nodePtr->left)
1219           nodePtr = nodePtr->left;
1220         else
1221         {     nodePtr->left = newNode;
1222           break;
1223         }
1224       }
1225       else if (num > nodePtr->value)
1226       {      if (nodePtr->right)
1227           nodePtr = nodePtr->right;
1228         else
1229         {      nodePtr->right = newNode;
1230           break;
1231         }
1232       }
1233       else
1234       {   cout << "Duplicate value found in tree.\n";
1235                 break;
1236       }
1237     }
1238   }
1239 }
1240
1241
1242 int main()
1243 {
1244   IntBinaryTree tree;
1245
1246   cout << "Inserting nodes.\n";
1247   tree.insertNode(5);
1248   tree.insertNode(8);
1249   tree.insertNode(3);
1250   tree.insertNode(12);
1251   tree.insertNode(9);
1252   if (tree.searchNode(3))
1253     cout << "3 is found in the tree.\n";
1254   else
1255     cout << "3 was not found in the tree.\n";
1256
1257     // IntBinaryTree tree;
1258
1259   // cout << "Inserting nodes. ";
1260   // tree.insertNode(5);
1261   // tree.insertNode(8);
1262   // tree.insertNode(3);
1263   // tree.insertNode(12);
1264   // tree.insertNode(9);
```

```
1265   // cout << "Done.\n";
1266 }
1267
1268 ------------------------------------------------------------------------
     ------------------------------------
1269 // Binary Search Tree Implementation..  //SIR KHURRAM
1270 // @KS.
1271 #include<iostream>
1272 using namespace std;
1273
1274 class Node {
1275     public:
1276     int data;
1277     Node* left;
1278     Node* right;
1279     Node(int data){
1280         this->data=  data;
1281         left= right= NULL;
1282     }
1283 };
1284 class BinarySearchTree{
1285     public:
1286     Node* root;
1287     BinarySearchTree(){
1288         root= NULL;
1289     }
1290
1291     Node* insert( Node* root, int val);
1292     Node* DeleteNodeInBST(Node* root,int data);
1293     Node* inOrderTraversal( Node* root);
1294     Node* preOrderTraversal( Node* root);
1295     Node* postOrderTraversal( Node* root);
1296     Node* merge( Node* r1, Node* r2);
1297     Node* FindMax(Node* root);
1298     int leafCount (Node* root);
1299     int treeHeight(Node *root);
1300 };
1301
1302 int main (){
1303     BinarySearchTree tree1, tree2;
1304
1305       tree1.insert(tree1.root,10);
1306     tree1.insert(tree1.root, 8);
1307     tree1.insert(tree1.root, 6);
1308     tree1.insert(tree1.root, 9);
1309     tree1.insert(tree1.root, 15);
1310     tree1.insert(tree1.root, 14);
1311     tree1.insert(tree1.root, 20);
1312
1313 //    tree.DeleteNodeInBST(tree.root ,9);
1314
1315
1316     cout<<"In Order Print (left--Root--Right)"<<endl;
1317     tree1.inOrderTraversal(tree1.root);
1318
1319     cout<<"\n----------------------"<<endl;
1320     cout<<"Pre Order Print (Root--left--Right)"<<endl;
1321
```

```
1322     tree1.preOrderTraversal(tree1.root);
1323
1324     cout<<"\n----------------------"<<endl;
1325     cout<<"Post Order Print (left--Right--Root)"<<endl;
1326
1327     tree1.postOrderTraversal(tree1.root);
1328     cout<<"\n\nThe total leaf node in tree are: "<<
     tree1.leafCount(tree1.root);
1329
1330     cout<<"\n\nThe height of root node is : "<<
     tree1.treeHeight(tree1.root);
1331
1332     // Merge .
1333
1334     tree2.insert(tree2.root, 7);
1335     tree2.insert(tree2.root, 33);
1336
1337     tree1.merge(tree2.root, tree1.root);
1338     cout<<"\n\nAfter Merging"<<endl;
1339     cout<<"In Order Print (left--Root--Right)"<<endl;
1340
1341     tree1.inOrderTraversal(tree1.root);
1342     cout<<"\n\nThe total leaf node in tree are: "<<
     tree1.leafCount(tree1.root);
1343
1344     cout<<"\n\nThe height of root node is : "<<
     tree1.treeHeight(tree1.root);
1345
1346     return 0;
1347 }
1348
1349 Node* BinarySearchTree::FindMax(Node* r){
1350
1351     while(r->right!=NULL){
1352         r= r->right;
1353     }
1354     return r;
1355
1356 }
1357
1358 Node* BinarySearchTree::insert(Node* r, int val ){
1359
1360  if (r==NULL)
1361     {
1362         Node* t= new Node(val);
1363
1364         if (r==root)
1365         root= r=t;
1366         else
1367         r=t;
1368
1369         return r;
1370     }
1371 //    else if (r->data== val){
1372 //        //cout<<"Duplicate Record  "<<val;
1373 //            return r;
1374 //    }
1375     else if (val < r->data)
```

```
1376         r->left = insert(r->left , val );
1377 
1378     else if (val > r->data)
1379         r->right= insert( r->right,val);
1380 
1381 }
1382 Node * BinarySearchTree::DeleteNodeInBST(Node* root, int data)
1383 {
1384 
1385     if(root==NULL)
1386      return root;
1387     else if(data<root->data)
1388         root->left = DeleteNodeInBST(root->left, data);
1389     else if (data> root->data)
1390         root->right = DeleteNodeInBST(root->right, data);
1391     else
1392     {
1393         //No child
1394         if(root->right == NULL && root->left == NULL)
1395         {
1396             delete root;
1397             root = NULL;
1398             return root;
1399         }
1400         //One child on left
1401         else if(root->right == NULL)
1402         {
1403             Node* temp = root;
1404             root= root->left;
1405             delete temp;
1406         }
1407         //One child on right
1408         else if(root->left == NULL)
1409         {
1410             Node* temp = root;
1411             root= root->right;
1412             delete temp;
1413         }
1414         //two child
1415         else
1416         {
1417             Node* temp = FindMax(root->left);
1418             root->data = temp->data;
1419             root->left = DeleteNodeInBST(root->left, temp->data);
1420         }
1421     }
1422     return root;
1423 }
1424 
1425 
1426 Node * BinarySearchTree::inOrderTraversal( Node* r){
1427      if (r == NULL)
1428         return NULL;
1429     /* first recur on left child */
1430     inOrderTraversal(r->left);
1431     /* then print the data of node */
1432     cout << " "<< r->data << " -> ";
1433     /* now recur on right child */
```

```
1434     inOrderTraversal(r->right);
1435
1436 }
1437
1438 Node* BinarySearchTree::preOrderTraversal( Node* r){
1439      if (r == NULL)
1440         return NULL;
1441
1442     cout << " "<< r->data << " -> ";
1443     preOrderTraversal(r->left);
1444     preOrderTraversal(r->right);
1445 }
1446 Node* BinarySearchTree::postOrderTraversal( Node* r){
1447      if (r == NULL)
1448         return NULL;
1449     postOrderTraversal(r->left);
1450     postOrderTraversal(r->right);
1451     cout << " "<< r->data << " -> ";
1452 }
1453
1454 int BinarySearchTree::leafCount(Node * r){
1455     int static count= 0;
1456     if(r == NULL)
1457         return 0;
1458     else if(r->left == NULL && r->right == NULL)
1459         return 1;
1460
1461     return count + leafCount(r->left) + leafCount(r->right);
1462 }
1463
1464 int BinarySearchTree::treeHeight(Node *root)
1465 {
1466     int static l_height=0;
1467     int static r_height=0;
1468     if (root == NULL)
1469         return -1;
1470     else
1471     {
1472     l_height = treeHeight(root->left);
1473        r_height = treeHeight(root->right);
1474         if (l_height > r_height)
1475             return (l_height + 1);
1476         else
1477             return (r_height + 1);
1478     }
1479 }
1480 // This method will merge tree1 into tree2
1481 Node * BinarySearchTree::merge( Node* r1, Node* r2){
1482      if (r1 == NULL)
1483         return NULL;
1484     /* first recur on left child */
1485     merge(r1->left, r2);
1486
1487     insert(r2, r1->data);
1488     /* now recur on right child */
1489     merge(r1->right, r2);
1490
1491 }
```

```
1492
1493 ------------------------------------------------------------------------
     ------------------------------------
1494 //BST TO AVL
1495
1496
1497
1498 #include<iostream>
1499 using namespace std;
1500 class node{
1501   public:
1502     node *left;
1503     node*right;
1504     int data;
1505     int height;
1506     node(int data)
1507     {
1508       this->data=data;
1509       height=0;
1510       left=right=NULL;
1511     }
1512 };
1513 class AVLtree{
1514   private:
1515   node*root;
1516   void makeEmpty(node* t);
1517   node* insert(int x,node*t);
1518   node* singleleftrotate(node* &C);
1519   node* singlerightrotate(node*&C);
1520
1521   node* doubleleftrightrotate(node* &C);
1522   node* doublerightleftrotate(node* &C);
1523
1524   node*findmin(node*t);
1525   node*findmax(node *t);
1526
1527   node *remove(int x,node*t);
1528   int height(node*t);
1529   int getBalance(node*t);
1530   void inorder(node *t);
1531
1532   public:
1533     AVLtree()
1534     {
1535       root=NULL;
1536     }
1537     void insert(int x){
1538       root=insert(x,root);
1539     }
1540     void remove(int x)
1541     {
1542       root=remove(x,root);
1543     }
1544     void display()
1545     {
1546       inorder(root);
1547       cout<<endl;
1548     }
```

```
1549
1550
1551 };
1552
1553 int main()
1554 {
1555   AVLtree tree;
1556   tree.insert(3);
1557   tree.insert(4);
1558   tree.insert(5);
1559   tree.insert(6);
1560   tree.insert(7);
1561   tree.display();
1562   return 0;
1563 }
1564
1565 node* AVLtree::singleleftrotate(node* &A)
1566 {
1567 node* newRoot = A->right;
1568 A->right = newRoot->left;
1569 newRoot->left = A;
1570 A->height = max(height(A ->left), height(A ->right)) + 1;
1571 newRoot ->height = max(height(newRoot->right), A->height) + 1;
1572 return newRoot;
1573 }
1574
1575 node* AVLtree::singlerightrotate(node* &C)
1576 {
1577 node* newRoot = C->left;
1578 C->left = newRoot->right;
1579 newRoot->right = C;
1580 C->height = max(height(C ->left), height(C ->right)) + 1;
1581 newRoot ->height = max(height(newRoot->left), C->height) + 1;
1582 return newRoot;
1583 }
1584
1585 node* AVLtree::doubleleftrightrotate(node*& t)
1586 {
1587 t->left = singleleftrotate(t->left);
1588 return singlerightrotate(t);
1589 }
1590
1591 node* AVLtree::doublerightleftrotate(node*& t)
1592 {
1593 t->right = singlerightrotate(t->right);
1594 return singleleftrotate(t);
1595 }
1596
1597 void AVLtree::inorder(node *t)
1598 {
1599   if(t==NULL)
1600   return;
1601   inorder(t->left);
1602   cout<<t->data<<" ->";
1603   inorder(t->right);
1604 }
1605 int AVLtree::height(node* t)
1606 {
```

```
1607   return(t==NULL ? -1 : t->height);
1608 }
1609 int AVLtree::getBalance(node*t)
1610 {
1611   if(t==NULL)
1612   return 0;
1613   else
1614   return height(t->left) - height(t->right);
1615 }
1616
1617
1618 node *AVLtree::findmin(node *t)
1619 {
1620   if(t==NULL)
1621   return NULL;
1622   else if(t->left==NULL)
1623     return  t;
1624   else
1625     return findmin(t->left);
1626 }
1627
1628 node *AVLtree::findmax(node *t)
1629 {
1630   if(t==NULL)
1631       return NULL;
1632   else if(t->right==NULL)
1633     return  t;
1634   else
1635     return findmax(t->right);
1636 }
1637 void AVLtree::makeEmpty(node* t) {
1638         if(t == NULL)
1639             return;
1640         makeEmpty(t->left);
1641         makeEmpty(t->right);
1642         delete t;
1643     }
1644
1645 node*  AVLtree::  insert(int x, node* t)
1646     {
1647         if(t == NULL)
1648         {
1649             t = new node (x);
1650         }
1651         else if(x < t->data)
1652         {
1653             t->left = insert(x, t->left);
1654             if(height(t->left) - height(t->right) == 2)
1655             {
1656                 if(x < t->left->data)
1657                     t = singlerightrotate(t);
1658                 else
1659                     t = doubleleftrightrotate(t);
1660             }
1661         }
1662         else if(x > t->data)
1663         {
1664             t->right = insert(x, t->right);
```

```
1665            if(height(t->right) - height(t->left) == 2)
1666            {
1667                if(x > t->right->data)
1668                    t = singleleftrotate(t);
1669                else
1670                    t = doublerightleftrotate(t);
1671            }
1672        }
1673
1674        t->height = max(height(t->left), height(t->right))+1;
1675        return t;
1676    }
1677
1678    node* AVLtree::remove(int x, node* t)
1679    {
1680        node* temp;
1681
1682        // Element not found
1683        if(t == NULL)
1684            return NULL;
1685
1686        // Searching for element
1687        else if(x < t->data)
1688            t->left = remove(x, t->left);
1689        else if(x > t->data)
1690            t->right = remove(x, t->right);
1691
1692        // Element found
1693        // With 2 children
1694        else if(t->left && t->right)
1695        {
1696            temp = findmin(t->right);
1697            t->data = temp->data;
1698            t->right = remove(t->data, t->right);
1699        }
1700        // With one or zero child
1701        else
1702        {
1703            temp = t;
1704            if(t->left == NULL)
1705                t = t->right;
1706            else if(t->right == NULL)
1707                t = t->left;
1708            delete temp;
1709        }
1710        if(t == NULL)
1711            return t;
1712
1713        t->height = max(height(t->left), height(t->right))+1;
1714
1715        // If node is unbalanced
1716        // If left node is deleted, right case
1717        if(height(t->left) - height(t->right) == 2)
1718        {
1719            // right right case
1720            if(height(t->left->left) - height(t->left->right) == 1)
1721                return singleleftrotate(t);
1722            // right left case
```

```
            else
                return doublerightleftrotate(t);
        }
        // If right node is deleted, left case
        else if(height(t->right) - height(t->left) == 2)
        {
            // left left case
            if(height(t->right->right) - height(t->right->left) == 1)
                return singlerightrotate(t);
            // left right case
            else
                return doubleleftrightrotate(t);
        }
        return t;
    }
-------------------------------------------------------------------------------
    ------------------------------------
//HASHIN LINEAR MAHAD



#include<iostream>
#include<string>

using namespace std;

class Students{
    public:
int rollNo;
// string name;

Students(){

}

};

class Hashtable {
Students **arr;
int size;
int count;
public:

Hashtable(int s){
size = s;
count = 0;
arr = new Students*[size];

for(int i =0 ; i<size ; i++)
arr[i] = NULL;
}

int hashin(int n){
 return n%size;
}


void insert(int key){  /// ,string value
```

```
1780     if(count == size){
1781     cout<<"hash is full";
1782     return;
1783     }
1784
1785     int hashindex = hashin(key);
1786     while(arr[hashindex] != NULL){
1787         hashindex = (hashindex +1) %size;
1788     }
1789     arr[hashindex] = new  Students();
1790     arr[hashindex]->rollNo = key;
1791     // arr[hashindex]->name = value;
1792     count++;
1793 }
1794
1795 int search (int key){
1796     if(count == 0){
1797         cout<< "empty";
1798     }
1799     int hashindex = hashin(key);
1800     int temp = hashindex;
1801     while(true){
1802         if(arr[hashindex] == NULL)
1803         hashindex = (hashindex +1)%size;
1804         else if(arr[hashindex]->rollNo != key)
1805         hashindex = (hashindex +1) %size;
1806         else
1807         break;
1808
1809         if(hashindex == temp){
1810             temp = -1;
1811             break;
1812         }
1813   }
1814     if(temp == -1)
1815     cout<< "element not found";
1816
1817      else
1818      cout<<"element found  ["<< arr[hashindex]->rollNo<<"]";
1819
1820
1821 }
1822
1823 void deleteitem(int key){
1824
1825     if(count == 0){
1826     cout<<"hash is empty";
1827     }
1828
1829     int hashindex = hashin(key);
1830     int temp = hashindex;
1831     while(true){
1832        if(arr[hashindex] == NULL)
1833         hashindex = (hashindex +1)%size;
1834         else if(arr[hashindex]->rollNo != key)
1835         hashindex = (hashindex +1) %size;
1836         else
1837         break;
```

```
1838
1839         if(hashindex == temp){
1840             temp = -1;
1841             break;
1842         }
1843
1844     }
1845     if(temp == -1)
1846     cout<<"not found";
1847
1848     else{
1849         delete arr[hashindex];
1850         arr[hashindex] = NULL;
1851     }
1852
1853 }
1854
1855 void displayitem(){
1856
1857     for(int i = 0 ; i<size ; i++){
1858         if(arr[i]!= NULL)
1859         cout<<"Hash table ["<<i<<"] : key  "<<arr[i]->rollNo<<endl; //
     arr[i]->name
1860     }
1861 }
1862
1863
1864
1865 // ~Hashtable(){
1866
1867 //     for(int i = 0 ; i<size ; i++){
1868 //         if(arr[i]!= NULL){
1869 //             cout<<"deleting key"<<arr[i]->rollNo<<"value"
     <<arr[i]->name<<endl;
1870 //             delete arr[i];
1871 //             arr[i] = NULL;
1872 //         }
1873 //     }
1874 // }
1875
1876
1877 };
1878
1879 int main(){
1880
1881     Hashtable mt(25);
1882
1883 mt.insert(652 );
1884 mt.insert(65402 );
1885 mt.insert(65405 );
1886 mt.insert(65403 );
1887 mt.displayitem();
1888 mt.getitem(6542);
1889
1890 return 0;
1891 }
1892 ------------------------------------------------------------------------
     -------------------------------------
```

```
1893 //HASHING SIR KHURRAM
1894
1895
1896
1897 #include<iostream>
1898 #include<list>
1899 using namespace std;
1900 class HashTable{
1901     int capacity;
1902     list<int> *table;
1903     public:
1904     HashTable(int V);
1905     void insertItem(int key, int data);
1906     void deleteItem(int key);
1907     int checkPrime(int n){
1908         int i;
1909         if(n==1 || n==0){
1910             return 0;
1911         }
1912         for(int i=2;i<n/2;i++){
1913             if(n%i==0){
1914                 return 0;
1915             }
1916         }
1917         return 1;
1918     }
1919     int getPrime(int n){
1920         if(n%2==0){
1921             n++;
1922         }
1923         while(!checkPrime(n)){
1924             n+=2;
1925         }
1926         return n;
1927     }
1928     int hashFunction(int key){
1929         return (key%capacity);
1930     }
1931     void displayHash();
1932 };
1933 HashTable::HashTable(int c){
1934     int size=getPrime(c); //OR      int size=c*2
1935     this->capacity=size;
1936     table=new list<int>[capacity];
1937 }
1938 void HashTable::insertItem(int key,int data){
1939     int index=hashFunction(key);
1940     table[index].push_back(data);
1941 }
1942 void HashTable::deleteItem(int key){
1943     int index=hashFunction(key);
1944     list<int>::iterator i;
1945     for(i=table[index].begin();i!=table[index].end();i++){
1946         if(*i==key)
1947         break;
1948     }
1949     if(i!=table[index].end())
1950     table[index].erase(i);
```

```
}
void HashTable::displayHash(){
    for(int i=0;i<capacity;i++){
        cout<<"table[" <<i<<"]";
        for(auto x:table[i])
        cout<<" --> "<<x;
        cout<<endl;
    }
}
int main(){
    int key[]={231,321,212,321,433,262};
    int data[]={123,432,523,43,423,111};
    int size=sizeof(key)/sizeof(key[0]);

    HashTable h(size);
    //HashTable h(12);
    for(int i=0;i<size;i++){
        h.insertItem(key[i],data[i]);
    }
    h.deleteItem(12);
    h.displayHash();

    return 0;
}

-------------------------------------------------------------------------------
-------------------------------------
//DUPLICATION OF NODES USING QUEUE



#include<iostream>
using namespace std;
class Node{
    public:
    int data;
    Node*next;
    Node(int valve){
        data=valve;
        next=NULL;
    }
};
class QueueL{
    private:
    Node*head;
    Node*front;
    Node*rear;
    int length;
    public:
    QueueL(){
        head=NULL;
        length=0;
    }
    int defination(){
        int var1=DequeueLastBackup();
        int var2=DequeueLastBackup();
        int var3=DequeueLastBackup();
        for(int i=0;i<var1;i++){
```

```
2008            Enqueue(var1);
2009        }
2010        for(int i=0;i<var2;i++){
2011            Enqueue(var2);
2012        }
2013        for(int i=0;i<var3;i++){
2014            Enqueue(var3);
2015        }
2016        return 0;
2017    }
2018    bool isFull(){
2019        if(head==NULL){
2020            return true;
2021        }
2022    }
2023    void Enqueue(int vault){
2024        /*if(isFull()){
2025            cout<<"Queue overflows"<<endl;
2026            return 0;
2027        }*/
2028        Node *n=new Node(vault);
2029        if(head==NULL){
2030            head=n;
2031            front=head;
2032            rear=head;
2033        }
2034        else{
2035            rear->next=n;
2036            rear=n;
2037        }
2038    }
2039    bool isEmpty(){
2040        if(front==NULL)
2041        return true;
2042        else
2043        return false;
2044    }
2045    void Dequeue(){
2046        if(isEmpty()){
2047            cout<<"Queue Underflows";
2048            return;
2049        }
2050        Node*vamp;
2051        vamp=front;
2052        front=front->next;
2053        cout<<vamp->data;
2054        delete vamp;
2055    }
2056    int DequeueBackup(){
2057        if(isEmpty()){
2058            cout<<"Queue Underflows";
2059            return 0;
2060        }
2061        Node*vamp;
2062        vamp=front;
2063        front=front->next;
2064        cout<<vamp->data;
2065        return vamp->data;
```

```
2066     }
2067     int DequeueLastBackup(){
2068         if(isEmpty()){
2069             cout<<"Queue Underflows";
2070             return 0;
2071         }
2072         Node*vamp,*vent;
2073         vamp=front;
2074         front=front->next;
2075         cout<<vamp->data;
2076         return vamp->data;
2077     }
2078     void Duplicate(){
2079         int var1=DequeueMana();
2080         int var2=DequeueMana();
2081         int var3=DequeueMana();
2082         for(int i=0;i<var1;i++){
2083             Enqueue(var1);
2084         }
2085         for(int i=0;i<var2;i++){
2086             Enqueue(var2);
2087         }
2088         for(int i=0;i<var3;i++){
2089             Enqueue(var3);
2090         }
2091     }
2092     int DequeueMana(){
2093         if(isEmpty()){
2094             cout<<"Queue Underflows";
2095             return 0;
2096         }
2097         Node*vamp;
2098         int mango;
2099         vamp=front;
2100         front=front->next;
2101         //cout<<vamp->data;
2102         mango=vamp->data;
2103         delete vamp;
2104         return mango;
2105     }
2106 };
2107 int main(){
2108     QueueL obj1;
2109     obj1.Enqueue(3);
2110     obj1.Enqueue(4);
2111     obj1.Enqueue(5);
2112     obj1.Duplicate();
2113     //obj1.Dequeue();
2114     //obj1.Dequeue();
2115     //obj1.defination();
2116     //int var1=obj1.DequeueMana();
2117     //int var2=obj1.DequeueMana();
2118     //obj1.Dequeue();
2119     //obj1.DequeueMana();
2120     //obj1.Dequeue();
2121     //int var1=obj1.DequeueBackup();
2122     //int var2=obj1.DequeueBackup();
2123     //int var3=obj1.DequeueBackup();
```

```
2124     //cout<<var1<<var2;
2125 }
2126 ------------------------------------------------------------------------------
     ------------------------------------
2127 //HEAP MAH
2128
2129
2130
2131 #include<iostream>
2132 #include<climits>
2133 using namespace std;
2134
2135 // Prototype of a utility function to swap two integers
2136 void swap(int *x, int *y);
2137
2138 // A class for Min Heap
2139 class MinHeap
2140 {
2141   int *harr; // pointer to array of elements in heap
2142   int capacity; // maximum possible size of min heap
2143   int heap_size; // Current number of elements in min heap
2144   public:
2145   // Constructor
2146   MinHeap(int capacity);
2147
2148   // to heapify a subtree with the root at given index
2149   void MinHeapify(int );
2150
2151   int parent(int i) { return (i-1)/2; }
2152
2153   // to get index of left child of node at index i
2154   int left(int i) { return (2*i + 1); }
2155
2156   // to get index of right child of node at index i
2157   int right(int i) { return (2*i + 2); }
2158
2159   // to extract the root which is the minimum element
2160   int extractMin();
2161
2162   // Decreases key value of key at index i to new_val
2163   void decreaseKey(int i, int new_val);
2164
2165   // Returns the minimum key (key at root) from min heap
2166   int getMin() { return harr[0]; }
2167
2168   // Deletes a key stored at index i
2169   void deleteKey(int i);
2170
2171   // Inserts a new key 'k'
2172   void insertKey(int k);
2173 };
2174
2175 // Constructor: Builds a heap from a given array a[] of given size
2176 MinHeap::MinHeap(int cap)
2177 {
2178   heap_size = 0;
2179   capacity = cap;
2180   harr = new int[cap];
```

```
2181 }
2182
2183 // Inserts a new key 'k'
2184 void MinHeap::insertKey(int k)
2185 {
2186   if (heap_size == capacity)
2187   {
2188     cout << "\nOverflow: Could not insertKey\n";
2189     return;
2190   }
2191
2192   // First insert the new key at the end
2193   heap_size++;
2194   int i = heap_size - 1;
2195   harr[i] = k;
2196
2197   // Fix the min heap property if it is violated
2198   while (i != 0 && harr[parent(i)] > harr[i])
2199   {
2200   swap(&harr[i], &harr[parent(i)]);
2201   i = parent(i);
2202   }
2203 }
2204
2205 // Decreases value of key at index 'i' to new_val. It is assumed that
2206 // new_val is smaller than harr[i].
2207 void MinHeap::decreaseKey(int i, int new_val)
2208 {
2209   harr[i] = new_val;
2210   while (i != 0 && harr[parent(i)] > harr[i])
2211   {
2212   swap(&harr[i], &harr[parent(i)]);
2213   i = parent(i);
2214   }
2215 }
2216
2217 // Method to remove minimum element (or root) from min heap
2218 int MinHeap::extractMin()
2219 {
2220   if (heap_size <= 0)
2221     return INT_MAX;
2222   if (heap_size == 1)
2223   {
2224     heap_size--;
2225     return harr[0];
2226   }
2227
2228   // Store the minimum value, and remove it from heap
2229   int root = harr[0];
2230   harr[0] = harr[heap_size-1];
2231   heap_size--;
2232   MinHeapify(0);
2233
2234   return root;
2235 }
2236
2237
2238 // This function deletes key at index i. It first reduced value to minus
```

```
2239 // infinite, then calls extractMin()
2240 void MinHeap::deleteKey(int i)
2241 {
2242   decreaseKey(i, INT_MIN);
2243   extractMin();
2244 }
2245
2246 // A recursive method to heapify a subtree with the root at given index
2247 // This method assumes that the subtrees are already heapified
2248 void MinHeap::MinHeapify(int i)
2249 {
2250   int l = left(i);
2251   int r = right(i);
2252   int smallest = i;
2253   if (l < heap_size && harr[l] < harr[i])
2254     smallest = l;
2255   if (r < heap_size && harr[r] < harr[smallest])
2256     smallest = r;
2257   if (smallest != i)
2258   {
2259     swap(&harr[i], &harr[smallest]);
2260     MinHeapify(smallest);
2261   }
2262 }
2263
2264 // A utility function to swap two elements
2265 void swap(int *x, int *y)
2266 {
2267   int temp = *x;
2268   *x = *y;
2269   *y = temp;
2270 }
2271
2272 // Driver program to test above functions
2273 int main()
2274 {
2275   MinHeap h(11);
2276   h.insertKey(3);
2277   h.insertKey(2);
2278   h.deleteKey(1);
2279   h.insertKey(15);
2280   h.insertKey(5);
2281   h.insertKey(4);
2282   h.insertKey(45);
2283   cout << h.extractMin() << " ";
2284   cout << h.getMin() << " ";
2285   h.decreaseKey(2, 1);
2286   cout << h.getMin();
2287   cout << endl;
2288   return 0;
2289 }
2290 ------------------------------------------------------------------------
    ------------------------------------
2291 //HEAP SIR KHURRAM
2292
2293
2294
2295 #include<iostream>
```

```
#include<assert.h>
using namespace std;
class MaxHeap{
    struct Node{
        int key;
        int value;
    };
    private:
    Node*arr;
    int capacity;
    int totalItems;
    void doubleCapacity(){
        if(this->arr==NULL){
            this->arr=new Node[1];
            this->capacity=1;
            return;
        }
        int newCapacity=capacity*2;
        Node*newArr=new Node[newCapacity];
        for(int i=0;i<this->totalItems;i++){
            newArr[i]=this->arr[i];
        }
        if(this->arr!=NULL)
        delete this->arr;
        this->capacity=newCapacity;
        this->arr=newArr;
    }
    void shiftUp(int index){
        if(index<1)
        return;
        int parent=(index-1)/2;
        if(this->arr[index].key>this->arr[parent].key){
            swap(this->arr[index],this->arr[parent]);
            shiftUp(parent);
        }
        return;
    }
    void shiftDown(int index){
        int maxIndex=-1;
        int lChildIndex=index*2+1;
        int rChildIndex=(index*2)+2;
        if(lChildIndex<totalItems){
            if(arr[index].key<arr[lChildIndex].key){
                maxIndex=lChildIndex;
            }
        }
        if(rChildIndex<totalItems){
            int newindex=(maxIndex==-1?index:maxIndex);
            if(arr[newindex].key<arr[rChildIndex].key){
                maxIndex=rChildIndex;
            }
        }
        if(maxIndex==-1)
        return;
        swap(arr[index],arr[maxIndex]);
        shiftDown(maxIndex);
    }
public:
```

```
2354     MaxHeap(){
2355         this->arr=NULL;
2356         this->capacity=0;
2357         this->totalItems=0;
2358     }
2359     MaxHeap(int _capacity){
2360         assert(_capacity>=1);
2361         this->arr=new Node[_capacity];
2362         this->capacity=_capacity;
2363         this->totalItems=0;
2364     }
2365     void insert(int key,int value){
2366         if(this->totalItems==this->capacity){
2367             doubleCapacity();
2368         }
2369         this->arr[totalItems].key=key;
2370         this->arr[totalItems].value=value;
2371         shiftUp(totalItems);
2372         this->totalItems++;
2373     }
2374     void getMax(int & value){
2375         assert(totalItems!=0);
2376         value=this->arr[0].value;
2377     }
2378     void deleteMax(){
2379         assert(totalItems!=0);
2380         swap(arr[0],arr[this->totalItems-1]);
2381         totalItems--;
2382         //shift down
2383         shiftDown(0);
2384     }
2385     bool isEmpty() const
2386     {
2387         return (totalItems==0);
2388     }
2389     void deleteAll(){
2390         if(this->arr!=NULL){
2391             delete[]arr;
2392             arr=NULL;
2393             this->capacity=0;
2394             this->totalItems=0;
2395         }
2396     }
2397     ~MaxHeap(){
2398         deleteAll();
2399     }
2400 };
2401 int main(){
2402     MaxHeap a;
2403     for(int i=1;i<=200;i++)
2404         a.insert(i,i);
2405     a.deleteAll();
2406     for(int i=201;i<=300;i++)
2407     a.insert(i,i);
2408     while(!a.isEmpty()){
2409         int s;
2410         a.getMax(s);
2411         cout<<s<<endl;
```

```
2412         a.deleteMax();
2413     }
2414 }
2415 ------------------------------------------------------------------------
    ------------------------------------
2416 //INFIX TO POSTFIX USING STACK
2417
2418
2419
2420 #include<iostream>
2421
2422 #include<stack>
2423
2424 using namespace std;
2425
2426 bool IsOperator(char);
2427
2428 bool IsOperand(char);
2429
2430 bool eqlOrhigher(char, char);
2431
2432 string convert(string);
2433
2434 int main()
2435
2436 {
2437
2438 string infix_expression, postfix_expression;
2439
2440 int ch;
2441
2442 do
2443
2444 {
2445
2446 cout << "Enter your expression ";
2447
2448 cin >> infix_expression;
2449
2450 postfix_expression = convert(infix_expression);
2451
2452 cout << "The Infix expression is.... "<<endl << infix_expression;
2453
2454 cout<<endl;
2455
2456 cout<<endl;
2457
2458 cout << "The Postfix expression is....."<<endl << postfix_expression;
2459
2460 cout<<endl;
2461
2462 cout<<endl;
2463
2464 cout << "Press 1 to enter new expression and 0 to stop the working ";
2465
2466 cout<<endl;
2467
2468 cin >> ch;
```

```
2469
2470 } while(ch == 1);
2471
2472 return 0;
2473
2474 }
2475
2476 bool IsOperator(char c)
2477
2478 {
2479
2480 if(c == '+' || c == '-' || c == '*' || c == '/' || c == '^' )
2481
2482 return true;
2483
2484 return false;
2485
2486 }
2487
2488 bool IsOperand(char c)
2489 {
2490
2491 if( c >= 'A' && c <= 'Z')
2492
2493 return true;
2494
2495 if (c >= 'a' && c <= 'z')
2496
2497 return true;
2498
2499 if(c >= '0' && c <= '9')
2500
2501 return true;
2502
2503 return false;
2504 }
2505
2506 int precedence(char op)
2507
2508 {
2509
2510 if(op == '+' || op == '-')
2511
2512 return 1;
2513
2514 if (op == '*' || op == '/')
2515
2516 return 2;
2517
2518 if(op == '^')
2519
2520 return 3;
2521
2522 return 0;
2523
2524 }
2525
2526 bool eqlOrhigher (char op1, char op2)
```

```
2527
2528 {
2529
2530 int p1 = precedence(op1);
2531
2532 int p2 = precedence(op2);
2533
2534 if (p1 == p2)
2535
2536 {
2537
2538 if (op1 == '^' )
2539
2540 return false;
2541
2542 return true;
2543
2544 }
2545
2546 return  (p1>p2 ? true : false);
2547
2548 }
2549
2550 string convert(string infix)
2551
2552 {
2553
2554 stack <char> S;
2555
2556 string postfix ="";
2557
2558 char ch;
2559
2560 S.push( '(' );
2561
2562 infix += ')';
2563
2564 for(int i = 0; i<infix.length(); i++)
2565
2566 {
2567
2568 ch = infix[i];
2569
2570 if(ch == ' ')
2571
2572 continue;
2573
2574 else if(ch == '(')
2575
2576 S.push(ch);
2577
2578 else if(IsOperand(ch))
2579
2580 postfix += ch;
2581
2582 else if(IsOperator(ch))
2583
2584 {
```

```
2585
2586 while(!S.empty() && eqlOrhigher(S.top(), ch))
2587
2588 {
2589
2590 postfix += S.top();
2591
2592 S.pop();
2593
2594 }
2595
2596 S.push(ch);
2597
2598 }
2599
2600 else if(ch == ')')
2601
2602 {
2603
2604 while(!S.empty() && S.top() != '(')
2605
2606 {
2607
2608 postfix += S.top();
2609
2610 S.pop();
2611
2612 }
2613
2614 S.pop();
2615
2616 }
2617
2618 }
2619
2620 return postfix;
2621
2622 }
2623 ------------------------------------------------------------------------
    ------------------------------------
2624 //AVL IMPLEMENTATION
2625
2626
2627
2628 #include<iostream>
2629 using namespace std;
2630 class node{
2631   public:
2632     node *left;
2633     node*right;
2634     int data;
2635     int height;
2636     node(int data)
2637     {
2638       this->data=data;
2639       height=0;
2640       left=right=NULL;
2641     }
```

```
2642 };
2643 class AVLtree{
2644   private:
2645   node*root;
2646   void makeEmpty(node* t);
2647   node* insert(int x,node*t);
2648   node* singleleftrotate(node* &C);
2649   node* singlerightrotate(node*&C);
2650
2651   node* doubleleftrightrotate(node* &C);
2652   node* doublerightleftrotate(node* &C);
2653
2654   node*findmin(node*t);
2655   node*findmax(node *t);
2656
2657   node *remove(int x,node*t);
2658   int height(node*t);
2659   int getBalance(node*t);
2660   void inorder(node *t);
2661
2662   public:
2663     AVLtree()
2664     {
2665       root=NULL;
2666     }
2667     void insert(int x){
2668       root=insert(x,root);
2669     }
2670     void remove(int x)
2671     {
2672       root=remove(x,root);
2673     }
2674     void display()
2675     {
2676       inorder(root);
2677       cout<<endl;
2678     }
2679
2680
2681 };
2682
2683 int main()
2684 {
2685   AVLtree tree;
2686   tree.insert(3);
2687   tree.insert(4);
2688   tree.insert(5);
2689   tree.insert(6);
2690   tree.insert(7);
2691   tree.display();
2692   return 0;
2693 }
2694
2695 node* AVLtree::singleleftrotate(node* &A)
2696 {
2697 node* newRoot = A->right;
2698 A->right = newRoot->left;
2699 newRoot->left = A;
```

```
2700 A->height = max(height(A ->left), height(A ->right)) + 1;
2701 newRoot ->height = max(height(newRoot->right), A->height) + 1;
2702 return newRoot;
2703 }
2704
2705 node* AVLtree::singlerightrotate(node* &C)
2706 {
2707 node* newRoot = C->left;
2708 C->left = newRoot->right;
2709 newRoot->right = C;
2710 C->height = max(height(C ->left), height(C ->right)) + 1;
2711 newRoot ->height = max(height(newRoot->left), C->height) + 1;
2712 return newRoot;
2713 }
2714
2715 node* AVLtree::doubleleftrightrotate(node*& t)
2716 {
2717 t->left = singleleftrotate(t->left);
2718 return singlerightrotate(t);
2719 }
2720
2721 node* AVLtree::doublerightleftrotate(node*& t)
2722 {
2723 t->right = singlerightrotate(t->right);
2724 return singleleftrotate(t);
2725 }
2726
2727 void AVLtree::inorder(node *t)
2728 {
2729   if(t==NULL)
2730   return;
2731   inorder(t->left);
2732   cout<<t->data<<" ->";
2733   inorder(t->right);
2734 }
2735 int AVLtree::height(node* t)
2736 {
2737   return(t==NULL ? -1 : t->height);
2738 }
2739 int AVLtree::getBalance(node*t)
2740 {
2741   if(t==NULL)
2742   return 0;
2743   else
2744   return height(t->left) - height(t->right);
2745 }
2746
2747
2748 node *AVLtree::findmin(node *t)
2749 {
2750   if(t==NULL)
2751   return NULL;
2752   else if(t->left==NULL)
2753     return  t;
2754   else
2755     return findmin(t->left);
2756 }
2757
```

```
node *AVLtree::findmax(node *t)
{
  if(t==NULL)
      return NULL;
  else if(t->right==NULL)
    return  t;
  else
    return findmax(t->right);
}
void AVLtree::makeEmpty(node* t) {
        if(t == NULL)
            return;
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }

node*  AVLtree::  insert(int x, node* t)
    {
        if(t == NULL)
        {
            t = new node (x);
        }
        else if(x < t->data)
        {
            t->left = insert(x, t->left);
            if(height(t->left) - height(t->right) == 2)
            {
                if(x < t->left->data)
                    t = singlerightrotate(t);
                else
                    t = doubleleftrightrotate(t);
            }
        }
        else if(x > t->data)
        {
            t->right = insert(x, t->right);
            if(height(t->right) - height(t->left) == 2)
            {
                if(x > t->right->data)
                    t = singleleftrotate(t);
                else
                    t = doublerightleftrotate(t);
            }
        }

        t->height = max(height(t->left), height(t->right))+1;
        return t;
    }

    node* AVLtree::remove(int x, node* t)
    {
        node* temp;

        // Element not found
        if(t == NULL)
            return NULL;

```

```
2816        // Searching for element
2817        else if(x < t->data)
2818            t->left = remove(x, t->left);
2819        else if(x > t->data)
2820            t->right = remove(x, t->right);
2821
2822        // Element found
2823        // With 2 children
2824        else if(t->left && t->right)
2825        {
2826            temp = findmin(t->right);
2827            t->data = temp->data;
2828            t->right = remove(t->data, t->right);
2829        }
2830        // With one or zero child
2831        else
2832        {
2833            temp = t;
2834            if(t->left == NULL)
2835                t = t->right;
2836            else if(t->right == NULL)
2837                t = t->left;
2838            delete temp;
2839        }
2840        if(t == NULL)
2841            return t;
2842
2843        t->height = max(height(t->left), height(t->right))+1;
2844
2845        // If node is unbalanced
2846        // If left node is deleted, right case
2847        if(height(t->left) - height(t->right) == 2)
2848        {
2849            // right right case
2850            if(height(t->left->left) - height(t->left->right) == 1)
2851                return singleleftrotate(t);
2852            // right left case
2853            else
2854                return doublerightleftrotate(t);
2855        }
2856        // If right node is deleted, left case
2857        else if(height(t->right) - height(t->left) == 2)
2858        {
2859            // left left case
2860            if(height(t->right->right) - height(t->right->left) == 1)
2861                return singlerightrotate(t);
2862            // left right case
2863            else
2864                return doubleleftrightrotate(t);
2865        }
2866        return t;
2867    }
2868
2869
```