JAVA IMPLEMENTATION

// Source code example for "A Practical Introduction to Data Structures and Algorithm Analysis"
// by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

// DSutil.java

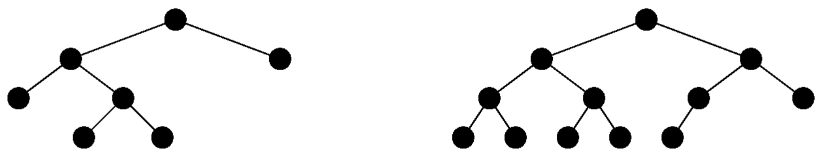
```java
import java.util.*;

// A bunch of utility functions.
public class DSutil {

  // Swap two objects in an array
  public static void swap(Object[] array, int p1, int p2) {
    Object temp = array[p1];
       array[p1] = array[p2];
       array[p2] = temp;
  }

  // Randomly permute the Objects in an array
  static void permute(Object[] A) {
     for (int i = A.length; i > 0; i--)  // for each i
        swap(A, i-1, DSutil.random(i));  //   swap A[i-1] with
  }                                      //   a random element

  // Create a random number function to return values
  // uniformly distributed within the range 0 to n-1.
  static private Random value = new Random();// Random class object
    static int random(int n) { // My own function
          return Math.abs(value.nextInt()) % n;
    }
  }

// Elem.java
// Elem interface.  This is just an Object with
// support for a key field.
interface Elem {             // Interface for generic element type
  public abstract int key(); // Key used for search and ordering
} // interface Elem

// IElem.java
// Sample implementation for Elem interface.
// A record with just an int field.
public class IElem implements Elem {

  private int value;
  public IElem(int v) { value = v; }
  public IElem() {value = 0;}

  public int key() { return value; }
  public void setkey(int v) { value = v; }

  public String toString() { // Override Object.toString
    return Integer.toString(value);
  }
}
```

Binary Trees

A binary tree is made up of a finite set of nodes that is either empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root.

Full binary tree: Each node is either a leaf or internal node with exactly two non-empty children.

Complete binary tree: If the height of the tree is *d*, then all leaves except possibly level *d* are completely full. The bottom level has all nodes to the left side.



(a) (b)

Traversals

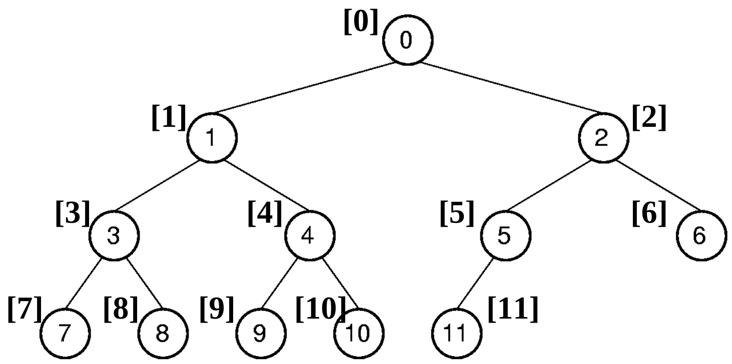
Any process for visiting the nodes in some order is called a traversal.

Any traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes.

•Preorder traversal: Visit each node before visiting its children.
•Postorder traversal: Visit each node after visiting its children.
•Inorder traversal: Visit the left subtree, then the node, then the right subtree.

Complete Binary Tree

Since a complete binary tree is so limited in its shape (there is only one possible shape for n nodes), it is reasonable to expect that space efficiency can be achieved with an array representation.



Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

Parent(r) = $(r - 1)/2$ if $0 < r < n$.
Leftchild(r) = $2r + 1$ if $2r + 1 < n$.
Rightchild(r) = $2r + 2$ if $2r + 2 < n$.
Leftsibling(r) = $r - 1$ if r is even, $r > 0$, and $r < n$.
Rightsibling(r) = $r + 1$ if r is odd and $r + 1 < n$.

```java
// BinNode.java

interface BinNode { // ADT for binary tree nodes

        // Return and set the element value
        public Object element();
        public Object setElement(Object v);
        // Return and set the left child
        public BinNode left();
        public BinNode setLeft(BinNode p);

        // Return and set the right child
        public BinNode right();
        public BinNode setRight(BinNode p);

        // Return true if this is a leaf node
        public boolean isLeaf();

} // end interface BinNode
```

```java
// BinNodePtr.java
public class BinNodePtr implements BinNode{
    private Object element;
    private BinNode left;
    private BinNode right;

    public BinNodePtr()                            // Constructor 1
    { left = right = null; }
    public BinNodePtr(Object val)                  // Constructor 2
    { left = right = null; element  = val; }
    public BinNodePtr(Object val, BinNode l, BinNode r) // Constructor 3
    { left = l; right = r; element = val; }

    public Object element() {return element; }
    public Object setElement(Object v) { return element = v; }

    public BinNode left() { return left; }
    public BinNode setLeft(BinNode p) { return left = p; }

    public BinNode right() { return left; }
    public BinNode setRight(BinNode p) { return right = p; }

    public boolean isLeaf() // Return true if this is a leaf node
    { return ((left == null)&&(right == null)); }

} // end class BinNodePtr
```

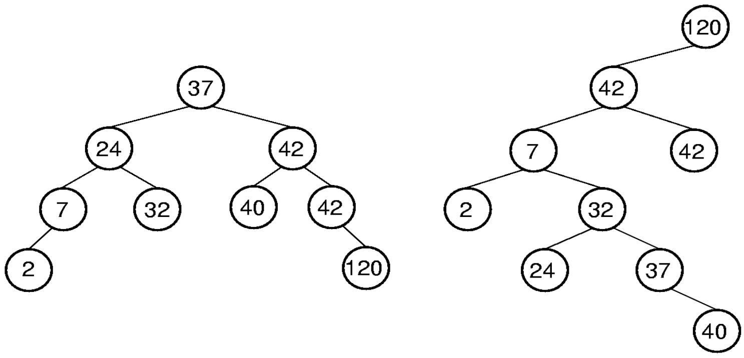
```java
// Main.java
public class Main {

    public static void main(String[] args) {
        // TODO code application logic here
        ……………
    }
    public static void visit(BinNode rt) {
       System.out.println(":"+rt.element());
    }
    public static void preorder(BinNode rt) // rt is root of subtree
    {
     if (rt == null) return; // Empty subtree
     visit(rt);
     preorder(rt.left());
     preorder(rt.right());
    }
    public static void postorder(BinNode rt) // rt is root of subtree
    {
     if (rt == null) return; // Empty subtree
     postorder(rt.left());
     postorder(rt.right());
     visit(rt);
    }

    public static void inorder(BinNode rt) // rt is root of subtree
    {
     if (rt == null) return; // Empty subtree
     inorder(rt.left());
     visit(rt);
     inorder(rt.right());
    }
} // end class Main
```

Binary Search Tree (BST)

Lists have a major problem: Either insert/delete on the one hand, or search on the other, must be O(*n*) time. How can we make both update and search efficient? Answer: Use a new data structure.

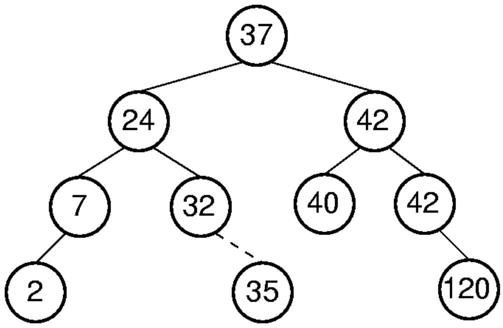
BST Property:

All elements stored in
the left subtree of a node with value *K* have values < *K*.
All elements stored in
the right subtree of a node with value *K* have values >= *K*.



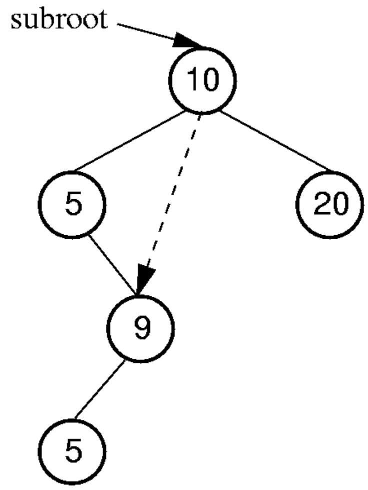

```
public class BST { // BST implementation
        private BinNode root; // The root of the tree

        public BST() { root = null; } // Initialize root

        public void clear() { root = null; }

        public boolean isEmpty() { return root == null; }

        public void print() {
                if (root == null)
                        System.out.println("The BST is empty.");
                else {
                        printhelp(root, 0);
                        System.out.println();
                }
        }

        private void printhelp(BinNode rt, int level) {
            if (rt == null) return;

            printhelp(rt.left(), level+1);

            for (int i = 0; i < level; i++) // Indent based on level
                System.out.print("  ");
            System.out.println(rt.element()); // Print node value

            printhelp(rt.right(), level+1);
        }
```

```
  public Elem find(int key)
  { return findhelp(root, key); }


  private Elem findhelp(BinNode rt, int key) {
          if (rt == null) return null;
          Elem it = (Elem)rt.element();
          if (key < it.key())
            return findhelp(rt.left(), key);
          else
            if (it.key() == key)
                return it;
            else
                return findhelp(rt.right(), key);
  }
```



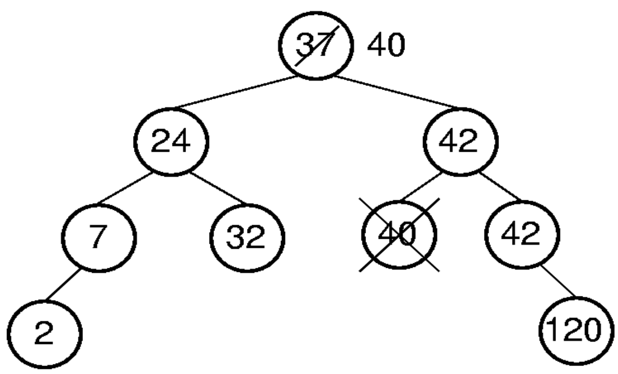

```
  public void insert(Elem val)
  { root = inserthelp(root, val); }

// Convention: Insert duplicates in the right subtree.
// First find where the key “val” would have been if it were in
// the tree: a leaf node or an internal node with no child in the
// appropriate direction. Then add a new node with key “val”.

// The method returns a subtree identical to the old one except
// that it has been modified to contain the new node being inserted

private BinNode inserthelp(BinNode rt, Elem val) {
          if (rt == null) return new BinNodePtr(val);
          Elem it = (Elem) rt.element();
          if (val.key() < it.key())
                  rt.setLeft(inserthelp(rt.left(), val));
          else
                  rt.setRight(inserthelp(rt.right(), val));
          return rt;
          // Only the parent of the added node will have its
          // child pointer modified.
  }
```

```
// Routines to get and remove the node with the smallest key.
// A node with the minimum key value will always be positioned as
// a left leaf of the BST, even in case of keys having duplicate
// values.
private Elem getmin(BinNode rt) {
        if (rt.left() == null)
           return (Elem)rt.element();
        else
           return getmin(rt.left());
}
```




```
// The method returns a subtree identical to the old
// one except that it has been modified deleting a
// node with the minimum key

 // The parent of the node with the minimum key (S)
 // has to change its left child to point to
 // the right child of S.
 private BinNode deletemin(BinNode rt) {
      if (rt.left() == null)
        return rt.right();
      else {
            rt.setLeft(deletemin(rt.left()));
            return rt;
      }
 }
```

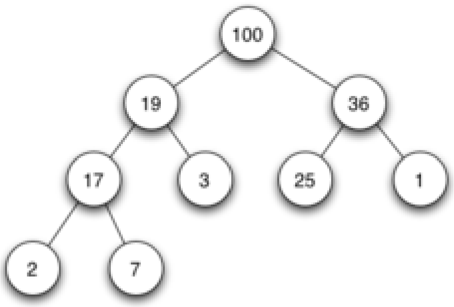
```
// Removing an arbitrary node R from the BST requires that:
// (1) we first find R
// (2) we remove it from the tree taking care of the following cases:
// -- If R has no children then the pointer of Parent(R) is set to NULL
// -- If R has one child then the pointer of Parent(R) is set to R's child
// -- If R has two children:
//        Find a value in one of the two subtree that can replace R
//        preserving the BST property...that is substitute R with
//
//                      the least value of the right subtree
// (in such a way we pick up a value that is less than others on the
// right and is also greater than all nodes on the left of the tree)
// (Preferred if the tree contains duplicates because of the convention
// about the insertion of duplicates)
//                                 OR
//                      the greatest value of the left subtree
```

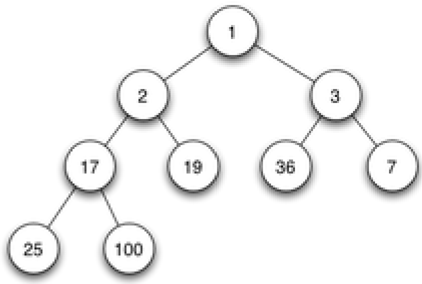


```java
public void remove(int key)
{ root = removehelp(root, key); }


// The method returns a subtree identical to the old
// one except that it has been modified deleting a
// node with the minimum key
private BinNode removehelp(BinNode rt, int key) {
         if (rt == null) return null;
         Elem it = (Elem) rt.element();
         if (key < it.key())
           rt.setLeft(removehelp(rt.left(), key));
         else if (key > it.key())
                rt.setRight(removehelp(rt.right(), key));
              else {
                     if (rt.left() == null)
                       rt = rt.right();
                    // Parent(R)'s link set to the other child of R
                     else if (rt.right() == null)
                            rt = rt.left();
                          else {
                               Elem temp = getmin(rt.right());
                               rt.setElement(temp);
                               rt.setRight(deletemin(rt.right()));
                          }
             }
         return rt;
 }

} // end class BST
```

HEAP

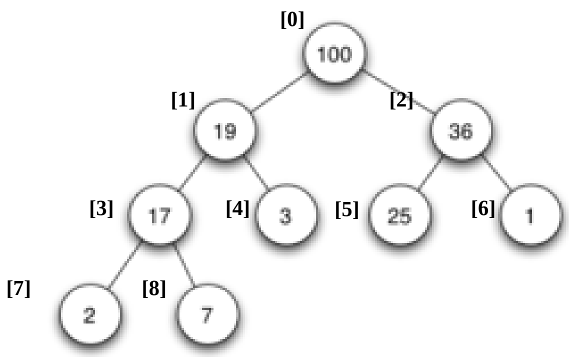
Complete binary tree with the <u>heap property</u>:
• Max-heap: every node store a value that is greater than or equal to the values of either of its children.
• Min-heap: every node store a value that is less than or equal to the values of either of its children.





The values are <u>**partially** ordered</u>.

Heap representation:
normally the array-based complete binary tree representation.



```
public class MaxHeap {
        private Elem[] Heap; // Pointer to heap array
        private int size;    // Maximum size of the heap
        private int n;       // Number of elements in heap

        public MaxHeap(Elem[] h, int num, int max)
        { Heap = h; n = num; size = max; buildheap(); }

        public int heapsize()          // Return size of heap
        { return n; }

        public boolean isLeaf(int pos) // TRUE if pos is leaf
        { return (pos >= n/2) && (pos < n); }

        public int parent(int pos) {   // Return pos for parent
               return (pos-1)/2;
        }

        // Return position for left child of pos
        public int leftchild(int pos) {
               return 2*pos + 1;
        }

        // Return position for right child of pos
        public int rightchild(int pos) {
               return 2*pos + 2;
        }

// If we have a heap, and we add an element, we can perform an operation
// known as sift-up in order to restore the heap property.
// We can do this in O(log n) time, using a binary heap, by following this
// algorithm:
//             (1) Add the element on the bottom level of the heap.
//             (2) Compare the added element with its parent;
//                     if they are in the correct order, stop.
//             (3) If not, swap the element with its parent and return to
//                 the previous step.
// We do this at maximum for each level in the tree – the height of the
// tree, which is O(log n). However, since approximately 50% of the
// elements are leaves and 75% are in the bottom two levels, it is likely
// that the new element to be inserted will only move a few levels upwards
// to maintain the heap. Thus, binary heaps support insertion in average
// constant time, O(1).
```


```
// X = 15

      public void insert(Elem val) {
          int curr = n++;
          Heap[curr] = val;   // Start at the end of the Heap
          // Sift up until curr's parent's key < curr's key
          while (curr!=0 && Heap[curr].key() > Heap[parent(curr)].key()) {
              DSutil.swap(Heap, curr, parent(curr));
              curr = parent(curr);
          }
      }

// Removing the max value in a Max-Heap

// The procedure starts by swapping it with the last element on the last
// level. So, if we have the same max-heap as before, we remove the 11 and
// replace it with the 4

// Now the heap property is violated since 8 is greater than 4.
// The operation that restores the property is called sift-down.
// In this case, swapping the two elements 4 and 8, is enough to restore
// the heap property and we need not swap elements further:

// In general, the wrong node is swapped with its larger child in
// a max-heap (in a min-heap it would be swapped with its smaller child),
// until it satisfies the heap property in its new position.

// Note that the down-heap operation (without the preceding swap) can be
// used in general to modify the value in any position ...siftdown(pos).
```


```
      public Elem removemax() {
         DSutil.swap(Heap,0,--n); // Swap max with the last value
         if (n!=0) // not the last element
           siftdown(0); // put a new heap root value in the correct place
      }
```

```
        private void siftdown(int pos) { // Put in place

                while (!isLeaf(pos)) {
                  int j = leftchild(pos); // rightchild(pos) == j+1 holds

                  if ( j < n-1 &&  Heap[j].key() < Heap[j+1].key() )
                    j++; // j now index of child with greater value

                  if (Heap[pos].key() >= Heap[j].key())
                    return;
                  DSutil.swap(Heap, pos, j);
                  pos = j;                                    // Move down
                }
         }


// Building a Heap.
// This procedure makes a heap out of an array.
// In other words, it rearranges elements of the array so the array
// satisfies the heap property.
// It works by heapifying the elements starting from the middle of the
// array (non leaf-nodes). The runtime of this algorithm is O(n) on an
// array-based heap implementation, where n is the number of nodes in the
// heap.

        public void buildheap() // Heapify contents of Heap
        { for (int i=n/2-1; i>=0; i--) siftdown(i); }



         // Remove value at specified position;

         public Elem remove(int pos) {
                 DSutil.swap(Heap, pos, --n); // Swap with last value

                 while (Heap[pos].key() > Heap[parent(pos)].key()) {
                      DSutil.swap(Heap, pos, parent(pos)); // sift up
                      pos = parent(pos);
                 }

                 if (n != 0) siftdown(pos); // push down
                 return Heap[n];
         }


} // end class MaxHeap
```

C++ IMPLEMENTATION

```
// From the software distribution accompanying the textbook
// "A Practical Introduction to Data Structures and Algorithm Analysis,
// Third Edition" by Clifford A. Shaffer, Prentice Hall, 2007.
// Source code Copyright (C) 2006 by Clifford A. Shaffer.

// File: Book.h
#ifndef _____BOOK_H_____
#define _____BOOK_H_____
#include <time.h>  // Used by timing functions
#include <iostream>
#include <stdlib.h>
using namespace std;

namespace sorting {

// A collection of various macros, constants, and small functions
// used for the software examples.

inline bool EVEN(int x) { return (x % 2) == 0; } // Return true iff x is even
inline bool ODD(int x) { return (x & 1) != 0; } // Return true iff x is odd

// Swap two elements in a generic array
template<class Elem>
inline void swap(Elem A[], int i, int j) {
  Elem temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
// Random number generator functions
inline void Randomize() { srand((unsigned)time( NULL )); } // Seed the generator
inline int Random(int n) { return rand() % (n); } // Return a value in [0,n-1]

#define THRESHOLD 9
template<class Elem> void print_array(Elem a[], int n) {
    int k;
    cout << "\n[ ";
    for (k= 0; k < n-1; k++)
          cout << a[k] << ", ";
    cout << a[k] << " ]\n";
}

template<class Elem> void copy_array(Elem dest[], Elem source[], int n) {
    for (int k= 0; k < n; dest[k] = source[k], k++) ;
}

class Int { // Your basic int type as an object.
private:
  int val;
public:
  Int(int input=0) { val = input; }
  // The following is for those times when we actually
  //   need to get a value, rather than compare objects.
  int key() const { return val; }
  // Overload = to support Int foo = 5 syntax
  Int operator= (int input) { val = input; }
};
} #endif
```

```
// File: Compare.h
#ifndef _____COMPARE_H_____
#define _____COMPARE_H_____
#include <string.h>
namespace sorting {   //Some definitions for Comparator classes

class getintKey { // Get the key from an int
public:
  static int key(int x) { return x; }
};

class getIntKey { // Get the key from an Int object
public:
  static int key(Int x) { return x.key(); }
};

class getIntsKey { // Get the key from a pointer to an Int object
public:
  static int key(Int* x) { return x->key(); }
};

class IntIntCompare {
public:
  static bool lt(Int x, Int y) { return x.key() < y.key(); }
  static bool eq(Int x, Int y) { return x.key() == y.key(); }
  static bool gt(Int x, Int y) { return x.key() > y.key(); }
};

class IntsIntsCompare {
public:
  static bool lt(Int* x, Int* y) { return x->key() < y->key(); }
  static bool eq(Int* x, Int* y) { return x->key() == y->key(); }
  static bool gt(Int* x, Int* y) { return x->key() > y->key(); }
};

class intintCompare {
public:
  static bool lt(int x, int y) { return x < y; }
  static bool eq(int x, int y) { return x == y; }
  static bool gt(int x, int y) { return x > y; }
};

class CCCompare { // Compare two character strings
public:
  static bool lt(char* x, char* y)
    { return strcmp(x, y) < 0; }
  static bool eq(char* x, char* y)
    { return strcmp(x, y) == 0; }
  static bool gt(char* x, char* y)
    { return strcmp(x, y) > 0; }
};

// Get the key for a character string, the key is just the string itself
class getCKey {
public:
  static char* key(char* x) { return x; }
};
} #endif
```

```cpp
// File ADT_Btnode.h
#ifndef __ADT_BTnode_HEADER__
#define __ADT_BTnode_HEADER__

// Binary tree node abstract class
template <class Elem> class BinNode {
public:
  // Return the node's value
  virtual Elem& val() = 0;

  // Set the node's value
  virtual void setVal(const Elem&) = 0;

  // Return the node's left child
  virtual BinNode* left() const = 0;

  // Set the node's left child
  virtual void setLeft(BinNode*) = 0;

  // Return the node's right child
  virtual BinNode* right() const = 0;

  // Set the node's right child
  virtual void setRight(BinNode*) = 0;

  // Return true if the node is a leaf, false otherwise
  virtual bool isLeaf() = 0;
};
#endif
```

```cpp
// File Binnode.h
#ifndef __BinNode_HEADER__
#define __BinNode_HEADER__

#include "ADT_BTnode.h"

// Simple binary tree node implementation
template <class Elem>
class BinNodePtr : public BinNode<Elem> {
private:
  Elem it;                      // The node's value
  BinNodePtr* lc;               // Pointer to left child
  BinNodePtr* rc;               // Pointer to right child

public:
  // Two constructors -- with and without initial values
  BinNodePtr() { lc = rc = NULL; }
  BinNodePtr(Elem e, BinNodePtr* l =NULL, BinNodePtr* r =NULL)
  { it = e; lc = l; rc = r; }
  ~BinNodePtr() {}              // Destructor

  // Functions to set and return the value
  Elem& val() { return it; }
  void setVal(const Elem& e) { it = e; }

  // Functions to set and return the children
  BinNode<Elem>* left() const { return lc; }
  void setLeft(BinNode<Elem>* b) { lc = (BinNodePtr*)b; }
  BinNode<Elem>* right() const { return rc; }
  void setRight(BinNode<Elem>* b) { rc = (BinNodePtr*)b; }

  // Return true if its a leaf, false otherwise
  bool isLeaf() { return (lc == NULL) && (rc == NULL); }
};

#endif
```

```
#ifndef __Traversal_HEADER__
#define __Traversal_HEADER__

// File Traversal.h
#include "book.h"
#include "binnode.h"

template <class Elem>
void preorder(BinNode<Elem>* subroot) {
  if (subroot == NULL) return;  // Empty
  visit(subroot);  // Perform some action
  preorder(subroot->left());
  preorder(subroot->right());
}


// Postorder traversal:
// Visit each node after visiting its children.

template <class Elem>
void postorder(BinNode<Elem>* subroot) {
  if (subroot == NULL) return;  // Empty
  postorder(subroot->left());
  postorder(subroot->right());
  visit(subroot);  // Perform some action
}

// Inorder traversal:
// Visit the left subtree, then the node, then the right subtree.

template <class Elem>
void inorder(BinNode<Elem>* subroot) {
  if (subroot == NULL) return;  // Empty
  inorder(subroot->left());
  visit(subroot);  // Perform some action
  inorder(subroot->right());
}

// Count the number of nodes in a binary tree
template <class Elem>
int count(BinNode<Elem>* root) {
  if (root == NULL) return 0;  // Nothing to count
  return 1 + count(root->left())
           + count(root->right());
}

#endif
```

```cpp
//File ADT_dictionary.h
#ifndef __ADT_dictionary_HEADER__
#define __ADT_dictionary_HEADER__

// The Dictionary abstract class.
// Class Compare compares two keys.
// Class getKey gets a key from an element.
template <class Key, class Elem, class Comp, class getKey>
class  Dictionary {
public:
  // Reinitialize dictionary
  virtual void clear() = 0;

  // Insert an element.  Return true if insert is
  // successful, false otherwise.
  virtual bool insert(const Elem&) = 0;

  // Remove some element matching Key.  Return true if such
  // exists, false otherwise.  If multiple entries match
  // Key, an arbitrary one is removed.
  virtual bool remove(const Key&, Elem&) = 0;

  // Remove and return an arbitrary element from dictionary.
  // Return true if some element is found, false otherwise.
  virtual bool removeAny(Elem&) = 0;

  // Return a copy of some element matching Key.  Return
  // true if such exists, false otherwise.  If multiple
  // elements match Key, return an arbitrary one.
  virtual bool find(const Key&, Elem&) const = 0;

  // Return the number of elements in the dictionary.
  virtual int size() = 0;
};


#endif
```

```
#ifndef __BST_HEADER__
#define __BST_HEADER__

// File BST.h
// This file includes all of the pieces of the BST implementation

// BST Property: All elements stored in the left subtree
// of a node with value K have values < K.
// All elements stored in the right subtree of a node
// with value K have values >= K.


// Include the node implementation
#include "BinNode.h"

// Include the dictionary ADT
#include "ADT_dictionary.h"

// Binary Search Tree implementation for the Dictionary ADT
template <class Key, class Elem, class Comp, class getKey>
class BST : public Dictionary<Key, Elem, Comp, getKey> {
private:
  BinNode<Elem>* root;   // Root of the BST
  int nodecount;         // Number of nodes in the BST

  // Private "helper" functions
  void clearhelp(BinNode<Elem>*);
  BinNode<Elem>* inserthelp(BinNode<Elem>*, const Elem&);
  BinNode<Elem>* deletemin(BinNode<Elem>*, BinNode<Elem>*&);
  BinNode<Elem>* removehelp(BinNode<Elem>*, const Key&, BinNode<Elem>*&);
  bool findhelp(BinNode<Elem>*, const Key&, Elem&) const;
  void printhelp(BinNode<Elem>*, int) const;

public:
   BST() { root = NULL; nodecount = 0; }  // Constructor
  ~BST() { clearhelp(root); }            // Destructor

  void clear()
  { clearhelp(root); root = NULL; nodecount = 0; }

  bool insert(const Elem& it) {
    root = inserthelp(root, it);
    nodecount++;
    return true;
  }

  bool remove(const Key& K, Elem& it) {
    BinNode<Elem>* t = NULL;
    root = removehelp(root, K, t);
    if (t == NULL) return false;  // Nothing found
    it = t->val();
    nodecount--;
    delete t;
    return true;
  }
```

```
  bool removeAny(Elem& it) {        // Delete min value
    if (root == NULL) return false; // Empty tree
    BinNode<Elem>* t;
    root = deletemin(root, t);
    it = t->val();
    delete t;
    nodecount--;
    return true;
  }

  bool find(const Key& K, Elem& it) const
    { return findhelp(root, K, it); }

  int size() { return nodecount; }

  void print() const {
    if (root == NULL) cout << "The BST is empty.\n";
    else printhelp(root, 0);
  }
};

// Clean up BST by releasing space back free store
template <class Key, class Elem, class Comp, class getKey>
void BST<Key, Elem, Comp, getKey>::clearhelp(BinNode<Elem>* root) {
  if (root == NULL) return;
  clearhelp(root->left());
  clearhelp(root->right());
  delete root;
}
// Insert a node into the BST, returning the updated tree
template <class Key, class Elem, class Comp, class getKey>
BinNode<Elem>* BST<Key, Elem, Comp, getKey>::
inserthelp(BinNode<Elem>* root, const Elem& it) {
  if (root == NULL)                     // Empty tree: create node
    return (new BinNodePtr<Elem>(it, NULL, NULL));
  if (Comp::lt(getKey::key(it), getKey::key(root->val())))
    root->setLeft(inserthelp(root->left(), it));
  else root->setRight(inserthelp(root->right(), it));
  return root;                         // Return tree with node inserted
}

// Delete the minimum value from the BST, returning the revised BST
template <class Key, class Elem, class Comp, class getKey>
BinNode<Elem>* BST<Key, Elem, Comp, getKey>::
deletemin(BinNode<Elem>* root, BinNode<Elem>*& S) {
  if (root->left() == NULL) {                         // Found min
    S = root;
    return root->right();
  }
  else {                                              // Continue left
    root->setLeft(deletemin(root->left(), S));
    return root;
  }
}
```

```
// Return in R the element (if any) with value K.
// Return the updated subtree with R removed from the tree.
template <class Key, class Elem, class Comp, class getKey>
BinNode<Elem>* BST< Key, Elem, Comp, getKey >::
removehelp(BinNode<Elem>* root, const Key& K,
           BinNode<Elem>*& R) {
  if (root == NULL) return NULL;                // Val is not in tree
  else if (Comp::lt(K, getKey::key(root->val())))
    root->setLeft(removehelp(root->left(), K, R));
  else if (Comp::gt(K, getKey::key(root->val())))
    root->setRight(removehelp(root->right(), K, R));
  else {                               // Found: remove it
    BinNode<Elem>* temp;
    R = root;
    if (root->left() == NULL)                   // Only a right child
      root = root->right();                     // so point to right
    else if (root->right() == NULL)             // Only a left child
      root = root->left();                      //  so point to left
    else {                                  // Both children are non-empty
      root->setRight(deletemin(root->right(), temp));
      Elem te = root->val();
      root->setVal(temp->val());
      temp->setVal(te);
      R = temp;
    }
  }
  return root;
}

// Find a node with the given key value
template <class Key, class Elem, class Comp, class getKey>
bool BST<Key, Elem, Comp, getKey>:: findhelp(
      BinNode<Elem>* root, const Key& K, Elem& e) const {
  if (root == NULL) return false;           // Empty tree
  else if (Comp::lt(K, getKey::key(root->val())))
    return findhelp(root->left(), K, e);    // Check left
  else if (Comp::gt(K, getKey::key(root->val())))
    return findhelp(root->right(), K, e);  // Check right
  else { e = root->val();  return true; }  // Found it
}

// Print out a BST
template <class Key, class Elem, class Comp, class getKey>
void BST<Key, Elem, Comp, getKey>::
printhelp(BinNode<Elem>* root, int level) const {
  if (root == NULL) return;            // Empty tree
  printhelp(root->left(), level+1);    // Do left subtree
  for (int i=0; i<level; i++)          // Indent to level
    cout << "  ";
  cout << root->val() << "\n";         // Print node value
  printhelp(root->right(), level+1);  // Do right subtree
}



#ifndef __Heap_HEADER__
```

```
#define __Heap_HEADER__
// File maxheap.h

// Max-heap class
template <class Elem, class Comp> class maxheap {
private:
  Elem* Heap;          // Pointer to the heap array
  int size;            // Maximum size of the heap
  int n;               // Number of elements now in the heap
  void siftdown(int);  // Put element in its correct place

public:
  maxheap(Elem* h, int num, int max)     // Constructor
    { Heap = h;  n = num;  size = max;  buildHeap(); }
  int heapsize() const       // Return current heap size
    { return n; }
  bool isLeaf(int pos) const // True if pos is a leaf
    { return (pos >= n/2) && (pos < n); }
  int leftchild(int pos) const
    { return 2*pos + 1; }    // Return leftchild position
  int rightchild(int pos) const
    { return 2*pos + 2; }    // Return rightchild position
  int parent(int pos) const  // Return parent position
    { return (pos-1)/2; }
  bool insert(const Elem&);  // Insert value into heap
  bool removemax(Elem&);     // Remove maximum value
  bool remove(int, Elem&);   // Remove from given position
  void buildHeap()           // Heapify contents of Heap
    { for (int i=n/2-1; i>=0; i--) siftdown(i); }
  void printHeapValues(void) const {
       cout << "\n";
       if (n == 0) { cout << "Empty!\n"; return; }
       for (int i = 0;i < n; i++)
         // "<<" has been overloaded to print Elem objects
         cout << Heap[i] << "  ";
       cout << "\n\n";
  }
};

template <class Elem, class Comp> // Utility function
void maxheap<Elem, Comp>::siftdown(int pos) {
  while (!isLeaf(pos)) {     // Stop if pos is a leaf
    int j = leftchild(pos);  int rc = rightchild(pos);
    if ((rc < n) && Comp::lt(Heap[j], Heap[rc]))
      j = rc;        // Set j to greater child's value
    if (!Comp::lt(Heap[pos], Heap[j])) return; // Done
    swap(Heap, pos, j);
    pos = j;         // Move down
  }
}
```

```
template <class Elem, class Comp> // Insert element
bool maxheap<Elem, Comp>::insert(const Elem& val) {
  if (n >= size) return false; // Heap is full
  int curr = n++;
  Heap[curr] = val;            // Start at end of heap
  // Now sift up until curr's parent > curr
  while ((curr!=0) &&(Comp::gt(Heap[curr], Heap[parent(curr)]))) {
    swap(Heap, curr, parent(curr));
    curr = parent(curr);
  }
  return true;
}

template <class Elem, class Comp> // Remove max value
bool maxheap<Elem, Comp>::removemax(Elem& it) {
  if (n == 0) return false; // Heap is empty
  swap(Heap, 0, --n);       // Swap max with last value
  if (n != 0) siftdown(0);  // Siftdown new root val
  it = Heap[n];             // Return deleted value
  return true;
}

// Remove the value at a specified position
template <class Elem, class Comp>
bool maxheap<Elem, Comp>::remove(int pos, Elem& it) {
  if ((pos < 0) || (pos >= n)) return false;     // Bad pos
  swap(Heap, pos, --n);            // Swap with last value
  while ((pos != 0) && (Comp::gt(Heap[pos], Heap[parent(pos)]))) {
    swap(Heap, curr, parent(curr));// Push up if large key
    curr = parent(curr);
  }
  siftdown(pos);                   // Push down if small key
  it = Heap[n];
  return true;
}


#endif
```

```
// Example of TEST program

#include <iostream>
using namespace std;
#include "book.h"
#include "compare.h"  // Include comparator functions
#include "permute.h"  // Include permutation function
#include "heap.h"     // Implementation for max heap

// Test out the max heap implementation
void main(void)  {
  int i, j;
  int n;
  Int* A[20];
  Int* B[20];
  maxheap<Int*, IntsIntsCompare> BH(B, 0, 20); // empty heap with 20 slots

  n = 10; // heapsize

  Randomize();
  for (i=0; i<n; i++) A[i] = new Int(i);
  permute(A, n);

  cout << "Initial values in an array A:\n";
  for (i=0; i<n; i++) cout << A[i] << "  ";
  cout << "\n\n";

  cout << "I am inserting A[] values in a maxheap BH (one by one)! \n\n";
  for (i=0; i<n; i++) BH.insert(A[i]);

  cout << "Printing maxheap values... \n";
  BH.printHeapValues();

  // Inizializing a heap with 20 slots out of an array with n values
  maxheap<Int*, IntsIntsCompare> AH(A, n, 20);
  Int* AHval;

  cout << "Printing AH maxheap values as heapfying result of A[]... \n";
  AH.printHeapValues();

  cout << "Removing the max value... \n";
  AH.removemax(AHval);
  cout << "Max value: " << AHval << "\n";

  cout << "Removing the max value... \n";
  AH.removemax(AHval);
  cout << "Max value: " << AHval << "\n";

  cout << "Printing AH maxheap values...\n";
  AH.printHeapValues();






  cout << "Removing the value in position 2....\n";
```

```
  AH.remove(2, AHval);
  cout << "Removed value: " << AHval << "\n";

  cout << "Printing AH maxheap values...\n";
  AH.printHeapValues();

  Int C[10] = {73, 6, 57, 88, 60, 34, 83, 72, 48, 85};
  maxheap<Int, IntIntCompare> Test(C, 10, 10);

  cout << "\n\nI am heapfying an automatic array of 10 Int Objects\n";
  Test.printHeapValues();

  Int Testval;
  for (j=0; j<10; j++) {
    Test.removemax(Testval);
     Test.printHeapValues();
  }
} // end main
```