```
#include <string.h>

/** Execute InsertionSort to sort the pointers in the array. */
void sortPointers (char **ar, int n) {
  int j;
  for (j = 1; j < n; j++) {
    int i = j-1;
    void *value = ar[j];
    while (i >= 0 && strcmp(ar[i], value)> 0) {
      ar[i+1] = ar[i];
      i--;
    }

    ar[i+1] = value;
  }
}
```

```
package algs.model.searchtree;

public class AStarSearch implements ISearch {

  /** Scoring function to use. */
  IScore scoringFunction;

  /** Prepare an A* search using the given scoring function. */
  public AStarSearch (IScore sf) { this.scoringFunction = sf; }

  /** Initiate the search for the goal state. */
  public Solution search(INode initial, INode goal) {
    // Start from the initial state
    INodeSet open = StateStorageFactory.create(StateStorageFactory.TREE);
    INode copy = initial.copy();
    scoringFunction.score(copy);
    open.insert(copy);

    // states we have already visited are stored in a queue unless configured.
    INodeSet closed = StateStorageFactory.create(StateStorageFactory.HASH);
    while (!open.isEmpty()) {
      // Remove node with smallest evaluation function and mark closed.
      INode n = open.remove();
      closed.insert(n);

      // Return if goal state reached.
      if (n.equals(goal)) {
        numOpen = open.size(); numClosed = closed.size();  /* STATS */
        return new Solution (initial, n);
      }

      // Compute successor moves and update OPEN/CLOSED lists.
      DepthTransition trans = (DepthTransition) n.storedData();
      int depth = 1;
      if (trans != null) { depth = trans.depth+1; }

      DoubleLinkedList<IMove> moves = n.validMoves();
      for (Iterator<IMove> it = moves.iterator(); it.hasNext(); ) {
        IMove move = it.next();

        // Make move and score the new board state.
        INode successor = n.copy();
        move.execute(successor);

        // Record previous move for solution trace and compute
        // evaluation function to see if we have improved upon
        // a state already closed
        successor.storedData(new DepthTransition(move, n, depth));
        scoringFunction.score(successor);

        // If already visited, see if we are revisiting with lower cost.
        // If not, just continue; otherwise, pull out of closed and process
        INode past = closed.contains(successor);
        if (past != null) {
          if (successor.score() >= past.score()) {
            continue;
          }
          closed.remove(past);    // we revisit with our lower cost.
        }
        open.insert (successor);  // place into open.
      }
    }
    return new Solution (initial, goal, false);    // No solution.
  }
}
```

```java
package algs.model.searchtree;

/**
 * Given an initial state and a target goal state, expand in breadth-first
 * manner all available moves until the target goal state is reached.
 * <p>
 * This search approach is guaranteed to find the shortest possible path to
 * the goal state, should one exist.
 */
public class BreadthFirstSearch implements ISearch {

  /**
   * Initiate the search for the target state.
   */
  public Solution search(INode initial, INode goal) {
    // Return now if initial is the goal
    if (initial.equals(goal)) {  return new Solution (initial, goal); }

    // Start from the initial state
    INodeSet open = StateStorageFactory.create(StateStorageFactory.QUEUE);
    open.insert(initial.copy());

    // states we have already visited.
    INodeSet closed = StateStorageFactory.create(StateStorageFactory.HASH);
    while (!open.isEmpty()) {
      INode n = open.remove();
      closed.insert(n);

      // All successor moves translate into appended OPEN states.
      DoubleLinkedList<IMove> moves = n.validMoves();
      for (Iterator<IMove> it = moves.iterator(); it.hasNext(); ) {
        IMove move = it.next();

        // make move on a copy
        INode successor = n.copy();
        move.execute(successor);

        // If already visited, search this state no more
        if (closed.contains(successor) != null) {
          continue;
        }

        // Record previous move for solution trace. If solution, leave
        // now, otherwise add to the OPEN set.
        successor.storedData(new Transition(move, n));
        if (successor.equals(goal)) {
          return new Solution (initial, successor);
        }
        open.insert(successor);
      }
    }

    // No solution.
    return new Solution (initial, goal, false);
  }

}
```

```java
package algs.model.searchtree;

public class DepthFirstSearch implements ISearch {

  /** Depth bound. */
  int depthBound;

  /**
   * Initiate the Depth First Search with the given fixed-depth bound to search.
   * @param bound    fixed depth to which to search.
   */
  public DepthFirstSearch (int bound) {
    this.depthBound = bound;
  }

  /**
   * Initiate the search for the target state.
   * Store with each INode object a Transition (Move m, INode prev) so we
   * can retrace steps to the original solution.
   */
  public Solution search(INode initial, INode goal) {
    // If goal is initial, return now.
    if (initial.equals(goal)) { return new Solution (initial, goal); }

    INodeSet open = StateStorageFactory.create(StateStorageFactory.STACK);
    open.insert(initial.copy());

    // states we have already visited.
    INodeSet closed = StateStorageFactory.create(StateStorageFactory.HASH);
    while (!open.isEmpty()) {
      INode n = open.remove();
      closed.insert(n);

      // Prepare for computations
      DepthTransition trans = (DepthTransition) n.storedData();

      // All successor moves translate into appended OPEN states.
      DoubleLinkedList<IMove> moves = n.validMoves();
      for (Iterator<IMove> it = moves.iterator(); it.hasNext(); ) {
        IMove move = it.next();

        // Execute move on a copy since we maintain sets of board states
        INode successor = n.copy();
        move.execute(successor);

        // If already visited, try another state
        if (closed.contains(successor) != null) { continue;  }

        int depth = 1;
        if (trans != null) { depth = trans.depth+1; }

        // Record previous move for solution trace. If solution, leave
        // now, otherwise add to the OPEN set if still within depth bound.
        successor.storedData(new DepthTransition(move, n, depth));
        if (successor.equals(goal)) {
          return new Solution (initial, successor);
        }
        if (depth < depthBound) { open.insert (successor); }
      }
    }

    // No solution.
    return new Solution (initial, goal, false);
  }
}
```

```
#include <iostream>
#include "BinaryHeap.h"
#include "Graph.h"

/**
 * Given directed, weighted graph, compute shortest distance to vertices
 * (dist) and record predecessor links (pred) for all vertices.
 * \param g     the graph to be processed.
 * \param s     the source vertex from which to compute all paths.
 * \param dist  array to contain shortest distances to all other vertices.
 * \param pred  array to contain previous vertices to be able to recompute paths.
 */
void singleSourceShortest(Graph const &g, int s,                  /* in */
                          vector<int> &dist, vector<int> &pred) { /* out */

  // initialize dist[] and pred[] arrays. Start with vertex s by setting
  // dist[] to 0. Priority Queue PQ contains all v in G.
  const int n = g.numVertices();
  pred.assign(n, -1);
  dist.assign(n, numeric_limits<int>::max());
  dist[s] = 0;
  BinaryHeap pq(n);
  for (int u = 0; u < n; u++) { pq.insert (u, dist[u]); }

  // find vertex in ever-shrinking set, V-S, whose dist[] is smallest.
  // Recompute potential new paths to update all shortest paths
  while (!pq.isEmpty()) {
    int u = pq.smallest();

    // For neighbors of u, see if newLen (best path from s->u + weight
    // of edge u->v) is better than best path from s->v. If so, update
    // in dist[v] and re-adjust binary heap accordingly. Compute in
    // long to avoid overflow error.
    for (VertexList::const_iterator ci = g.begin(u); ci != g.end(u); ++ci) {
      int v = ci->first;
      long newLen = dist[u];
      newLen += ci->second;
      if (newLen < dist[v]) {
        pq.decreaseKey (v, newLen);
        dist[v] = newLen;
        pred[v] = u;
      }
    }
  }
}
```

```
#include "bfs.h"

/**
 * Perform breadth-first search on graph from vertex s, and compute BFS
 * distance and pred vertex for all vertices in the graph.
 */
void bfs_search (Graph const &graph, int s,              /* in */
                 vector<int> &dist, vector<int> &pred)  /* out */
{
  // initialize dist and pred to mark vertices as unvisited. Begin at s
  // and mark as Gray since we haven't yet visited its neighbors.
  const int n = graph.numVertices();
  pred.assign(n, -1);
  dist.assign(n, numeric_limits<int>::max());
  vector<vertexColor> color (n, White);

  dist[s] = 0;
  color[s] = Gray;

  queue<int> q;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();

    // Explore neighbors of u to expand the search horizon
    for (VertexList::const_iterator ci = graph.begin(u);
         ci != graph.end(u); ++ci) {
      int v = ci->first;
      if (color[v] == White) {
        dist[v] = dist[u]+1;
        pred[v] = u;
        color[v] = Gray;
        q.push(v);
      }
    }

    q.pop();
    color[u] = Black;
  }
}
```

```
#include "dfs.h"

/**
 * Visit a vertex, u, in the graph and update information.
 * \param graph    the graph being searched.
 * \param u        the vertex being visited.
 * \param pred     array of previous vertices in the depth-first search tree.
 * \param color    array of vertex colors in the depth-first search tree.
 */
void dfs_visit (Graph const &graph, int u,                      /* in */
                vector<int> &pred, vector<vertexColor> &color)  /* out */
{
  color[u] = Gray;

  // process all neighbors of u.
  for (VertexList::const_iterator ci = graph.begin(u);
       ci != graph.end(u); ++ci) {
    int v = ci->first;

    // Explore unvisited vertices immediately and record pred[].
    // Once recursive call ends, backtrack to adjacent vertices.
    if (color[v] == White) {
      pred[v] = u;
      dfs_visit (graph, v, pred, color);
    }
  }

  color[u] = Black;  // our neighbors are complete; now so are we.
}

/**
 * Perform Depth First Search starting from vertex s, and compute the
 * predecessor vertex to u in resulting depth-first search forest).
 *
 * \param graph    the graph being searched.
 * \param s        the vertex to use as the source vertex.
 * \param pred     array of previous vertices in the depth-first search tree.
 */
void dfs_search (Graph const &graph, int s,         /* in */
                 vector<int> &pred)                 /* out */
{
  // initialize d[], f[], and pred[] arrays. Mark all vertices White
  // to signify unvisited. Clear out edge labels.
  const int n = graph.numVertices();
  vector<vertexColor> color (n, White);
  pred.assign(n, -1);

  // Search starting at the source vertex; when done, visit any
  // vertices that remain unvisited.
  dfs_visit (graph, s, pred, color);
  for (int u = 0; u < n; u++) {
    if (color[u] == White) {
      dfs_visit (graph, u, pred, color);
    }
  }
}
```

```c
#include <string.h>
#include <stdio.h>

/**
 * Execute a binary probe on sorted array front half and then execute
 * a block move of pointers. Over time, this should require only log(n)
 * probes and replace O(n) swaps with a single block move.
 */
void sortPointers (char **ar, int n) {
  int j;

  for (j = 1; j < n; j++) {

    /** invariant: ar[0, j) is sorted. */

    /** Search for the desired target within the search structure. */
    int low = 0, high = j-1, ix, rc, sz;
    char *target = ar[j];
    while (low <= high) {
      ix = (low + high)/2;
      rc = strcmp(target, ar[ix]);

      if (rc < 0) {
        /* target is less than ar[i] */
        high = ix - 1;
      } else if (rc > 0) {
        /* target is greater than ar[i] */
        low = ix + 1;
      } else {
        /* found the item. */
        break;
      }
    }

    /** low determines index value in to which it should be inserted (not
        ix as stated on p. 116) only move if we are not already properly in
        place. */
    if (low != j) {
      sz = (j-low)*sizeof(char *);
      memmove (&ar[low+1], &ar[low], sz);
      ar[low] = target;
    }

  }
}
```

```java
package algs.model.search;

/**
 * Binary Search in Java given a pre-sorted array of the parameterized type.
 *
 * @param T   elements of the collection being searched are of this type.
 *            The parameter T must implement Comparable.
 *
 * @author George Heineman
 * @version 1.0, 6/15/08
 * @since 1.0
 */
public class BinarySearch<T extends Comparable<T>> {

  /** Search for target in collection. Return true on success. */
  public boolean search(T[] collection, T target) {
    // null is never included in the collection
    if (target == null) { return false; }

    int low = 0, high = collection.length - 1;
    while (low <= high) {
      int ix = (low + high)/2;
      int rc = target.compareTo(collection[ix]);

      if (rc < 0) {
        // target is less than collection[i]
        high = ix - 1;
      } else if (rc > 0) {
        // target is greater than collection[i]
        low = ix + 1;
      } else {
        // found the item.
        return true;
      }
    }

    return false;
  }
}
```

```
/** Heapify the subarray ar[0,max). */
static void heapify (void **ar, int(*cmp)(const void *,const void *),
                     int idx, int max) {
  int left = 2*idx + 1;
  int right = 2*idx + 2;
  int largest;

  /* Find largest element of A[idx], A[left], and A[right]. */
  if (left < max && cmp (ar[left], ar[idx]) > 0) {
    largest = left;
  } else {
    largest = idx;
  }

  if (right < max && cmp(ar[right], ar[largest]) > 0) {
    largest = right;
  }

  /* If largest is not already the parent then swap and propagate. */
  if (largest != idx) {
    void *tmp;

    tmp = ar[idx];
    ar[idx] = ar[largest];
    ar[largest] = tmp;

    heapify(ar, cmp, largest, max);
  }
}

/** Build the heap from the given array by repeatedly invoking heapify. */
static void buildHeap (void **ar, int(*cmp)(const void *,const void *),
                       int n) {
  int i;
  for (i = n/2-1; i>=0; i--) {
    heapify (ar, cmp, i, n);
  }
}

/** Sort the array using Heap Sort implementation. */
void sortPointers (void **ar, int n,
                   int(*cmp)(const void *,const void *))
{
  int i;
  buildHeap (ar, cmp, n);
  for (i = n-1; i >= 1; i--) {
    void *tmp;

    tmp = ar[0];
    ar[0] = ar[i];
    ar[i] = tmp;

    heapify (ar, cmp, 0, i);
  }
}
```

```c
/**
 * In linear time, group the sub-array ar[left, right) around a pivot
 * element pivot=ar[pivotIndex] by storing pivot into its proper location,
 * store, within the sub-array (whose location is returned by this
 * function) and ensuring that all ar[left,store) <= pivot and all
 * ar[store+1,right) > pivot.
 *
 * @param ar            array of elements to be sorted.
 * @param cmp           comparison function to order elements.
 * @param left          lower bound index position  (inclusive)
 * @param right         upper bound index position  (exclusive)
 * @param pivotIndex    index around which the partition is being made.
 * @return              location of the pivot index properly positioned.
 */
int partition (void **ar, int(*cmp)(const void *,const void *),
               int left, int right, int pivotIndex) {
  void *tmp, *pivot;
  int idx, store;

  pivot = ar[pivotIndex];

  /* move pivot to the end of the array */
  tmp = ar[right];
  ar[right] = ar[pivotIndex];
  ar[pivotIndex] = tmp;

  /* all values <= pivot are moved to front of array and pivot inserted
   * just after them. */
  store = left;
  for (idx = left; idx < right; idx++) {
    if (cmp(ar[idx], pivot) <= 0) {
      tmp = ar[idx];
      ar[idx] = ar[store];
      ar[store] = tmp;
      store++;
    }
  }

  tmp = ar[right];
  ar[right] = ar[store];
  ar[store] = tmp;
  return store;
}
```

```java
package algs.model.gametree;

public class MinimaxEvaluation implements IEvaluation {
  IGameState state;      /** Game state. */
  int ply;               /** Ply depth. */
  IPlayer original;      /** Use perspective from Initial Player. */

  /** Create an evaluator with the ply-depth. */
  public MinimaxEvaluation (int ply) { this.ply = ply; }

  /** Initiates MiniMax of ply-depth to find best move for player. */
  public IGameMove bestMove (IGameState s, IPlayer player, IPlayer opponent) {
    this.original = player;
    this.state = s.copy();

    MoveEvaluation move = minimax(ply, IComparator.MAX, player, opponent);
    return move.move;
  }

  /**
   * Given the game state, use minimax algorithm to locate best move
   * for original player.
   *
   * @param ply        the fixed depth to look ahead.
   * @param comp       the type (MIN or MAX) of this level, to evaluate moves.
   *                   MAX selects best move while MIN selects worst moves.
   * @param player     the current player.
   * @param opponent   the opponent.
   */
  private MoveEvaluation minimax (int ply, IComparator comp,
      IPlayer player, IPlayer opponent) {

    // If no allowed moves or a leaf node, return game state score.
    Iterator<IGameMove> it = player.validMoves(state).iterator();
    if (ply == 0 || !it.hasNext()) {
      return new MoveEvaluation (original.eval(state));
    }

    // Try to improve on this lower-bound (based on selector).
    MoveEvaluation best = new MoveEvaluation (comp.initialValue());

    // Generate game states that result from all valid moves for this player.
    while (it.hasNext()) {
      IGameMove move = it.next();

      move.execute(state);

      // Recursively evaluate position. Compute MiniMax and swap params
      MoveEvaluation me = minimax (ply-1, comp.opposite(), opponent, player);

      move.undo(state);

      // Select maximum (minimum) of children if we are MAX (MIN)
      if (comp.compare(best.score, me.score) < 0) {
        best = new MoveEvaluation (move, me.score);
      }
    }
    return best;
  }
}
```