```cpp
template <class val_t>
class kdTree {
private:
  using vval_t = vector<val_t>;
  struct node {
    int id;
    int deg;
    vval_t val;
    node *parent;
    node *child_l, *child_r;
    vval_t range_l, range_r;
  };
  int dimension; // dimension
  int n; // the number of nodes
  node *root; // the root of the tree
  node *nil; // the node for leaves of the tree
  struct idval_t {
    int id;
    vval_t val;
  };
  vector<idval_t> ary;
  inline void update_cover_range(node *focus, node* target) {
    if (target == nil) return;
    else {
      Loop(i, dimension) {
        focus->range_l[i] = min(focus->range_l[i], target->range_l[i]);
        focus->range_r[i] = max(focus->range_r[i], target->range_r[i]);
      }
      return;
    }
  }
  node* build_kdTree_rec(node *parent, int l, int r, int depth) {
    if (r - l == 0) return nil;
    node *ret = new node;
    int axis = depth % dimension;
    int mid = (l + r) / 2;
    nth_element(ary.begin() + l, ary.begin() + mid, ary.begin() + r, [=](const idval_t& l, const idval_t& r) { return l.val[axis] < r.val[axis]; });
    *ret = { ary[mid].id, depth, ary[mid].val, nil, nil, nil, ary[mid].val, ary[mid].val };
    ret->child_l = build_kdTree_rec(ret, l, mid, depth + 1);
    update_cover_range(ret, ret->child_l);
    ret->child_r = build_kdTree_rec(ret, mid + 1, r, depth + 1);
    update_cover_range(ret, ret->child_r);
    return ret;
  }
  inline bool check_crossed_find_range(node *focus, pair<vval_t, vval_t> &range) {
    if (focus == nil) return false;
    Loop(i, dimension) {
      if (range.first[i] <= focus->range_r[i] && focus->range_l[i] <= range.second[i]) continue;
      else return false;
    }
    return true;
  }
  inline bool check_in_range(node *focus, pair<vval_t, vval_t> &range) {
    if (focus == nil) return false;
    Loop(i, dimension) {
      if (range.first[i] <= focus->val[i] && focus->val[i] <= range.second[i]) continue;
      else return false;
    }
    return true;
  }
  void find_in_range_rec(node *focus, pair<vval_t, vval_t> &range, int depth, vi &in_range_list) {
    if (focus == nil) return;
    else {
      int axis = depth % dimension;
      if (check_in_range(focus, range)) in_range_list.push_back(focus->id);
      if (check_crossed_find_range(focus->child_l, range)) {
        find_in_range_rec(focus->child_l, range, depth + 1, in_range_list);
      }
      if (check_crossed_find_range(focus->child_r, range)) {
        find_in_range_rec(focus->child_r, range, depth + 1, in_range_list);
      }
    }
  }
public:
  kdTree(const vector<vval_t> &A, int dimension) {
    n = (int)A.size();
    this->dimension = dimension;
    ary.resize(n);
    Loop(i, n) ary[i] = { i, A[i] };
    nil = new node;
    root = build_kdTree_rec(nil, 0, n, 0);
    return;
  }
  // return id of vals in [range.first, range.second]
  vi find_in_range(pair<vval_t, vval_t> range) {
    vi ret;
    find_in_range_rec(root, range, 0, ret);
    sort(ret.begin(), ret.end());
    return ret;
  }
};
```