```cpp
// Insertion Sort, left to right
void InsertionSort(Item* A, size_t n) {
    for (size_t i = 1; i < n; ++i) {
        Item key = A[i];
        ssize_t j = i - 1;
        while (j >= 0 && A[j] > key) {
            swap(A[j], A[j + 1]);
            j--;
        }
    }
}

// Selection Sort, left to right
void SelectionSort(Item* A, size_t n) {
    for (size_t i = 0; i < n - 1; ++i) {
        size_t j_min = i;
        for (size_t j = i + 1; j < n; ++j) {
            if (A[j] < A[j_min])
                j_min = j;
        }
        swap(A[i], A[j_min]);
    }
}
```

```
// BubbleSort O(n^2)
void BubbleSort(Item* A, size_t n) {
    for (size_t i = 0; i < n - 1; ++i) {
        for (size_t j = 0; j < n - 1 - i; ++j) {
            if (A[j] > A[j + 1])
                swap(A[j], A[j + 1]);
        }
    }
}

// Shell's Sort
void ShellSort(Item* A, size_t n) {
    size_t incs[16] = {
        1391376, 463792, 198768, 86961, 33936, 13776, 4592, 1968,
        861, 336, 112, 48, 21, 7, 3, 1
    };
    for (size_t k = 0; k < 16; k++) {
        for (size_t h = incs[k], i = h; i < n; i++) {
            Item v = A[i];
            size_t j = i;
            while (j >= h && A[j - h] > v) {
                A[j] = A[j - h];
                j -= h;
            }
            A[j] = v;
        }
    }
}
```

```
// QuickSort with Hoare's partition scheme
void QuickSortLR(Item* A, ssize_t lo, ssize_t hi) {
    ssize_t p = QuickSortSelectPivot(A, lo, hi + 1);
    ssize_t i = lo, j = hi;
    while (i <= j) {
        while (A[i] < A[p])
            i++;
        while (A[j] > A[p])
            j--;
        if (i <= j) {
            swap(A[i], A[j]);
            // follow pivot if it is swapped
            p = (p == i ? j : p == j ? i : p);
            i++, j--;
        }
    }
    if (lo < j)
        QuickSortLR(A, lo, j);
    if (i < hi)
        QuickSortLR(A, i, hi);
}
void QuickSortLR(Item* A, size_t n) {
    QuickSortLR(A, 0, n - 1);
}
```

```
// QuickSort with Lomuto's partition scheme
size_t PartitionLL(Item* A, size_t lo, size_t hi) {
    // pick pivot and move to back
    size_t p = QuickSortSelectPivot(A, lo, hi);
    Item& pivot = A[p];
    swap(A[p], A[hi - 1]);
    ssize_t i = lo;
    for (size_t j = lo; j < hi - 1; ++j) {
        if (A[j] <= pivot)
            swap(A[i], A[j]), ++i;
    }
    swap(A[i], A[hi - 1]);
    return i;
}
void QuickSortLL(Item* A, size_t lo, size_t hi) {
    if (lo + 1 < hi) {
        size_t mid = PartitionLL(A, lo, hi);
        QuickSortLL(A, lo, mid), QuickSortLL(A, mid + 1, hi);
    }
}
void QuickSortLL(Item* A, size_t n) {
    QuickSortLL(A, 0, n);
}
```

```
// Dual-Pivot Quick Sort (code by Yaroslavskiy)
void QuickSortDualPivotYaroslavskiy(Item* A, int left, int right) {
    if (right > left) {
        if (A[left] > A[right])
            swap(A[left], A[right]);
        const Item p = A[left], q = A[right];
        ssize_t l = left + 1, g = right - 1, k = l;
        while (k <= g) {
            if (A[k] < p) {
                swap(A[k], A[l]), ++l;
            }
            else if (A[k] >= q) {
                while (A[g] > q && k < g)
                    --g;
                swap(A[k], A[g]), --g;
                if (A[k] < p)
                    swap(A[k], A[l]), ++l;
            }
            ++k;
        }
        --l, ++g;
        swap(A[left], A[l]), swap(A[right], A[g]);
        QuickSortDualPivotYaroslavskiy(A, left, l - 1);
        QuickSortDualPivotYaroslavskiy(A, l + 1, g - 1);
        QuickSortDualPivotYaroslavskiy(A, g + 1, right);
    }
}
```

```
void Merge(Item* A, size_t lo, size_t mid, size_t hi) {
    Item out[hi - lo];                          // allocate output
    size_t i = lo, j = mid, o = 0;              // merge first and second halves
    while (i < mid && j < hi) {
        Item ai = A[i], aj = A[j];              // copy out for fewer time steps
        out[o++] = (ai < aj ? (++i, ai) : (++j, aj));
    }
    while (i < mid)                             // copy rest
        out[o++] = A[i++];
    while (j < hi)
        out[o++] = A[j++];
    for (i = 0; i < hi - lo; ++i)               // copy back
        A[lo + i] = out[i];
}
void MergeSort(Item* A, size_t lo, size_t hi) {
    if (lo + 1 < hi) {
        size_t mid = (lo + hi) / 2;
        MergeSort(A, lo, mid), MergeSort(A, mid, hi);
        Merge(A, lo, mid, hi);
    }
}
void MergeSort(Item* A, size_t n) {
    return MergeSort(A, 0, n);
}
```

```
void HeapSort(Item* A, size_t n) {
    size_t i = n / 2;
    while (1) {
        if (i > 0) {
            // build heap, sift A[i] down the heap
            i--;
        }
        else {
            // pop largest element from heap: swap front to back, and sift
            // front A[0] down the heap
            if (--n == 0)
                return;
            swap(A[0], A[n]);
        }
        size_t parent = i, child = i * 2 + 1;
        // sift operation - push the value_ of A[i] down the heap
        while (child < n) {
            if (child + 1 < n && A[child + 1] > A[child]) {
                child++;
            }
            if (A[child] > A[parent]) {
                swap(A[parent], A[child]);
                parent = child, child = parent * 2 + 1;
            }
            else
                break;
        }
    }
}
```

```
void LinearProbingHT(Item* A, size_t n) {
    size_t cshift = random(n);
    for (size_t i = 0; i < n; ++i) {
        // pick a new item to insert
        Item v = Item((i + cshift) % n);
        size_t idx = (hash(v.value()) >> 2) % n;
        while (A[idx].value() != black) {
            idx = (idx + 1) % n;
        }
        A[idx] = v;
    }
}

void QuadraticProbingHT(Item* A, size_t n) {
    size_t cshift = random(n);
    for (size_t i = 0; i < n; ++i) {
        // pick a new item to insert
        Item v = Item((i + cshift) % n);
        size_t idx = (hash(v.value()) >> 2) % n;
        size_t p = 0;
        while (A[idx].value() != black) {
            idx = (idx + (p + p * p) / 2) % n;
            if (++p == n)
                return;  // cycled. stop hashing.
        }
        A[idx] = v;
    }
}
```

```
void CuckooHashingTwo(Item* A, size_t n) {
    size_t cshift = random(n);
    for (size_t i = 0; i < n; ++i) {
        // pick a new item to insert
        Item v = Item((i + cshift) % n);
        uint32_t pos = hash2(0, v.value()) % n;
        if (A[pos].value() == black) {
            A[pos] = v; continue;
        }
        pos = hash2(1, v.value()) % n;
        if (A[pos].value() == black) {
            A[pos] = v; continue;
        }
        size_t r = 0;
        int hashfunction = 1;
        while (true) {
            pos = hash2(hashfunction, v.value()) % n;
            swap(v, A[pos]);
            if (v.value() == black)
                break;
            if (hash2(hashfunction, v.value()) % n == pos)
                hashfunction = (hashfunction + 1) % 2;
            if (++r >= n)
                return;
        }
    }
}
```

```
void CuckooHashingThree(Item* A, size_t n) {
    size_t cshift = random(n);
    for (size_t i = 0; i < n; ++i) {
        // pick a new item to insert
        Item v = Item((i + cshift) % n);
        uint32_t pos = hash3(0, v.value()) % n;
        if (A[pos].value() == black) {
            A[pos] = v; continue;
        }
        pos = hash3(1, v.value()) % n;
        if (A[pos].value() == black) {
            A[pos] = v;
            continue;
        }
        pos = hash3(2, v.value()) % n;
        if (A[pos].value() == black) {
            A[pos] = v;
            continue;
        }
        // all three places full,
        // pick a random one to displace
        int hashfunction = random(3);
        pos = hash3(hashfunction, v.value()) % n;
        swap(v, A[pos]);
        size_t r = 0;
        while (true) {
            hashfunction = random(2);
            if (hash3(hashfunction, v.value()) % n == pos)
                hashfunction = 2;
            pos = hash3(hashfunction, v.value()) % n;
            swap(v, A[pos]);
            if (v.value() == black)
                break;
            if (++r >= n)
                return;
        }
    }
}
```