common

- isEmpty
- size
- getNext
- getItem
- checkInvariants (optional)

singly-linked list & circular list

- insertAfter
- remove (or removeNext)
- addHead
- removeHead
- C implementation:
  - uses head and tail for linked list
  - uses just head for circular list

doubly-linked list

- insertAfter
- insertBefore
- remove
- addHead
- removeHead
- addTail
- removeTail
- C implementation:
  - uses head and tail

bag

- add
- Java implementation:
  - just uses head

stack

- push
- pop
- peek
- Java implementation:
  - just uses head and runs push, pop, and peek from the head
- C implementation:
  - just uses List and adds push, pop, and peek (from the head)

queue

- peek
- enqueue
- dequeue
- Java implementation:
  - uses head and tail
  - enqueue works from the tail
  - dequeue works from the head
- C implementation:
  - just uses List and adds enqueue (from the tail), dequeue (from the head), and peek

deque

- pushHead

- popHead
- pushTail
- popTail
- head (for peek)
- tail (for peek)

Notes

- each of the above can use an array as the underlying storage but generally requires resizing and storage of the size
- without array resizing, consider as a fixed-capacity data structure
- keep in mind that conceptually a link is the connection between nodes, so while the "next" member may be a node/pointer to a node, it represents a separate concept (and can be represented differently, i.e. and array index, etc)
- Java generic array creating/casting

```
Item [] a = (Item[]) new Object[cap];

// or an array of stacks of strings
Stack<String>[] a = (Stack<String>[]) new Stack[N];
```

- Java resize

```
private void resize(int capacity) {
    assert capacity >= n;
    Item[] temp = (Item[]) new Object[capacity];
    for (int i = 0; i < n; i++) {
        temp[i] = a[i];
    }
    a = temp;
}
```

- C resize

```
int resize(Item *array, size_t capacity) {
    if (capacity < sizeof(array) / sizeof(Item) {
        return -1;
    }

    array = realloc(array, capacity * sizeof(Item));

    if (array == NULL) {
        return -1;
    }
    return 0;
}
```

- C++ resize (with exception)

```
template<typename Item>
Item *resize(Item *array, size_t orig_size, size_t new_size) {
    if (new_size < orig_size) {
        return;
    }

    if (array == nullptr) {
        std::cerr << "Bad array pointer \n";
        return;
    }

    Item *tmp;
    try {
        tmp = new Item[new_size];
    } catch (std::bad_alloc &e) {
        std::cerr << "bad_alloc: " << e.what() << "\n";
        return;  // or exit
    }
    // std::memcpy(tmp, array, orig_size);
    for (ptrdiff_t i = 0; i < orig_size; i++) {
        *(tmp + i) = *(array + i);
    }
    return tmp;
}
```

- C++ resize (without exceptions)

```
template<typename Item>
Item *resize(Item *array, size_t orig_size, size_t new_size) {
    if (new_size < orig_size) {
        return;
    }

    if (array == nullptr) {
        std::cerr << "Bad array pointer \n";
        return;
    }

    Item *tmp = new(std::nothrow) Item[capacity];
    if (a == nullptr) {
        std::cerr << "operator new with std::nothrow returned nullptr\n";
        return; // or exit
    }
    // std::memcpy(tmp, array, orig_size);
    for (ptrdiff_t i = 0; i < orig_size; i++) {
        *(tmp + i) = *(array + i);
    }
    return tmp;
}
```

head and tail conventions in linked lists

circular, never empty

```
// first insert
head->next = head;

// insert t after x
t->next = x->next;
x->next = t;

// remove after x
x->next = x->next->next;

// traversal loop
t = head;
do {
    ...
    t = t->next;
} while (t != head);

// test for single item
if (head->next == head)
```

head pointer, no tail

```
// initialize
head = nullptr;

// insert t after x
if (x == nullptr) {
    head = t;
    head->next = nullptr;
} else {
    t->next = x->next;
    x->next = t;
}

// remove after x
t = x->next;
x->next = t->next;

// traversal loop
for (t = head; t != nullptr; t = t->next)

// test for empty list
if (head == nullptr)
```

dummy head node, no tail

```
// initialize
head = new node;
head->next = nullptr;

// insert t after x
t->next = x->next;
x->next = t;

// remove after x
t = x->next;
x->next = t->next;

// traversal loop
for (t = head->next; t != nullptr; t = t->next)

// test for empty list
if (head->next == nullptr)
```

dummy head and tail nodes

```
// initialize
head = new node;
tail = new node;
head->next = tail;
tail->next = tail;

// insert t after x
t->next = x->next;
x->next = t;

// remove after x
x->next = x->next->next;

// traversal loop
for (t = head->next; t != tail; t = t->next)

// test for empty list
if (head->next == tail)
```