*Quickselect - use randomized pivot to partition - O(n) to find the k^{th} largest / smallest element in unsorted list.

Arrays and Linked Lists

Arrays

Done using a Java array of a sequence of n items, which occupies a **contiguous** block of memory.

1. Insertion

To insert an item at position i, we need to shift all items from index i to the nth-item by 1 spot to the right, before inserting at the gap created.

- Best Case: O(1) – inserting at the back
- Worst Case: O(n) – inserting at the front
- Average Case: O(n) – O(n/2) to be precise

2. Deletion

Shift current items from index i+1 onwards to the left by 1 to cover up the gap. Need to use num_items to “remove” the original final element.

- Best Case: O(1) – deleting from the back
- Worst Case: O(n) – deleting from the front
- Average Case: O(n) – O(n/2) to be precise

3. Enlarging

Create a new array of double the size and copy the elements over, then point the reference to it.

- O(n)

4. Getters/Accessors

Return element at arr[index].

- O(1) – indexing into an array is constant time due to random access memory of the computer

5. Supporting Operations

empty() – O(1), return true if num_items==0
size() – O(1), return num_items
indexOf(int item) – O(n), to scan through the whole array and return if found, else -1
contains(int item) – O(n), to return if indexOf(item)!=-1

Space Complexity

- Best Case: n space for n items
- Worst Case: 2n space for n+1 items
- Overall: O(n)

Circular Arrays

Circular Arrays are maintained using a front and back index, allowing us to loop from the end of an array back to the front without any overwriting of information. This is only applicable when we would be constantly removing items from the front, and do not wish to keep closing up gaps e.g. in a Queue. The wasted space would be then subsequently utilized by the back of the array when it wraps around.

The operations are changed as such:

1. Insertion

When inserting to the front:

```
front = (front+maxSize-1)%maxSize;
arr[front] = item;
numItems++;
```

When inserting to the back:

```
arr[back] = item;
back = (back+1)%maxSize;
numItems++;
```

2. Deletion

When deleting from the front:

```
front = (front+1)%maxSize;
numItems--;
```

When deleting from the back:

```
back = (back+maxSize-1)%maxSize;
numItems--;
```

3. Enlarging

When numItems==maxSize-1 (empty slot between front and back to differentiate between a full and empty array):

```
Item[] newArr = new Item[maxSize*2];
int counter = 0;
for (int i=front; i!=back; i=(i+1)%maxSize){
      newArr[counter++] = arr[i];
}
// OR
for (int j=0; j<maxSize; j++){
      newArr[j] = arr[(front+j)%maxSize];
}
arr = newArr;
front = 0;
back = maxSize-1;
maxSize *= 2;
```

4. Supporting Operations

empty() – return front==back
size() – return numItems
indexOf() – using the above loop method, loop through and return index if found.

Linked List

Each item is stored in a node, which also contains a next pointer that points to the next node in order. This allows us to order the nodes by associating each with its neighbors and allows elements to occupy non-contiguous memory.

Basic Linked List

We use a head reference to indicate where the first node is. From the head we can access the rest of the linked list.

1. Getters/Accessors

To access an item at index i, we have a reference curr that starts from the head and "moves" towards the node of the correct index.

- Best Case: O(1) – root node
- Worst Case: O(n) – tail node needed
- Average Case: O(n) – O(n/2) to be precise

2. Insertion

To insert a new item at index i, we first create a new node n with the new item, find the node curr at index i-1, then point the next pointer of our n to the curr's next. Then we point curr's next to n. Increment the number of nodes.

- Best Case: O(1) – inserting at root node
- Worst Case: O(n) – inserting at the tail, which requires O(n) accessing the tail
- Average Case: O(n) – O(n/2) to be precise
- Special Cases:
    - Inserting at the front – Point n.next to head then point head to n
    - Empty List – Same as above
    - Inserting at the back – No special handling

3. Deletion

To remove an item at index i, we simply need to point the next pointer of index i-1 to the node at index i+1. We point curr to index i-1, point it to the neighbor of neighbor of curr, and decrement the number of nodes.

- Best Case: O(1) – root node
- Worst Case: O(n) – tail node deleted
- Average Case: O(n) – O(n/2) to be precise
- Special Cases:
    - Removing from the front – Point head to head.next

Tailed Linked List

Have an extra tail pointer to point to the tail

1. Insertion

Tail insertion is now O(1) as we no longer need to run through the whole list to access the tail. After inserting at the tail, point the tail reference to the new tail. If list is empty, adding an element results in head==tail.

2. Accessing

O(1) access to the tail.

<u>Circular Linked List</u>

It is a linked list where the tail node points back to the head node, hence there is no terminating null pointer. This can be useful in allowing us to cycle through a list repeatedly e.g. to allocate shared resources. This is what an operating system does.

Another use case is the maintenance of a queue using a circular linked list. Only one pointer to the last inserted node needs to be maintained, and the front of the queue would be the next node from that.

This can be a Singly Circular Linked List or a Doubly Circular Linked List.

<u>Doubly Linked List</u>

In addition to the next pointer attached to all nodes, we introduce a previous pointer that points to the previous node.

1. <u>Accessing</u>

The time taken is still O(n), but it's of a smaller constant, as we can now start our accessing from either the front or the back, depending on which end is closer to the node we are looking for.

- Worst: O(n/2) -> O(n), when the node is in the middle
- Best: O(1), when the node is the head or tail
- Average: O(n/4) -> O(n)

2. <u>Insertion</u>

The time taken is still O(n), but it's of a smaller constant, as we can now access faster. The time complexities are that of accessing, as insertion by itself is an O(1) operation.

3. <u>Deletion</u>

The time taken is still O(n), but it's of a smaller constant, as we can now access faster. The time complexities are that of accessing, as deletion by itself is an O(1) operation.

	Arrays			Linked Lists		
	Best	**Worst**	**Average**	**Best**	**Worst**	**Average**
Get	O(1) Random Access Memory	O(1) Random Access Memory	O(1) Random Access Memory	O(1) Accessing the head node	O(n) Accessing the tail node	O(n) Accessing n/2 node
Insert	O(1) Inserting at the back O(n) If enlarging is required	O(n) Inserting at the front due to shifting and enlarging	O(n) Due to shifting	O(1) Inserting before head node	O(n) Inserting at tail node due to accessing	O(n) Due to accessing
Delete	O(1) Deleting from the back	O(n) Deleting from the front due to shifting	O(n) Due to shifting	O(1) Delete head node	O(n) Delete tail node due to accessing	O(n) Due to accessing
Adv	• Use Array if only adding to the back • Use Array if we need few insertions/deletions but a lot of getting/accessing • O(1) search time due to contiguous memory • Benefits from Cache Locality			• Use Linked List if only adding to the front • If insertion/deletion at a fixed index is required, then maintain the reference to the node at index-1, allowing O(1) operations thereafter • Dynamic data structure • Efficient memory utilization since no empty space is used • Fast insertion/deletion		
Disadv	• Wasted memory due to empty array • Slow deletion and insertion except at end of array			• Reverse traversing is difficult • Larger space consumption due to the need to store the pointer to the next node • Searching is time-consuming		

Cache Locality

In particular, arrays are contiguous memory blocks, so large chunks of them will be loaded into the cache upon first access. This makes it comparatively quick to access future elements of the array. Linked lists on the other hand aren't necessarily in contiguous blocks of memory, and could lead to more cache misses, which increases the time it takes to access them.

Consider the following possible memory layouts for an array `data` and linked list `l_data` of large structs:

```
Address      Contents          | Address      Contents
ffff 0000    data[0]           | ffff 1000    l_data
ffff 0040    data[1]           |   ....
ffff 0080    data[2]           | ffff 3460    l_data->next
ffff 00c0    data[3]           |   ....
ffff 0100    data[4]           | ffff 8dc0    l_data->next->next
                               | ffff 8e00    l_data->next->next->next
                               |   ....
                               | ffff 8f00    l_data->next->next->next->next
```

If we wanted to loop through this array, the first access to `ffff 0000` would require us to go to memory to retrieve (a very slow operation in CPU cycles). However, after the first access the rest of the array would be in the cache, and subsequent accesses would be much quicker. With the linked list, the first access to `ffff 1000` would also require us to go to memory. Unfortunately, the processor will cache the memory directly surrounding this location, say all the way up to `ffff 2000`. As you can see, this doesn't actually capture any of the other elements of the list, which means that when we go to access `l_data->next`, we will again have to go to memory.

	Tailed Linked Lists			Doubly Linked Lists		
	Best	**Worst**	**Average**	**Best**	**Worst**	**Average**
Get	O(1) Either head or tail node	O(n) The n-1th node, right before tail	O(n) Accessing n/2 node	O(1) Accessing the head or tail node	O(n) Accessing the middle node: O(n/2)	O(n) Accessing the n/4th node: O(n/4)
Insert	O(1) Inserting at the head or tail	O(n) Inserting n-1 due to accessing	O(n) Inserting at n/2	O(1) Inserting before head node or after tail node	O(n) Inserting at n/2 due to accessing	O(n) Inserting at n/4 due to accessing
Delete	O(1) Deleting the head or the tail node	O(n) Deleting n-1 due to accessing	O(n) Deleting n/2 node	O(1) Delete head node or tail node	O(n) Delete n/2 node due to accessing	O(n) Deleting n/4 node due to accessing
Adv	• Use tailed linked list to allow insertion to the back at O(1) time			• Removal from the back is also O(1) • Traversal in both directions can be done easily		
Disadv	• Extra 4-8 bytes more than normal linked list and implementer needs to keep track			• Extra space needed to maintain previous pointer for all nodes		