Data structures refresher

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

January 24, 2023

Today's topics

- Algorithms using data structures
- Basics – array structures
- Linked list refresher

Simple data structures

Example 1

How long time does this procedure take?

```
List<Integer> fn(int n) {
   List<Integer> array = new ArrayList<Integer>(n);

   for (int i=0; i<n; i++) {
      array.add(0,i); // Insert i at position 0
   }
   return array;
}
```

Example 1

How long time does this procedure take?

```
List<Integer> fn(int n) {
   List<Integer> array = new ArrayList<Integer>(n);

   for (int i=0; i<n; i++) {
      array.add(0,i); // Insert i at position 0
   }
   return array;
}
```

Answer: $\Theta(n^2)$ time!

Example 2

What about this one?

```
List<Integer> fn(int n) {
   List<Integer> array = new ArrayList<Integer>();

   for (int i=0; i<n; i++) {
      array.add(i); // Insert i at the end
   }
   return array;
}
```

Example 2

What about this one?

```
List<Integer> fn(int n) {
   List<Integer> array = new ArrayList<Integer>();

   for (int i=0; i<n; i++) {
      array.add(i); // Insert i at the end
   }
   return array;
}
```

Answer: $\Theta(n)$ time. (This one is harder than it looks!)

The point

- Like function calls, method calls can take some time to finish
- For non-trivial data structures/objects (e.g., `ArrayList<Integer> array` rather than `int[] array`), this can hide a lot of computation
- The details depend on:
  - The method's implementation
  - The data structure's current size
- Different data types, different profiles (trade-offs) – later

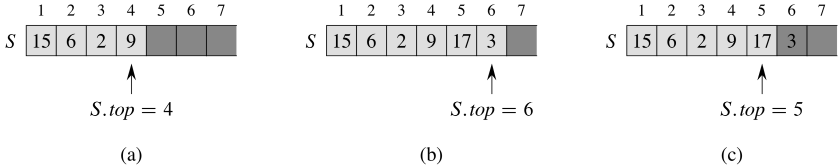
The point

- Like function calls, method calls can take some time to finish
- For non-trivial data structures/objects (e.g., `ArrayList<Integer> array` rather than `int[] array`), this can hide a lot of computation
- The details depend on:
  - The method's implementation
  - The data structure's current size
- Different data types, different profiles (trade-offs) – later

The point

- Like function calls, method calls can take some time to finish
- For non-trivial data structures/objects (e.g., `ArrayList<Integer> array` rather than `int[] array`), this can hide a lot of computation
- The details depend on:
  - The method's implementation
  - The data structure's current size
- Different data types, different profiles (trade-offs) – later

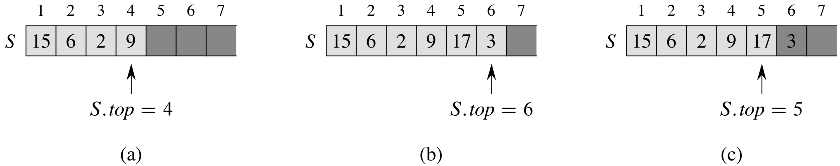
ArrayList – internals



(Picture borrowed from stack implementation)

- Reserved space – contiguous memory block
- Currently used – light gray part
- Add/delete at end – easy (while there's space)
- Add/delete at beginning – must relocate items

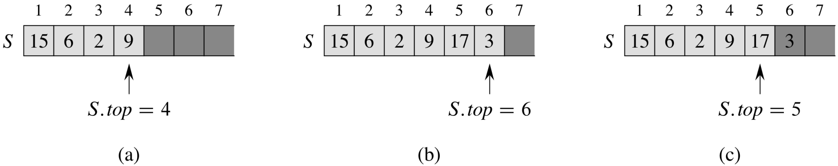
ArrayList – internals



(Picture borrowed from stack implementation)

- Reserved space – contiguous memory block
- Currently used – light gray part
- Add/delete at end – easy (while there's space)
- Add/delete at beginning – must relocate items

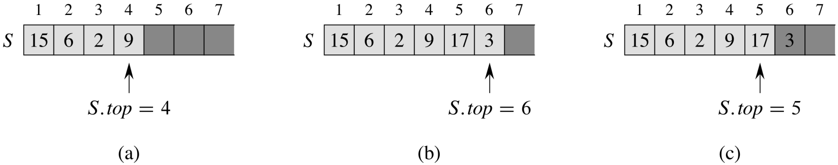
ArrayList – internals



(Picture borrowed from stack implementation)

- Reserved space – contiguous memory block
- Currently used – light gray part
- Add/delete at end – easy (while there's space)
- Add/delete at beginning – must relocate items

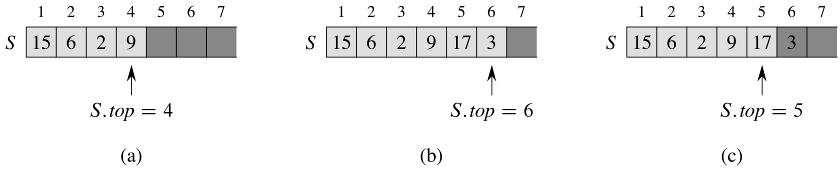
ArrayList – internals



(Picture borrowed from stack implementation)

- Reserved space – contiguous memory block
- Currently used – light gray part
- Add/delete at end – easy (while there's space)
  - $\mathcal{O}(1)$ time to move "top" pointer
- Add/delete at beginning – must relocate items

ArrayList – internals



(Picture borrowed from stack implementation)

- Reserved space – contiguous memory block
- Currently used – light gray part
- Add/delete at end – easy (while there's space)
  - $\mathcal{O}(1)$ time to move "top" pointer
- Add/delete at beginning – must relocate items
  - $\mathcal{O}(s)$ if array contains s elements

Example 1, revisited

Main loop only:

```
for (int i=0; i<n; i++) {
  array.add(0,i);
}
```

- First loop: array empty; single step
- Second loop: one item moved; two steps
- ...
- Loop i: i items moved; $i+1$ steps
- ...
- Loop $n-1$: $n-1$ items moved; n steps

Steps performed: $1 + 2 + \ldots + n = \Theta(n^2)$

Example 1, revisited

Main loop only:

```
for (int i=0; i<n; i++) {
  array.add(0,i);
}
```

- First loop: array empty; single step
- Second loop: one item moved; two steps
- $\ldots$
- Loop i: i items moved; $i+1$ steps
- $\ldots$
- Loop $n-1$: $n-1$ items moved; n steps

Steps performed: $1 + 2 + \ldots + n = \Theta(n^2)$

Contrasting examples

Example 3a:

```
// array is ArrayList
// starts out empty

for (i=0; i<n; i++) {
  array.add(0,i);
}
for (i=0; i<n; i++) {
  array.remove(0);
}
```

Example 3b:

```
// array is ArrayList
// starts out empty

for (i=0; i<n; i++) {
  array.add(0,i);
  array.remove(0);
}
```

Contrasting examples

Example 3a:

```
// array is ArrayList
// starts out empty

for (i=0; i<n; i++) {
  array.add(0,i);
}
for (i=0; i<n; i++) {
  array.remove(0);
}
```

Array grows to size n; operations take $\Theta(n)$ time each; total time $\Theta(n^2)$.

Example 3b:

```
// array is ArrayList
// starts out empty

for (i=0; i<n; i++) {
  array.add(0,i);
  array.remove(0);
}
```

Contrasting examples

Example 3a:

```
// array is ArrayList
// starts out empty

for (i=0; i<n; i++) {
  array.add(0,i);
}
for (i=0; i<n; i++) {
  array.remove(0);
}
```

Array grows to size n; operations take $\Theta(n)$ time each; total time $\Theta(n^2)$.

Example 3b:

```
// array is ArrayList
// starts out empty

for (i=0; i<n; i++) {
  array.add(0,i);
  array.remove(0);
}
```

Array stays at constant size; operations take constant time each; total time $\Theta(n)$.

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n - i)$ time
- Get/set at position i: $\Theta(1)$ time

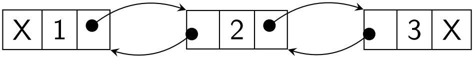
More operations

Assuming standard version of ArrayList; already contains n items.

- Add new item at start: $\Theta(n)$ time
- Delete item at start: $\Theta(n)$ time
- Add new item at end: $\Theta(1)$ time (unless list is "full")
- Delete item at end: $\Theta(1)$ time
- Add/delete at position i: $\Theta(n-i)$ time
- Get/set at position i: $\Theta(1)$ time

Linked Lists

Linked structures



- Structures using nodes and pointers
- Recall linked lists (CS 1801)
- Instead of elements being “lined up” next to each other as in an array, every “location” is a node, containing information (pointer) on how to find the next part.

Linked lists

Advantages and disadvantages

- Disadvantage: Harder to navigate
  - No random access: To find item number 5, first find item 1, then 2, then 3...
  - Not local in memory: Probably slower code
- Advantage: Easier to modify and extend
  - Insert/delete in constant time
  - Can even delete items “from the middle” (with a node pointer)

Linked lists: Structure

Central class: `ListNode`.

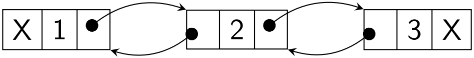
```
class ListNode {
  int data;  // The node contents (String, Object, ...)
  ListNode previous;  // Pointers backwards,
  ListNode next;      // forwards.
}
```

Wrapper class:

```
class LinkedList {
  ListNode firstNode;
  ListNode lastNode;   // lastNode optional
}
```

Linked list: Drawings

Example linked list (three ListNodes, no LinkedList wrapper):



With `ListNode` variables *a*, *b*, *c*:

a.data=1	b.data=2	c.data=3
a.prev=null	b.prev=a	c.prev=b
a.next=b	b.next=c	c.next=null

This is the list [1, 2, 3] as (double) linked list.

List iteration

Typical code for iterating through a list:

```
ListNode node = list.firstNode;
while (node != null) {
   // operate on node
   // then finish with:
   node = node.next;
}
```

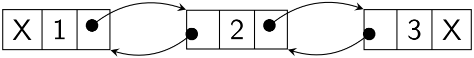
List iteration

Example: searching through a list.

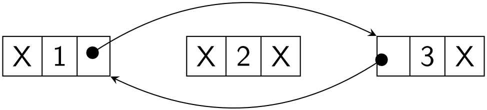
```
boolean listSearch(LinkedList list, int query)
{
   ListNode node = list.firstNode;
   while (node != null) {
      if (node.data == query)
         return true;
      node = node.next;
   }
   // Exit loop: reached pointer "null", so key not found.
   return false;
}
```

Deleting a node

Assume we want to delete the node '2' from this list:



Then we want the result to look like this:



Delete: Bypassing a node

We can "bypass" (delete) a node by making only four changes:

- `node.prev.next = node.next;`
- `node.next.prev = node.prev;`
- `node.prev = null;`
- `node.next = null;`

(Slightly different if removing first/last item.)

Compare to array data types: To delete in the middle, must copy every single item afterwards to a new place ($O(n)$ work)

Insert new node

Very similar: Insert *node2* after *node1*:

- `node2.next=node1.next;`
- `node2.prev=node1;`
- `node1.next.prev=node2;`
- `node1.next=node2;`

(Slightly different if *node1* is the last node.)

A slight problem

Where did you find this *node* pointer?

- Claim: The following operations can be performed in $O(1)$ (i.e., constant) time:
  - `insertFirst`, `insertLast`
  - `deleteFirst`, `deleteLast`
  - `deleteNode`, if you already have the *node* pointer
- But the following take $O(n)$ (linear) time:
  - Locate an item by value (e.g., query)
  - Access the i:th item in the lists
  - Delete an item by value

A slight problem

Where did you find this *node* pointer?

- Claim: The following operations can be performed in $O(1)$ (i.e., constant) time:
  - `insertFirst`, `insertLast`
  - `deleteFirst`, `deleteLast`
  - `deleteNode`, if you already have the *node* pointer
- But the following take $O(n)$ (linear) time:
  - Locate an item by value (e.g., query)
  - Access the i:th item in the lists
  - Delete an item by value

A slight problem

Where did you find this *node* pointer?

- Claim: The following operations can be performed in $O(1)$ (i.e., constant) time:
  - `insertFirst`, `insertLast`
  - `deleteFirst`, `deleteLast`
  - `deleteNode`, if you already have the *node* pointer
- But the following take $O(n)$ (linear) time:
  - Locate an item by value (e.g., query)
  - Access the i:th item in the lists
  - Delete an item by value

Summary

- Linked lists allow constant time insert/delete first/last operations, some list modifications
- However, have drawbacks:
  - No random access
  - Not local in memory – slow traversal
  - Uses more memory
- Remark: Can get on average constant time insert/delete first/last operations on arrays using two tricks:
  1. Circular buffers
  2. The "doubling allocation" trick

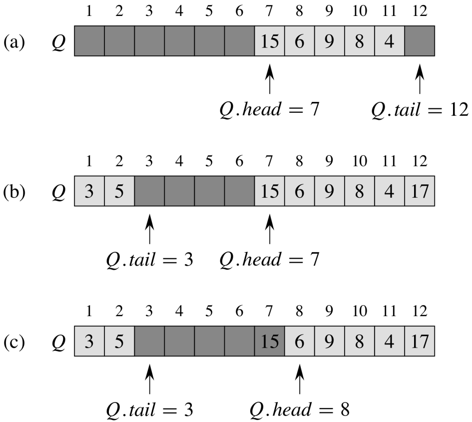
Summary

- Linked lists allow constant time insert/delete first/last operations, some list modifications
- However, have drawbacks:
  - No random access
  - Not local in memory – slow traversal
  - Uses more memory
- Remark: Can get on average constant time insert/delete first/last operations on arrays using two tricks:
  1. Circular buffers
  2. The "doubling allocation" trick

Summary

- Linked lists allow constant time insert/delete first/last operations, some list modifications
- However, have drawbacks:
  - No random access
  - Not local in memory – slow traversal
  - Uses more memory
- Remark: Can get on average constant time insert/delete first/last operations on arrays using two tricks:
  1. Circular buffers
  2. The “doubling allocation” trick

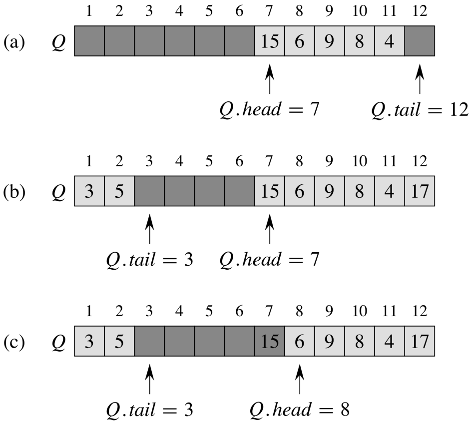
Array-based variant – Circular buffer



Add/delete at start/end in constant time (while there's space)

Warning – implementation is messy

Array-based variant – Circular buffer



Add/delete at start/end in constant time (while there's space)
Warning – implementation is messy

Doubling allocation trick

- Recall – array has fixed size in memory (say n)
- Adding item $n+1$ involves:
  1. Allocate new memory for array of size $n' > n$
  2. Copy old n items to new array ($\Theta(n)$ work)
  3. Finally add item $n+1$
- How much work in total for adding n items to an empty array?
  - If $n' = n+1$:
  - If $n' = n+1000$
  - If $n' = 2n$

Doubling allocation trick

- Recall – array has fixed size in memory (say n)
- Adding item $n+1$ involves:
  1. Allocate new memory for array of size $n' > n$
  2. Copy old n items to new array ($\Theta(n)$ work)
  3. Finally add item $n+1$
- How much work in total for adding n items to an empty array?
  - If $n' = n + 1$:
  - If $n' = n + 1000$
  - If $n' = 2n$

Doubling allocation trick

- Recall – array has fixed size in memory (say n)
- Adding item $n+1$ involves:
    1. Allocate new memory for array of size $n' > n$
    2. Copy old n items to new array ($\Theta(n)$ work)
    3. Finally add item $n+1$
- How much work in total for adding n items to an empty array?
    - If $n' = n+1$:
    - If $n' = n+1000$
    - If $n' = 2n$

Doubling allocation trick

- Recall – array has fixed size in memory (say n)
- Adding item $n+1$ involves:
  1. Allocate new memory for array of size $n' > n$
  2. Copy old n items to new array ($\Theta(n)$ work)
  3. Finally add item $n+1$
- How much work in total for adding n items to an empty array?
  - If $n' = n+1$:
  - If $n' = n+1000$
  - If $n' = 2n$

Doubling allocation trick

- Idea: Double the size of the array each time it gets full.
- Suppose you start with an empty array and add $N = 2^{n-1}$ items. Then the total amount of work spent reallocating memory is

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1 = \mathcal{O}(N)$$

- So the average amount of work per item added is $\mathcal{O}(1)$.

Doubling allocation trick

- Idea: Double the size of the array each time it gets full.
- Suppose you start with an empty array and add $N = 2^{n-1}$ items. Then the total amount of work spent reallocating memory is

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1 = \mathcal{O}(N)$$

- So the average amount of work per item added is $\mathcal{O}(1)$.

Doubling allocation trick

- Idea: Double the size of the array each time it gets full.
- Suppose you start with an empty array and add $N = 2^{n-1}$ items. Then the total amount of work spent reallocating memory is

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1 = \mathcal{O}(N)$$

- So the average amount of work per item added is $\mathcal{O}(1)$.

Comparison

Assume the array/lists already contain n items:

Operation	Time (array)	Time (linked list)
Add at 0		
Delete 0		
Add at end		
Delete from end		
Random access		
Delete in middle		

Comparison

Assume the array/lists already contain n items:

Operation	Time (array)	Time (linked list)
Add at 0	$\Theta(n)$	$\Theta(1)$
Delete 0		
Add at end		
Delete from end		
Random access		
Delete in middle		

Comparison

Assume the array/lists already contain n items:

Operation	Time (array)	Time (linked list)
Add at 0	$\Theta(n)$	$\Theta(1)$
Delete 0	$\Theta(n)$	$\Theta(1)$
Add at end		
Delete from end		
Random access		
Delete in middle		

Comparison

Assume the array/lists already contain n items:

Operation	Time (array)	Time (linked list)
Add at 0	$\Theta(n)$	$\Theta(1)$
Delete 0	$\Theta(n)$	$\Theta(1)$
Add at end	$\Theta(1)$ unless full	$\Theta(1)$
	$\Theta(n)$ if full (rarely)	
Delete from end		
Random access		
Delete in middle		

Comparison

Assume the array/lists already contain n items:

Operation	Time (array)	Time (linked list)
Add at 0	$\Theta(n)$	$\Theta(1)$
Delete 0	$\Theta(n)$	$\Theta(1)$
Add at end	$\Theta(1)$ unless full	$\Theta(1)$
	$\Theta(n)$ if full (rarely)	
Delete from end	$\Theta(1)$	$\Theta(1)$
Random access		
Delete in middle		

Comparison

Assume the array/lists already contain n items:

Operation	Time (array)	Time (linked list)
Add at 0	$\Theta(n)$	$\Theta(1)$
Delete 0	$\Theta(n)$	$\Theta(1)$
Add at end	$\Theta(1)$ unless full	$\Theta(1)$
	$\Theta(n)$ if full (rarely)	
Delete from end	$\Theta(1)$	$\Theta(1)$
Random access	$\Theta(1)$	$\mathcal{O}(n)$
Delete in middle		

Comparison

Assume the array/lists already contain n items:

Operation	Time (array)	Time (linked list)
Add at 0	$\Theta(n)$	$\Theta(1)$
Delete 0	$\Theta(n)$	$\Theta(1)$
Add at end	$\Theta(1)$ unless full	$\Theta(1)$
	$\Theta(n)$ if full (rarely)	
Delete from end	$\Theta(1)$	$\Theta(1)$
Random access	$\Theta(1)$	$\mathcal{O}(n)$
Delete in middle	$\Theta(n)$	$\Theta(1)$ with pointer
		$\Theta(n)$ without pointer