Lecture 7

Algorithms & Data Structures

Goldsmiths Computing

November 19, 2018

Outline

Outline

Lecture

- Binary search trees
  - contents property
  - improved find
- Mergesort
- Master theorem

Lab

- Be a data structure implementor
  1. even more methods on linked lists
  2. understand how mergesort behaves
- Use data structures
  1. populate stack and queue appropriately

VLE activities

Dynamic arrays quiz

Statistics so far:

- 242 attempts: average mark 4.36
- 98 students: average mark 4.18
  - 57 under 4.00, 21 over 6.99, 7 at 10.00

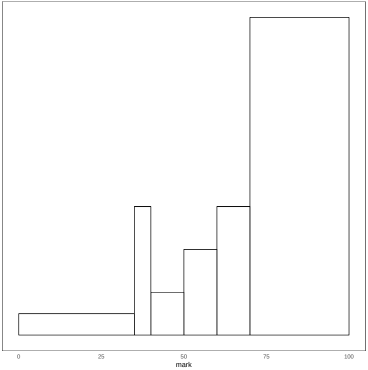
Quiz closes at 16:00 on Friday 23rd November

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

VLE activities (cont'd)

Mergesort submission

- 126 final uploads: average mark 83.06



Outline

Motivation

We have seen a number of data structures for storing data by now. Is there a unifying concept behind storing data items?

Definition

collection a grouping of some variable number of data items. aka: “container” (C++)

linear collection a collection with an underlying linear order

collection	linear?
linked list	✓
dynamic array	✓
binary tree	?
set	✗
multiset	✗
stack	✓
queue	✓
priority queue	✓
deque	✓

Operations

Generic collection

size	how many elements does the collection contain?
insert[o]	add o to the collection
find[o]	is the object o in the collection?
remove[o]	return a collection with all instances of o removed
count[o]	how many times is o stored in the collection?
sum	what is the sum of the objects in the collection?

Operations

Generic collection

size	how many elements does the collection contain?
insert[o]	add o to the collection
find[o]	is the object o in the collection?
remove[o]	return a collection with all instances of o removed
count[o]	how many times is o stored in the collection?
sum	what is the sum of the objects in the collection?
iterate[f]	visit all items of the collection, calling f on each item

Operations

Generic collection

size	how many elements does the collection contain?
insert[o]	add o to the collection
find[o]	is the object o in the collection?
remove[o]	return a collection with all instances of o removed
count[o]	how many times is o stored in the collection?
sum	what is the sum of the objects in the collection?
iterate[f]	visit all items of the collection, calling f on each item

Linear collection

position[o]	what index is o at, if any?
get[i]	get the object at index i

Work

1. Reading
   - Drozdek [C++], section 1.7.1 (Containers), 3.7 (Lists in the STL), 4.4-4.7 (Stacks, Queues, Priority Queues, Deques in the STL)
   - Drozdek [Java], section 1.5 (Vectors in `java.util`), 3.7 (Lists in `java.util`), 4.1.1 (Stacks in `java.util`)

Outline

Introduction

Collections

Hash tables

Cycle detection

Motivation

A different way to implement a collection, with different performance implications

Definition

A hash table is a data structure that can represent a set, or more generally a map of keys to values (an associative array), by computing a numeric value for each key using a hash function and then using that numeric value to compute an index into an array to look up the value.

Set operations

insert[o]	insert the object o into the set
find[o]	is the object o in the set?

and also

delete[o]	delete the object o from the set

Sets of small integers

Represent sets of non-negative integers smaller than N using an array of size N. e.g. for domain [0,5]:

Sets of small integers

Represent sets of non-negative integers smaller than N using an array of size N. e.g. for domain [0,5]:

✓	✗	✗	✓	✗	✓

represents the set $\{0, 3, 5\}$

Sets of small integers

Represent sets of non-negative integers smaller than N using an array of size N. e.g. for domain [0,5]:

✓	✗	✗	✓	✗	✓

represents the set $\{0, 3, 5\}$

insert[o]	S[o] $\leftarrow$ true
find[o]	**return** S[o]
delete[o]	S[o] $\leftarrow$ false

Sets of unbounded integers

Apply the same representation?

Sets of unbounded integers

Apply the same representation?

✓	✗	✗	✓	✗	✓	⋯											

2^{32} integers? 2^{29} bytes of RAM (512MB)

Sets of unbounded integers

If the expected size of the sets is small (even if the range of possible values is large):

1. choose a reasonable size for the array, say twice expected size
2. reduce the integer to within the range of array indices using a function f(n)
3. store the (unreduced) integer in the array slot

Then

insert[o] $S[f(o)] \leftarrow o$

find[o] **return** $S[f(o)] = o$

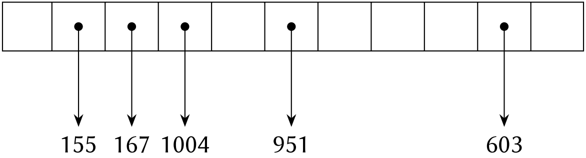
delete[o] $S[f(o)] \leftarrow \mathrm{NIL}$

Example

range $[0,2^{10})$

set size 5

Choose array size of (say) 11 and compute index as f(n) = n mod 11



represents the set {155, 167, 603, 951, 1004}

Complexity analysis

Provided the reducing function f(n) is $\Theta(1)$

insert

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\implies \Theta(1)$$

Complexity analysis

Provided the reducing function f(n) is $\Theta(1)$

insert

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\implies \Theta(1)$$

find

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\implies \Theta(1)$$

Complexity analysis

Provided the reducing function f(n) is $\Theta(1)$

insert

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\implies \Theta(1)$$

find

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\implies \Theta(1)$$

delete

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\implies \Theta(1)$$

Complexity analysis

Provided the reducing function f(n) is $\Theta(1)$

insert

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\Longrightarrow \Theta(1)$$

find

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\Longrightarrow \Theta(1)$$

delete

$\Theta(1)$ reduction and $\Theta(1)$ memory operations

$$\Longrightarrow \Theta(1)$$

So what am I not telling you?

Sets of arbitrary things

- compute an integer (a *hash code*) for the things using a *hash function*
  - equal things must have equal hash codes
  - unequal things should be unlikely to share hash codes

computing an integer for the things:

Java `public int hashcode()`

C++ `operator()` functor second template argument to container

equal things must have equal integer codes:

Java `public boolean equals(Object o)`

C++ `operator()` functor third template argument to container

Work

1. Reading
   - CLRS, sections 11.1 and 11.2
   - DPV, section 1.5
   - Drozdek, sections 10.1

Outline

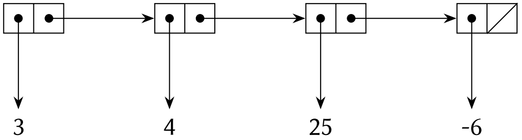
Motivation

Did your SLList code suffer from baffling infinite loops at any point?

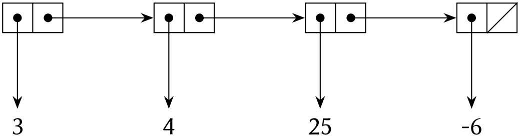
Definition

Cycle detection algorithms detect whether there are loops in a graph, graph, or repeated values in a function with same domain and range, and where and how long those loops are.

Linked list implementation

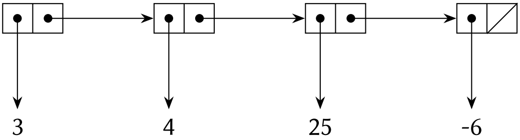


Linked list implementation

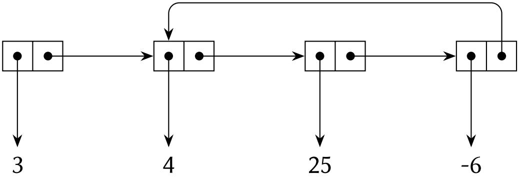


SET-REST!(REST(REST(REST(list))), REST(list))

Linked list implementation



SET-REST!(REST(REST(REST(list))), REST(list))



Naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Helper function

```
function IS-IN?(sequence,node,end)
    j ← 0
    x ← sequence
    while j < end do
        if x = node then
            return true
        end if
        x ← REST(x)
        j ← j + 1
    end while
    return false
end function
```

Naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Main function

Require: sequence :: list
  node ← REST(sequence)
  end ← 0
  while node ≠ NIL **do**
    end ← end + 1
    if IS-IN?(sequence,node,end) **then**
      return true
    end if
    node ← REST(node)
  end while
  return false

Naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Complexity analysis

Space

No extra space required

$$\implies \Theta(1)$$

Time

$$T_k = T_{k-1} + \Theta(k)$$

If there is no cycle, the algorithm traverses the entire list, checking an increasing amount of the entire list each time

$$\implies \Theta(N^2)$$

If there is a cycle, the algorithm stops at the first repeated node after once round the cycle

$$\implies \Theta((j + l)^2)$$

Less naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Hash-table memory

Require: sequence :: list
  table $\leftarrow$ new Hashtable
  node $\leftarrow$ sequence
  while node $\neq$ NIL **do**
    if node $\in$ table **then**
      return true
    end if
    table[node] $\leftarrow$ true
  end while
  return false

Less naïve circularity detection

Algorithm: at each step, check all previous steps for repeat

Complexity analysis

Space

Hash table with N entries required

$$\implies \Theta(N) \text{ extra space}$$

Time

$$T_k = T_{k-1} + \Theta(1)$$

If there is no cycle, the algorithm traverses the entire list, doing a constant-time lookup each time

$$\implies \Theta(N)$$

If there is a cycle, the algorithm stops at the first repeated node

$$\implies \Theta(j + l)$$

(assumes hash-table lookup is $\Theta(1)$)

Hare and tortoise

Also known as Floyd's cycle-finding algorithm

Key insight

for circularity of length l beginning at position j

$$L[k + nl] = L[k]$$

for all $k > j, n \geq 0$.

Hare and tortoise

Also known as Floyd's cycle-finding algorithm

Key insight

for circularity of length l beginning at position j

$$L[k + nl] = L[k]$$

for all $k > j, n \geq 0$.

... or in words

If two nodes at different positions in the list are identical, the difference in positions is an integer multiple of the circularity length.

Hare and tortoise

Also known as Floyd's cycle-finding algorithm

Key insight

for circularity of length l beginning at position j

$$L[k + nl] = L[k]$$

for all $k > j, n \geq 0$.

... or in words

If two nodes at different positions in the list are identical, the difference in positions is an integer multiple of the circularity length.

Converse

If there is a circularity and two iterators are each within it, incrementing the *difference* between two list iterators by 1 will always lead to the two iterators arriving at the same list node.

Hare and tortoise

Also known as Floyd's cycle-finding algorithm

Algorithm

Require: sequence :: list
  tortoise $\leftarrow$ REST(sequence)
  hare $\leftarrow$ REST(tortoise)
  while hare $\neq$ NIL **do**
    if hare = tortoise **then**
      return true
    end if
    tortoise $\leftarrow$ REST(tortoise)
    hare $\leftarrow$ REST(REST(hare))
  end while
  return false

Hare and tortoise

Also known as Floyd’s cycle-finding algorithm

Complexity analysis

Space

No extra space needed

$$\implies \Theta(1)$$

Time

If there is no cycle, the hare traverses the list; in that time, the tortoise traverses half the list:

$$\implies \Theta(N)$$

If there *is* a cycle, the hare and tortoise meet no more than l steps after the tortoise is in the cycle

$$\implies \Theta(j + l)$$

Additional information from algorithm

Position of first repeat

1. reset tortoise to the head of the list
2. move hare and tortoise one step at a time (same speed)
3. count steps until hare and tortoise are equal

Additional information from algorithm

Position of first repeat

1. reset tortoise to the head of the list
2. move hare and tortoise one step at a time (same speed)
3. count steps until hare and tortoise are equal

Length of circularity

1. hold tortoise still
2. move hare one step at a time
3. count steps until hare and tortoise are equal again