Apr 25, 2024

Lecture 2: Data Structures

Data Structure Interfaces

- A **data structure** is a way to store data, with algorithms that support **operations** on the data
- Collection of supported operations is called an **interface** (also API or ADT)
- Interface is a **specification**: what operations are supported (the problem!)
- Data structure is a **representation**: how operations are supported (the solution!)
- In this class, two main interfaces: **Sequence** and **Set**

Sequence Interface (L02, L07)

→ can be array based or pointer based

- Maintain a sequence of items (order is **extrinsic**)
- Ex: $(x_0, x_1, x_2, \ldots, x_{n-1})$ (zero indexing)
- (use n to denote the number of items stored in the data structure)
- Supports sequence operations:

fixed size at build

Container operation	`build(X)` `len()`	given an iterable `X`, build sequence from items in `X` return the number of stored items

Static
operations
	`iter_seq()`
	`get_at(i)`
	`set_at(i, x)`
		return the stored items one-by-one in sequence order
		return the i^{th} item
		replace the i^{th} item with x

Dynamic
operations
	`insert_at(i, x)`
	`delete_at(i)`
	`insert_first(x)`
	`delete_first()`
	`insert_last(x)`
	`delete_last()`
		add x as the i^{th} item
		remove and return the i^{th} item
		add x as the first item
		remove and return the first item
		add x as the last item
		remove and return the last item

} in real life a data structure that implements this interface is a "Static Array"
↓
in Python, closest is list( ) although that's a dynamic array

efficient w/:
① Linked List
② Dynamic Array

- Special case interfaces:

stack	`insert_last(x)` and `delete_last()`
queue	`insert_last(x)` and `delete_first()`

Set Interface (L03-L08)

- Sequence is about **extrinsic** order, set is about **intrinsic** order
- Maintain a set of items having **unique keys** (e.g., item `x` has key `x.key`)
- (Set or multi-set? We restrict to unique keys for now.)
- Often we let key of an item be the item itself, but may want to store more info than just key
- Supports set operations:

Container	`build(X)`	given an iterable `X`, build sequence from items in `X`
	`len()`	return the number of stored items
Static	`find(k)`	return the stored item with key `k`
Dynamic	`insert(x)`	add `x` to set (replace item with key `x.key` if one already exists)
	`delete(k)`	remove and return the stored item with key `k`
Order	`iter_ord()`	return the stored items one-by-one in key order
	`find_min()`	return the stored item with smallest key
	`find_max()`	return the stored item with largest key
	`find_next(k)`	return the stored item with smallest key larger than `k`
	`find_prev(k)`	return the stored item with largest key smaller than `k`

- Special case interfaces:

dictionary	set without the Order operations

- In recitation, you will be asked to implement a Set, given a Sequence data structure.

* "Array" = consecutive chunk of "virtual" memory
↳ recall Word-RAM model

"Static" Array Sequence

- Array is great for static operations! `get_at(i)` and `set_at(i, x)` in $\Theta(1)$ time!
↳ and len()
- But not so great at dynamic operations...
- (For consistency, we maintain the invariant that array is full)
- Then inserting and removing items requires:
  - reallocating the array → cost $\Theta(n)$ time ... space complexity < time complexity
  - shifting all items after the modified item

assumes that the machine-word size w:
$w \geq \log n$
↳ says that we should be able to address n data
(usually w = 64-bit)

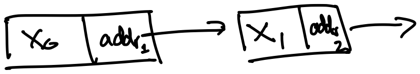
Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Array	n	1	n	n	n

O(n) per build(), iter_seq()

↳ note, fixed size ...

↳ constant time complexity

since requires allocating new array w/ bigger size to maintain contiguous mem chunk



Linked List Sequence

- Pointer data structure (this is **not** related to a Python "list")
- Each item stored in a **node** which contains a pointer to the next node in sequence
- Each `node` has two fields: `node.item` and `node.next`
- Can manipulate nodes simply by relinking pointers!
- Maintain pointers to the first node in sequence (called the head)
- Can now insert and delete from the front in $\Theta(1)$ time! Yay!
- (Inserting/deleting efficiently from back is also possible; you will do this in PS1)
- But now `get_at(i)` and `set_at(i, x)` each take $O(n)$ time... :(
- Can we get the best of both worlds? Yes! (Kind of...)

Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Linked List	n	n	1	n	n

alloc ↓ $2 \cdot n$ nodes

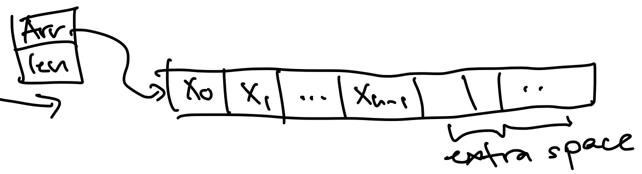
↓ need to traverse

↓ only require s alloc $\boxed{x_0 | addr}$ and have it point to orig first node...

need to traverse linked list...

Dynamic Array Sequence

- Make an static array efficient for **last** dynamic operations → insert_last, delete_last } we want $O(1)$
- Python "list" is a dynamic array
- **Idea!** Allocate extra space so reallocation does not occur with every dynamic operation
  - ↳ since in static array, we need to change location of other elem's ~~when~~ inserting/deleting... ↳ everytime ↓ w/ dynamic arr, we only do this whenever $r=1$
  - relax constraint that $size(arr) = n$ ↓ $size = \Theta(n)$ i.e $= c \cdot n$
- **Fill ratio:** $0 \le r \le 1$ the ratio of items to space
- Whenever array is full ($r = 1$), allocate $\Theta(n)$ extra space at end to fill ratio r_i (e.g., $1/2$)
- Will have to insert $\Theta(n)$ items before the next reallocation ↳ linear cost
- A single operation can take $\Theta(n)$ time for reallocation
- However, any sequence of $\Theta(n)$ operations takes $\Theta(n)$ time ~~total~~.
- amortize ↳ So each operation takes $\Theta(1)$ time "on average"



⇒ as long as you have buffer insert_last = $A[len] = x$, $len \mathrel{+}= 1$

unless $n = size$, then make array bigger

↳ how big?

↳ one possibility: $size = 2 \cdot size$

$$\text{cost} = \Theta\left(\sum_{i=0}^{\lg n} 2^i\right) = \Theta(2^{\lg n}) \leftarrow \text{geom seq} = \Theta(n)$$

particular kind of averaging over seq of operations

Amortized Analysis

- Data structure analysis technique to distribute cost over many operations
- Operation has **amortized cost** $T(n)$ if k operations cost at most $\leq kT(n)$
- "$T(n)$ amortized" roughly means $T(n)$ "on average" over many operations
- Inserting into a dynamic array takes $\mathbf{\Theta(1)}$ **amortized** time
- More amortization analysis techniques in 6.046!

Dynamic Array Deletion

- Delete from back? $\Theta(1)$ time without effort, yay!
- However, can be very wasteful in space. Want size of data structure to stay $\Theta(n)$
- **Attempt:** if very empty, resize to $r = 1$. Alternating insertion and deletion could be bad...
- **Idea!** When $r < r_d$, resize array to ratio r_i where $r_d < r_i$ (e.g., $r_d = 1/4$, $r_i = 1/2$)
- Then $\Theta(n)$ cheap operations must be made before next expensive resize
- Can limit extra space usage to $(1+\varepsilon)n$ for any $\varepsilon > 0$ (set $r_d = \frac{1}{1+\varepsilon}$, $r_i = \frac{r_d+1}{2}$)
- Dynamic arrays only support dynamic **last** operations in $\Theta(1)$ time
- Python List `append` and `pop` are amortized $O(1)$ time, other operations can be $O(n)$!
- (Inserting/deleting efficiently from front is also possible; you will do this in PS1)

Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n

↳amortized

good for random access

great for working @