Recap

Key points –

- the container ADTs Vector/List/Sequence, Stack, Queue
  - characterization and typical operations
- the data structures array and linked list
  - characteristics and tradeoffs
- using arrays and linked lists to implement Queue, Stack
  - explain how the elements in the queue/stack are arranged in the array/linked list
    - both queue/stack and array/linked list have a linear order
    - identify which end of the array/linked list corresponds to the beginning/top of queue/stack
  - carrying out insert, remove operations
- improving implementations
  - a strategy: store instead of computing
    - e.g. tail pointer, circular array
  - but: have to make sure that maintenance of the stored information doesn't increase the big-Oh

---

How Do We Apply This Stuff?

- ADTs
  - algorithm may boil down to just manipulating the right ADT, or become much simpler with the right ADT
  - once you have an algorithm, identify the operations it needs
  - find a standard ADT that provides those operations (and ideally little else) and choose an efficient implementation, or design a new implementation to efficiently support those operations
- data structures
  - to choose an efficient implementation for standard ADT
  - to design your own data structure or customize a standard implementation if a standard ADT/implementation doesn't meet your needs
- why study different implementations?
  - often not a single best choice – tradeoffs mean making one operation faster can make another slower

---

Containers in Java

ADT	**in Java**
Vector / List / Sequence	`List` – interface `LinkedList` – linked list implementation `ArrayList` – array implementation `Vector` – legacy class and use is discouraged (array implementation) `Deque` (double-ended queue) – interface `ArrayDeque` – array implementation `LinkedList` – linked list implementation
Stack	`Stack` – legacy class, `Deque` preferred
Queue	`Queue` – interface `ArrayDeque` – array implementation `LinkedList` – linked list implementation

Collections Framework overview:
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/doc-files/coll-overview.html

---

ADTs – Map/Dictionary and Set

- searching and lookup

Map / Dictionary variations • **OrderedDictionary** – also supports min/max, predecessor(k)/ successor(k) based on an ordering of the keys	lookup (no duplicate keys)	• find(k) – find elt with key k if it exists • insert(k,v) – add elt v with key k • delete(k) – remove elt, key with key k (may return elt)
Set	membership (no duplicate elements)	• add(x) – add elt x if not already present • remove(x) – remove elt x • contains(x) – return whether x is present

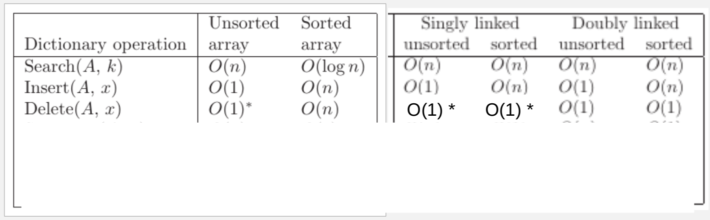
ADTs for Algorithm Design

The ordering of elements imposed by different types of containers can be exploited to achieve algorithmic goals.

ADT	some applications of the ADT
Vector / List / Sequence	general-purpose container round-robin scheduling, taking turns
Stack	match most recent thing, proper nesting, reversing DFS – go deep before backing up has ties to recursive procedures – supports iterative implementation of recursive ideas
Queue	FIFO order minimizes waiting time BFS – spread out in levels round-robin scheduling, taking turns
Map Set	duplicate removal, set union – look up each new element in collection of already-seen ones
OrderedDictionary	sorting – insert elements, then go through keys in order

---

Map/Dictionary

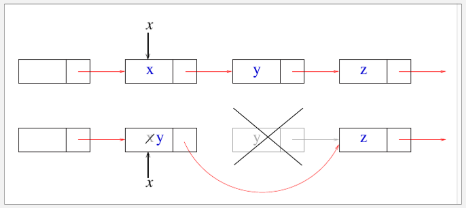


Dictionary operation	Unsorted array	Sorted array	Singly linked unsorted	Singly linked sorted	Doubly linked unsorted	Doubly linked sorted
Search(A, k)	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(A, x)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Delete(A, x)	$O(1)^*$	$O(n)$	O(1) *	O(1) *	$O(1)$	$O(1)$

delete operation as defined in ADM assumes that the element is already found (known array index, pointer to the linked list node) – otherwise find operation is required first

* denotes cleverness or subtlety

Constant-Time Deletion in a Singly-Linked List



- O(1) deletion

```
x.setValue(x.getNext().getValue())
x.setNext(x.getNext().getNext())
```

Map/Dictionary

- basic container is a Vector/List/Sequence
  - choice of array or linked list implementation depends on which operations are used
- ordering of elements within Sequence is up to the Map – can be sorted or not
  - unsorted leads to O(1) insert/delete but O(n) search for both arrays and linked lists
  - sorted leads to differences between arrays and linked lists
    - O(log n) search and O(n) delete for arrays
    - O(n) search and O(1) delete for linked lists

Can we do better?

- can we exploit the sorted order to improve searching in linked lists?
- O(n) delete in arrays is due to shifting – can't do much about that
  - circular arrays worked for queues because insert/delete was only at the ends

Improving an Implementation – Map

Binary search exploits the sorted order – but it requires efficient random access.

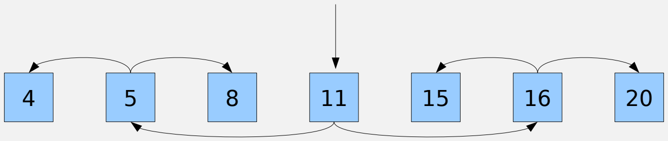
Or does it?

- the first iteration of binary search requires knowing the middle element
- successive iterations require knowing the middle element of one of the halves

Finding the middle element is achieved in arrays by arithmetic involving array indexes, but what if we just stored the necessary info instead?

- store instead of computing…

Doing Better

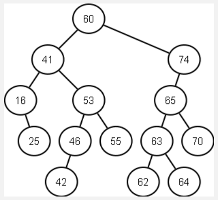


- each middle element only needs to store the location of two other middle elements → binary tree structure
- overall the elements are ordered, so the “other middle elements” are smaller and larger than the “middle element”, respectively → binary search tree

---

Binary Search Trees

- a binary tree with an ordering property for the elements
  - for every node, all of the elements in the left subtree are less than or equal to the node's element and all of the elements in the right subtree are greater than the node's element
- operations
  - find
  - insert
  - remove
  - visit all elements (traverse) in order

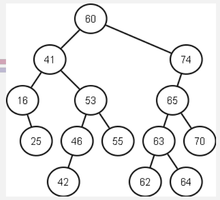


(dummy leaves not shown)

implementation note: every internal node in a *proper binary tree* has exactly two children – for BST, we only store elements at internal nodes

---

Binary Search Trees

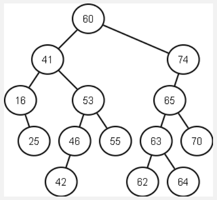


(dummy leaves not shown)

- find
  - moving down, 1-finger (only go to one child) pattern → loop
  - observation: if the element isn't there, search ends at a (dummy) leaf
- insert
  - can only insert at a leaf
  - the correct insertion point is the leaf where an unsuccessful search for the element ends up
- remove
  - can only remove above a leaf
  - if the element to remove does not have at least one leaf child, swap it with a safe element which does has at least one leaf child
    - i.e. the next element larger or smaller than the one to remove

Binary Search Trees

- visit all elements in order
  - moving down, both children pattern → recursion
  - need to visit smaller elements before the current node's element before the larger elements → inorder traversal



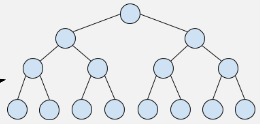
(dummy leaves not shown)

Implementing Map

- can store (key,value) pairs in a binary search tree ordered by key
  - let h be the height of the tree
  - all operations are O(h) as it may be necessary to go from the root all the way down to a leaf

BST Height

- height of a binary search tree
  - best case is O(log n)
  - worst case is O(n)
- whether a BST of a given size is *balanced* (O(log n) height) or unbalanced (O(n) height) depends on the order of insertions and removals, not the elements in the tree
- can we do better?
  - try to keep the tree balanced...



$\sum_{i=0}^{n} 2^i \quad = \Theta(2^n)$