Midterm Review

Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants

Data Structures So Far

- Array List
  - (Extendable) Array
- Node List
  - Singly or Doubly Linked List
- Stack
  - Array
  - Singly Linked List
- Queue
  - Array
  - Singly or Doubly Linked List
- Priority Queue
  - Unsorted doubly-linked list
  - Sorted doubly-linked list
  - Heap (array-based)
- Adaptable Priority Queue
  - Sorted doubly-linked list with location-aware entries
  - Heap with location-aware entries
- Tree
  - Linked Structure
- Binary Tree
  - Linked Structure
  - Array

Topics on the Midterm

- **Data Structures & Object-Oriented Design**
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants

Data Structures & Object-Oriented Design

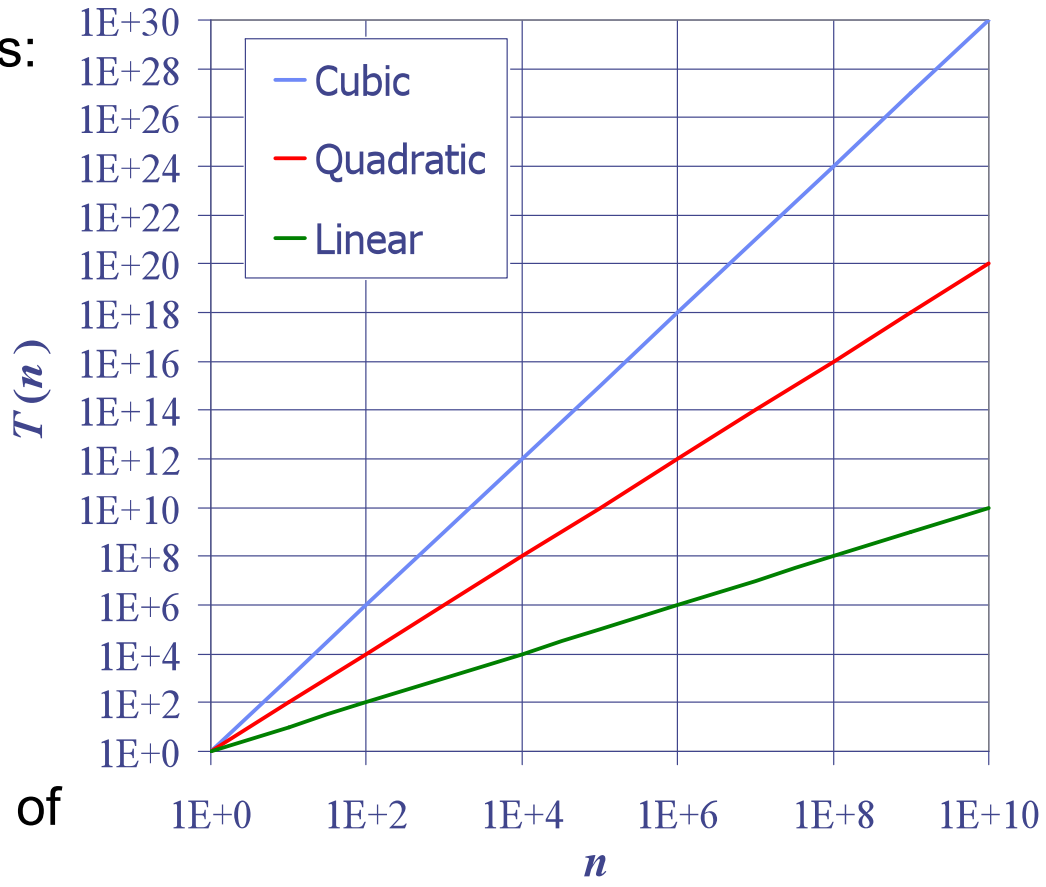
- Definitions
- Principles of Object-Oriented Design
- Hierarchical Design in Java
- Abstract Data Types & Interfaces
- Casting
- Generics
- Pseudo-Code

Software Engineering

- Software must be:
  - Readable and understandable
    - Allows correctness to be verified, and software to be easily updated.
  - Correct and complete
    - Works correctly for all expected inputs
  - Robust
    - Capable of handling unexpected inputs.
  - Adaptible
    - All programs evolve over time. Programs should be designed so that re-use, generalization and modification is easy.
  - Portable
    - Easily ported to new hardware or operating system platforms.
  - Efficient
    - Makes reasonable use of time and memory resources.

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ log n
  - Linear ≈ n
  - N-Log-N ≈ n log n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n



- In a log-log chart, the slope of the line corresponds to the growth rate of the function.

Topics on the Midterm

- Data Structures & Object-Oriented Design
- **Run-Time Analysis**
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants

Some Math to Review

- Summations
- Logarithms and Exponents
- Existential and universal operators
- Proof techniques
- Basic probability

- **existential and universal operators**

$\exists g \forall b \; Loves(b, g)$

$\forall g \exists b \; Loves(b, g)$

➢ **properties of logarithms:**

$\log_b(xy) = \log_b x + \log_b y$

$\log_b (x/y) = \log_b x - \log_b y$

$\log_b x^a = a\log_b x$

$\log_b a = \log_x a / \log_x b$

➢ **properties of exponentials**:

$a^{(b+c)} = a^b a^c$

$a^{bc} = (a^b)^c$

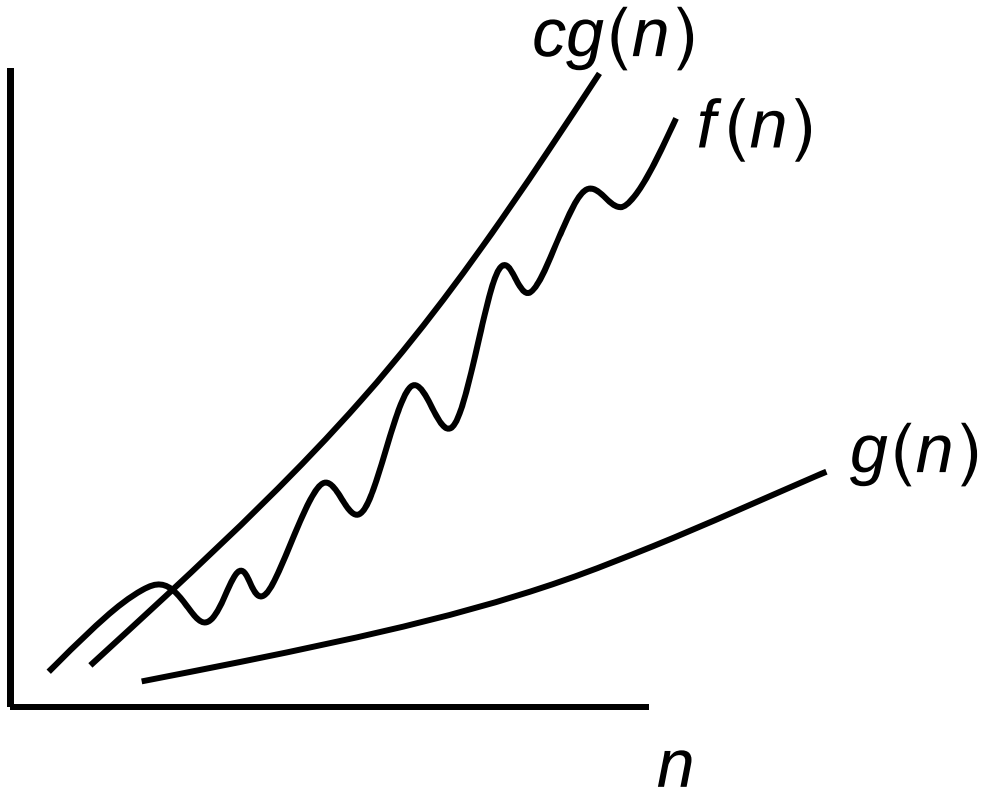
$a^b / a^c = a^{(b-c)}$

$b = a^{\log_a b}$

$b^c = a^{c*\log_a b}$

Definition of "Big Oh"

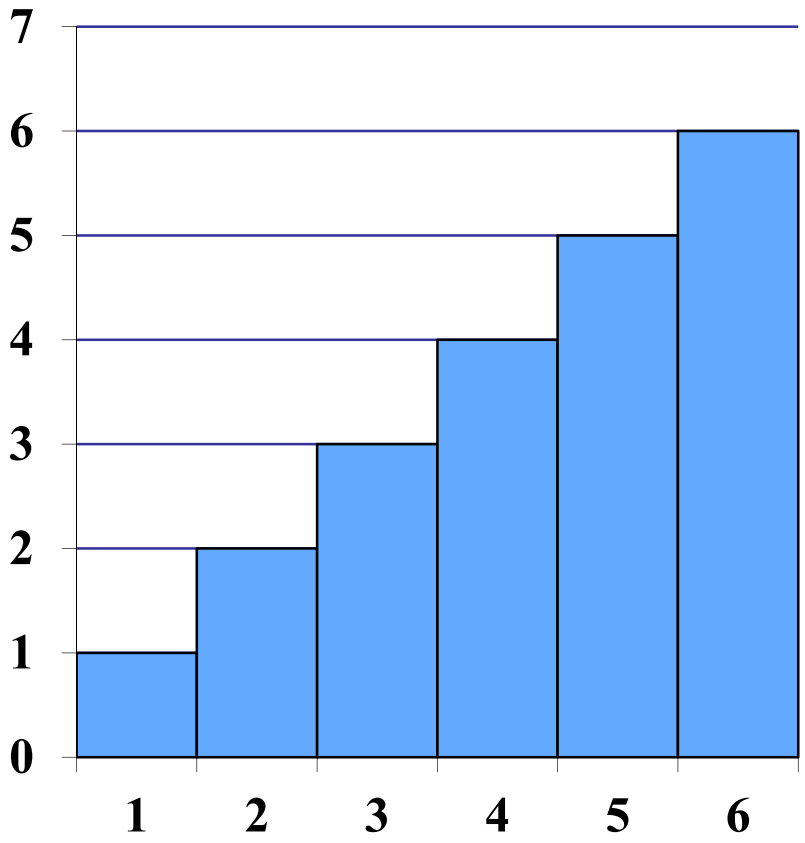
$$f(n) \in O(g(n))$$



$$\exists c, n_0 > 0 : \forall n \geq n_0, f(n) \leq cg(n)$$

Arithmetic Progression

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time



Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that $c_1 \bullet g(n) \leq f(n) \leq c_2 \bullet g(n)$ for $n \geq n_0$

Time Complexity of an Algorithm

The time complexity of an algorithm is the *largest* time required on *any* input of size n. (Worst case analysis.)

- $O(n^2)$: For any input size $n \geq n_0$, the algorithm takes no more than cn^2 time on every input.
- $\Omega(n^2)$: For any input size $n \geq n_0$, the algorithm takes at least cn^2 time on at least one input.
- $\theta(n^2)$: Do both.

Time Complexity of a Problem

The time complexity of a problem is the time complexity of the *fastest* algorithm that solves the problem.

- $O(n^2)$: Provide an algorithm that solves the problem in no more than this time.
  - Remember: for every input, i.e. worst case analysis!
- $\Omega(n^2)$: Prove that no algorithm can solve it faster.
  - Remember: only need one input that takes at least this long!
- $\theta(n^2)$: Do both.

Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- **Linear Data Structures**
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants

Arrays

Arrays

- Array: a sequence of indexed components with the following properties:
  - **array size is fixed** at the time of array's construction
    - **int**[] numbers = **new int** [10];
  - **array elements are placed contiguously** in memory
    - address of any element can be calculated directly as its offset from the beginning of the array
  - consequently, array components **can be efficiently inspected or updated** in O(1) time, using their indices
    - **randomNumber = numbers[5];**
    - **numbers[2] = 100;**

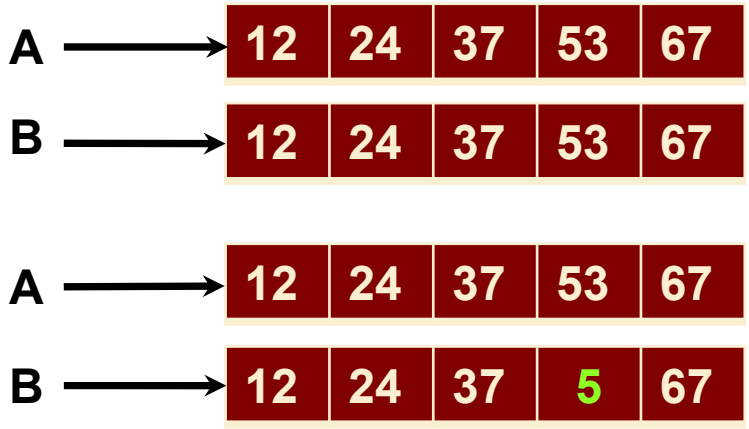
Arrays in Java

- Since an array is an object, the name of the array is actually a **reference** (pointer) to the place in memory where the array is stored.
  - reference to an object holds the **address** of the actual object
- Example [arrays as objects]

```
int[] A={12, 24, 37, 53, 67};
int[] B=A;
B[3]=5;
```

- Example [cloning an array]

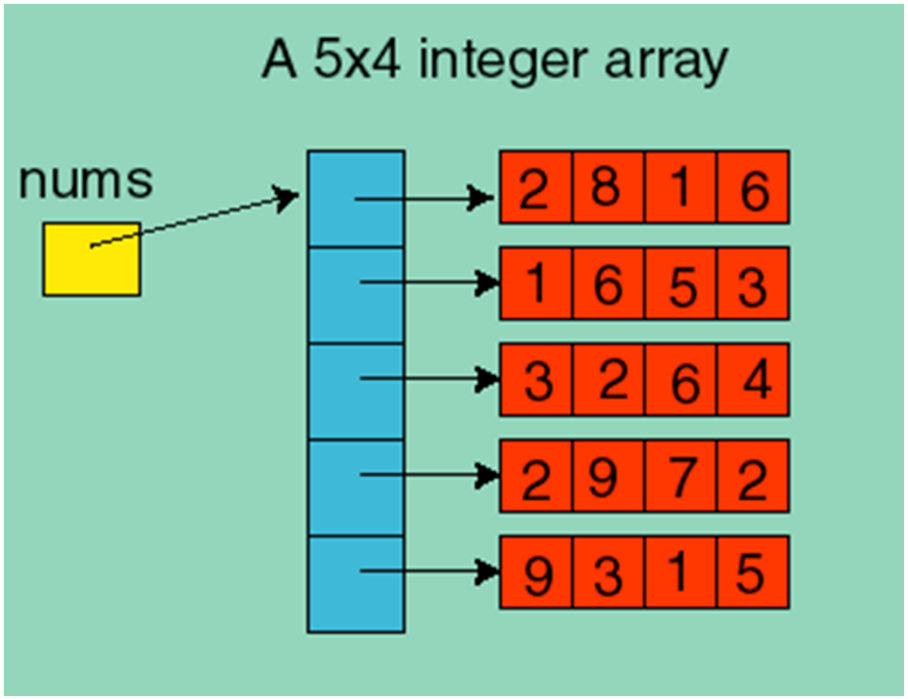
```
int[] A={12, 24, 37, 53, 67};
int[] B=A.clone();
B[3]=5;
```



Example

Example [2D array in Java = array of arrays]

```
int[][] nums = new int[5][4];

int[][] nums;

nums = new int[5][];

for (int i=0; i<5; i++) {

    nums[i] = new int[4];

}
```



Array Lists

The Array List ADT (§6.1)

- The Array List ADT extends the notion of array by storing a sequence of arbitrary objects
- An element can be accessed, inserted or removed by specifying its rank (number of elements preceding it)
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)

The Array List ADT

```
public interface IndexList<E> {
/** Returns the number of elements in this list */
public int size();
/** Returns whether the list is empty. */
public boolean isEmpty();
/** Inserts an element e to be at index I, shifting all elements after this. */
public void add(int I, E e) throws IndexOutOfBoundsException;
/** Returns the element at index I, without removing it. */
public E get(int i) throws IndexOutOfBoundsException;
/** Removes and returns the element at index I, shifting the elements after this. */
public E remove(int i) throws IndexOutOfBoundsException;
/** Replaces the element at index I with e, returning the previous element at i. */
public E set(int I, E e) throws IndexOutOfBoundsException;
}
```

Performance

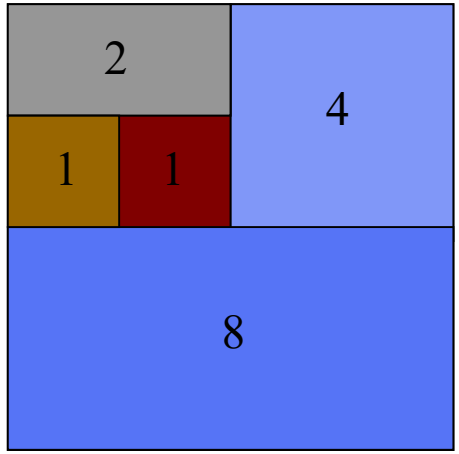
- In the array based implementation
  - The space used by the data structure is $O(n)$
  - ***size***, ***isEmpty***, ***get*** and ***set*** run in $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time
- In an ***add*** operation, when the array is full, instead of throwing an exception, we could replace the array with a larger one.
- In fact **java.util.ArrayList** implements this ADT using **extendable arrays** that do just this.

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n **add(o)** operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 = 2n - 1$$

geometric series

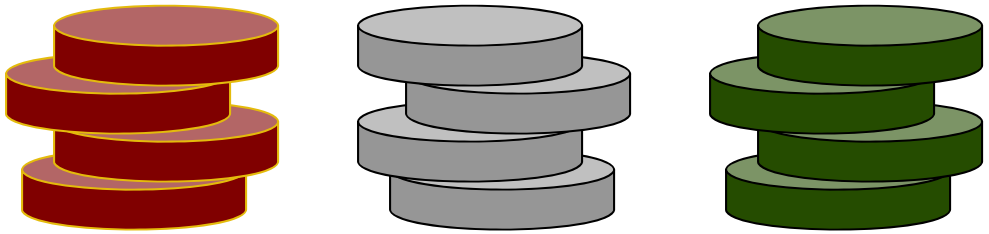


- Thus $T(n)$ is $O(n)$
- **The amortized time of an add operation is $O(1)$!**

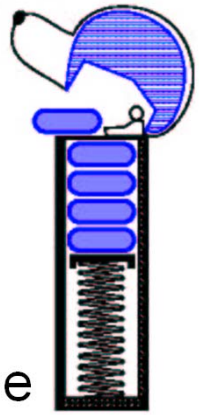
Recall: $\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$

Stacks

Chapter 5.1



The Stack ADT



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

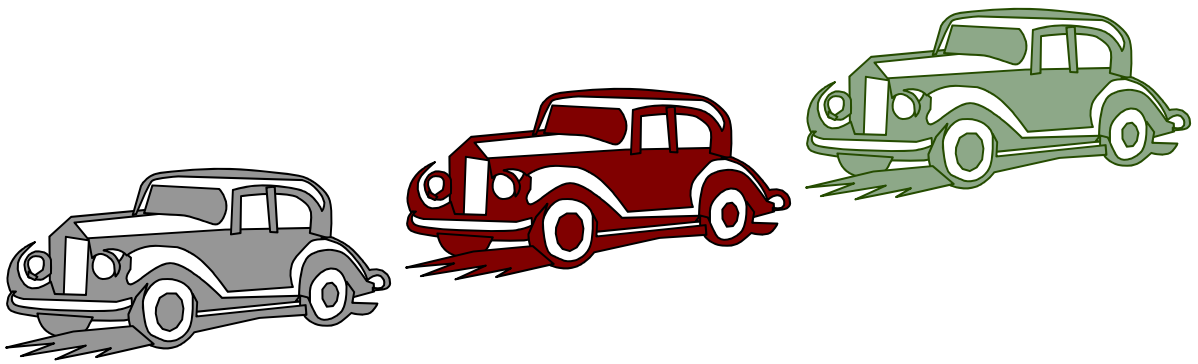
Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
    return t + 1

Algorithm pop()
    if isEmpty() then
        throw EmptyStackException
    else
        t ← t - 1
        return S[t + 1]
```

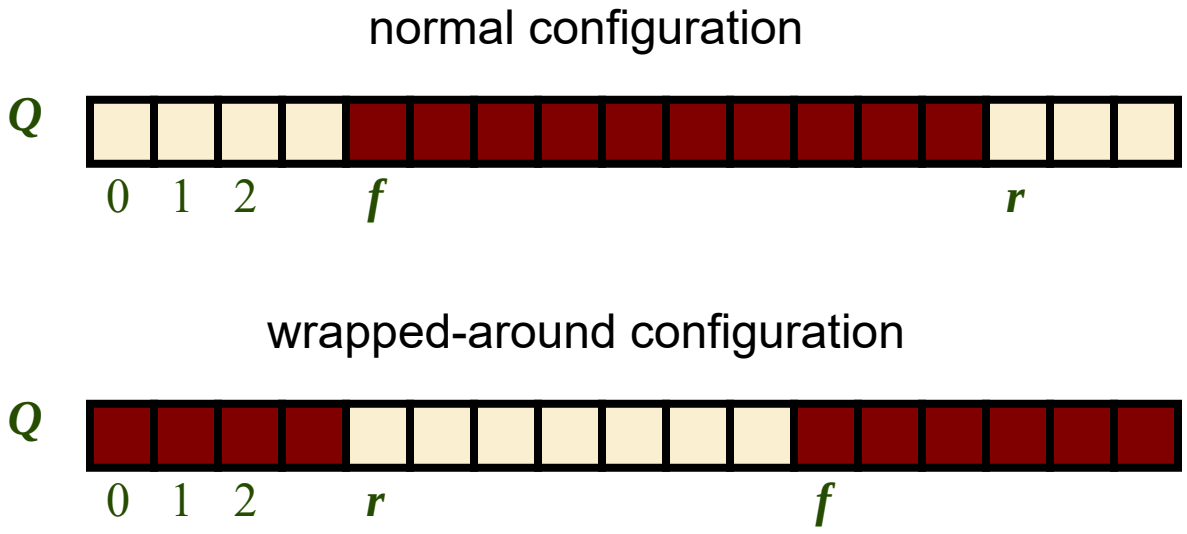
Queues

Chapters 5.2-5.3



Array-Based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty



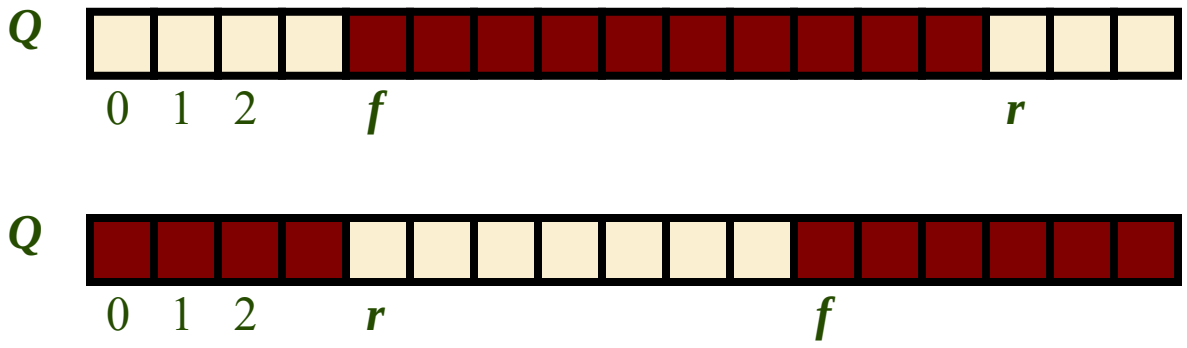
Queue Operations

- We use the modulo operator (remainder of division)

Algorithm ***size***()
  return $(N - f + r) \bmod N$

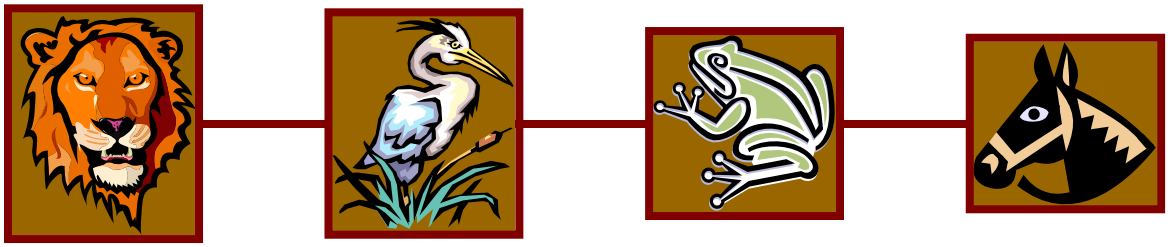
Algorithm ***isEmpty***()
  return $(f = r)$

Note: $N - f + r = (r + N) - f$



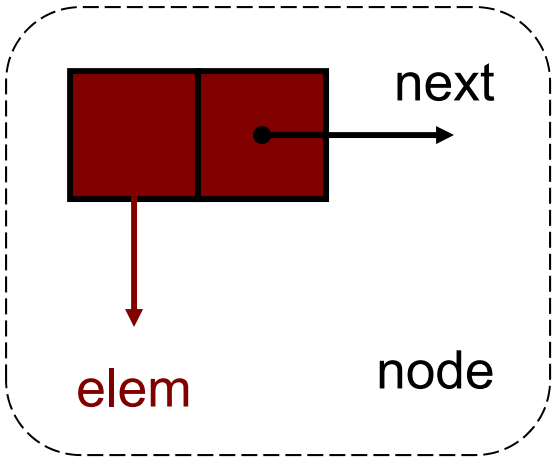
Linked Lists

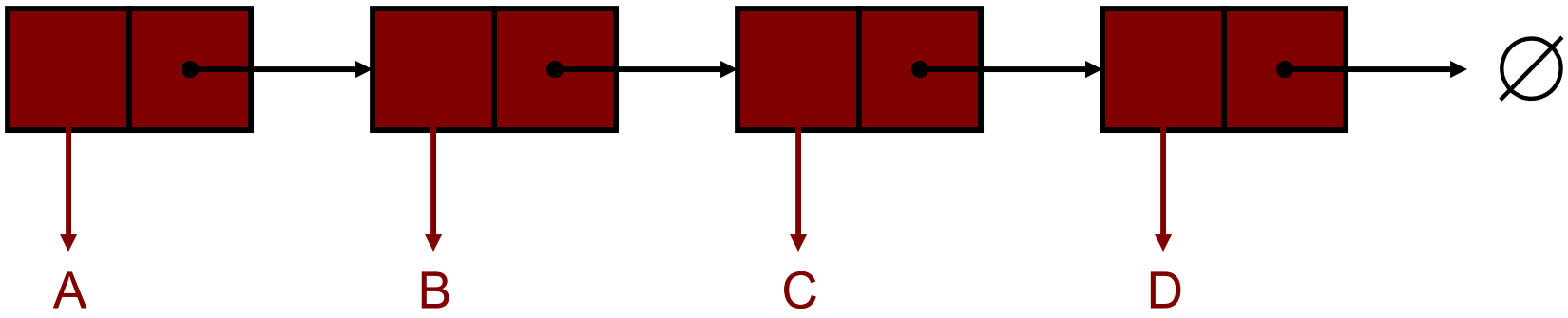
Chapters 3.2 – 3.3



Singly Linked List (§ 3.2)

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



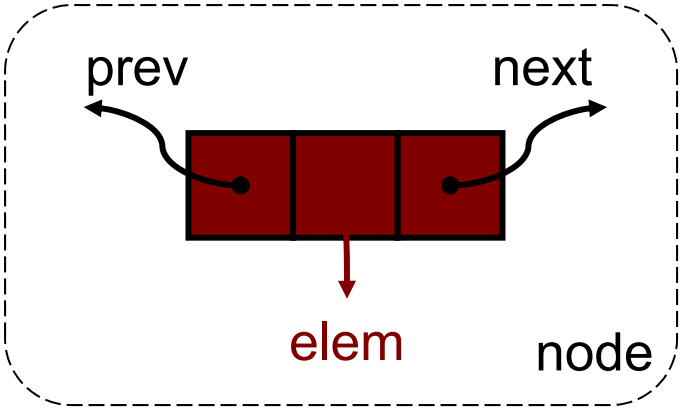


Running Time

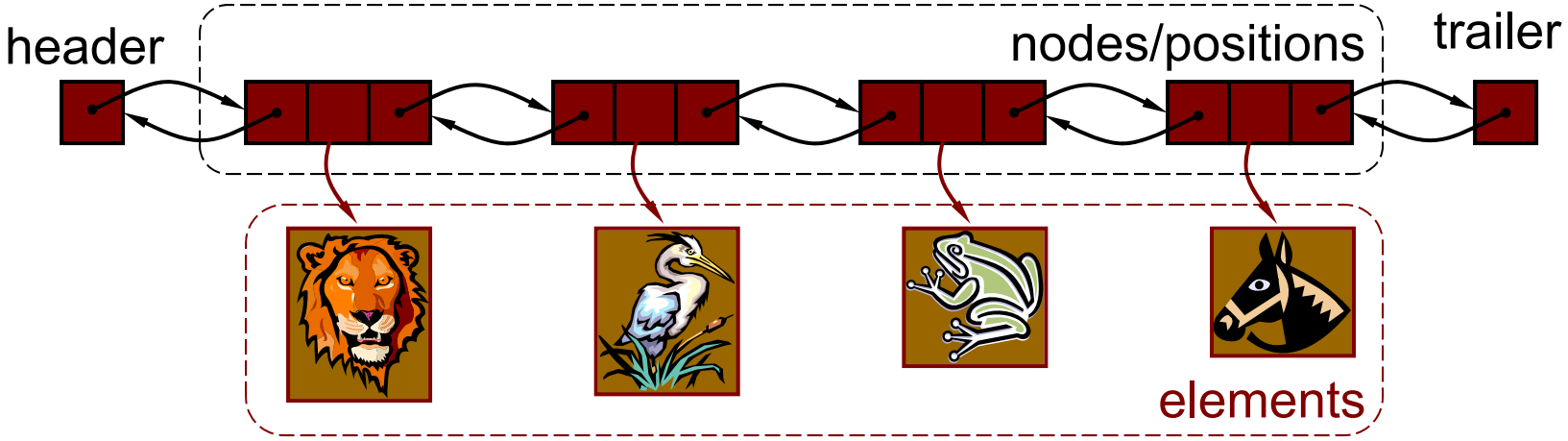
- Adding at the head is O(1)
- Removing at the head is O(1)
- **How about tail operations?**

Doubly Linked List

- Doubly-linked lists allow more flexible list management (constant time operations at both ends).
- Nodes store:
  - element
  - link to the previous node
  - link to the next node



- Special trailer and header (sentinel) nodes



Topics on the Midterm

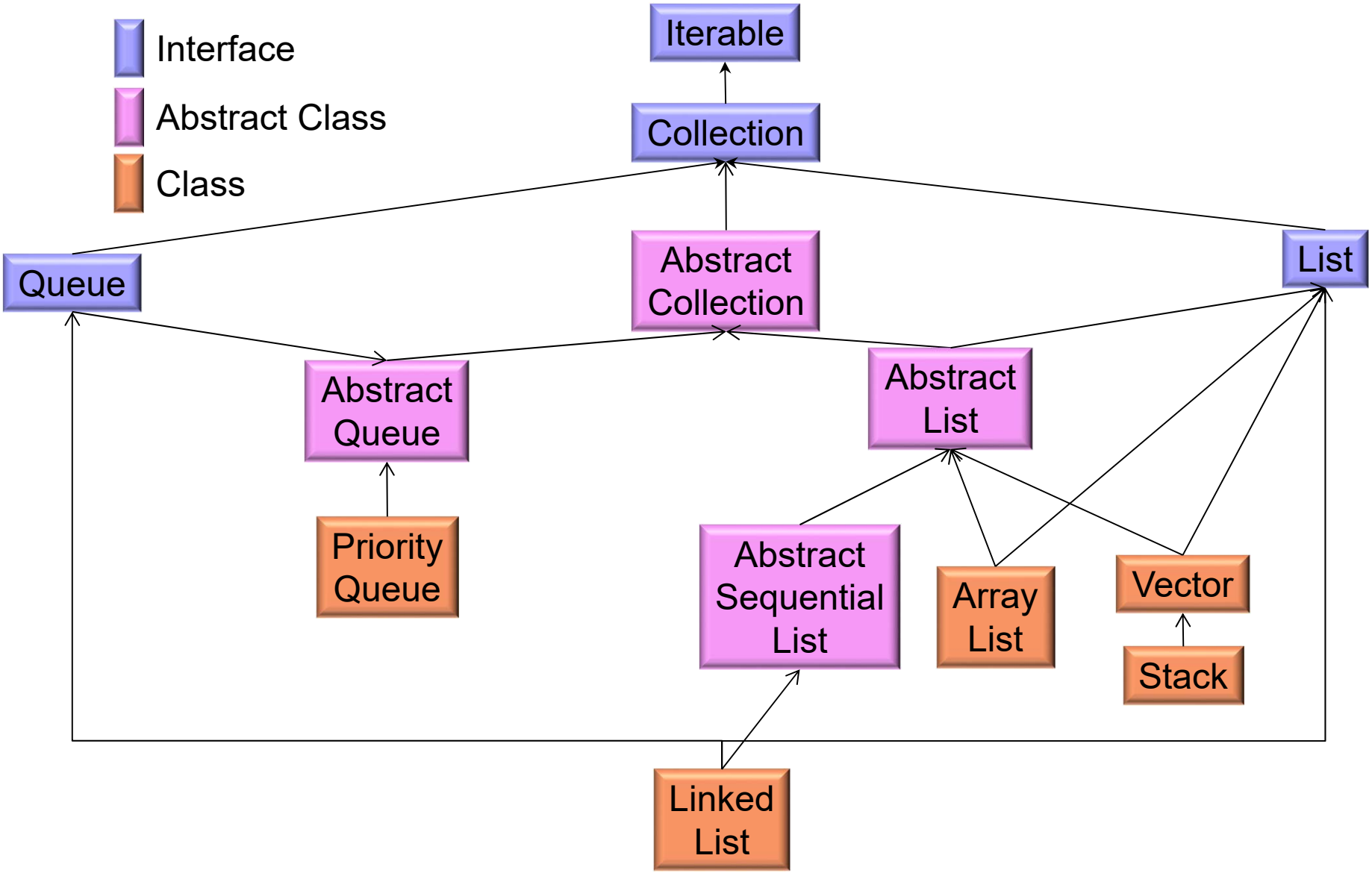
- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- **The Java Collections Framework**
- Recursion
- Trees
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants

Iterators

- An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.
- You get an Iterator for a collection by calling its iterator method.
- Suppose **collection** is an instance of a **Collection**. Then to print out each element on a separate line:

```
Iterator<E> it = collection.iterator();

while (it.hasNext())

    System.out.println(it.next());
```

The Java Collections Framework (Ordered Data Types)



Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- **Recursion**
- Trees
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants

Linear Recursion Design Pattern

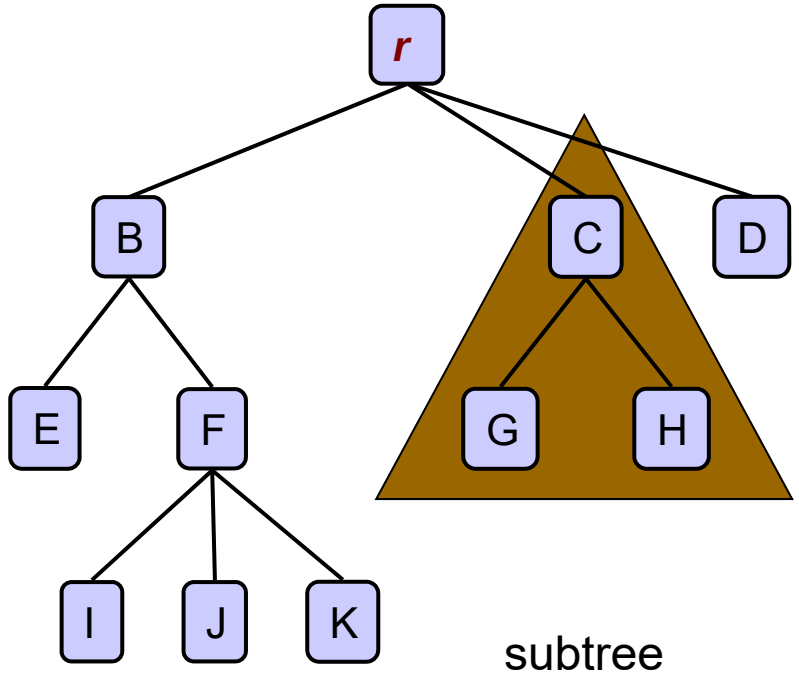
- **Test for base cases**
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls **must** eventually reach a base case, and the handling of each base case should not use recursion.
- ***Recurse once***
  - Perform a single recursive call. (This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.)
  - Define each possible recursive call so that it makes **progress** towards a base case.

Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.
- Example 1: **The Fibonacci Sequence**

Formal Definition of Rooted Tree

- A rooted tree may be empty.
- Otherwise, it consists of
  - A root node ***r***
  - A set of **subtrees** whose roots are the children of ***r***

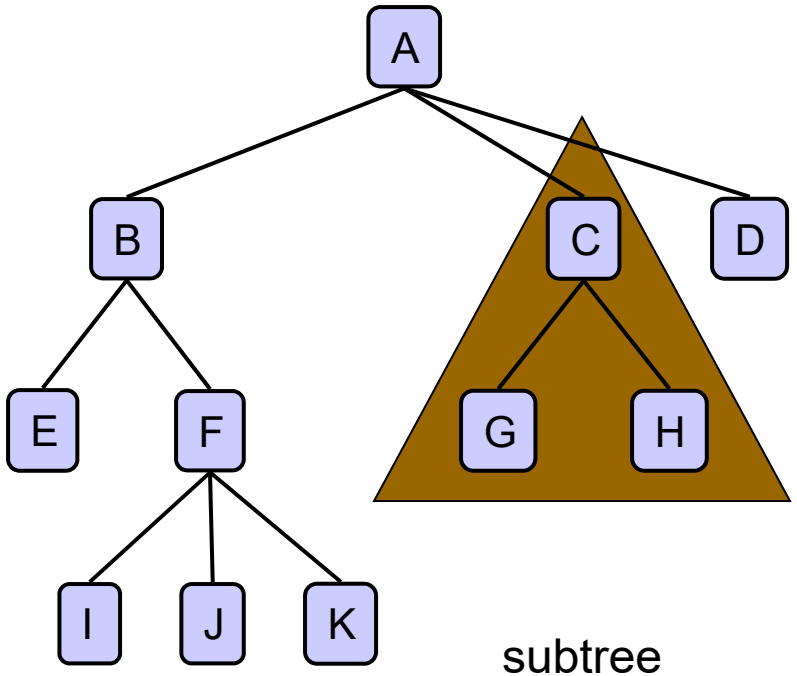


Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- **Trees**
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants

Tree Terminology

- **Root:** node without parent (A)
- **Internal node:** node with at least one child (A, B, C, F)
- **External node (a.k.a. leaf)**: node without children (E, I, J, K, G, H, D)
- **Ancestors of a node:** parent, grandparent, grand-grandparent, etc.
- **Descendant of a node:** child, grandchild, grand-grandchild, etc.
- **Siblings**: two nodes having the same parent
- **Depth of a node:** number of ancestors (excluding self)
- **Height of a tree:** maximum depth of any node (3)
- **Subtree:** tree consisting of a node and its descendants



Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
  - a node of a tree
- Just one method:
  - object element(): returns the element stored at the position

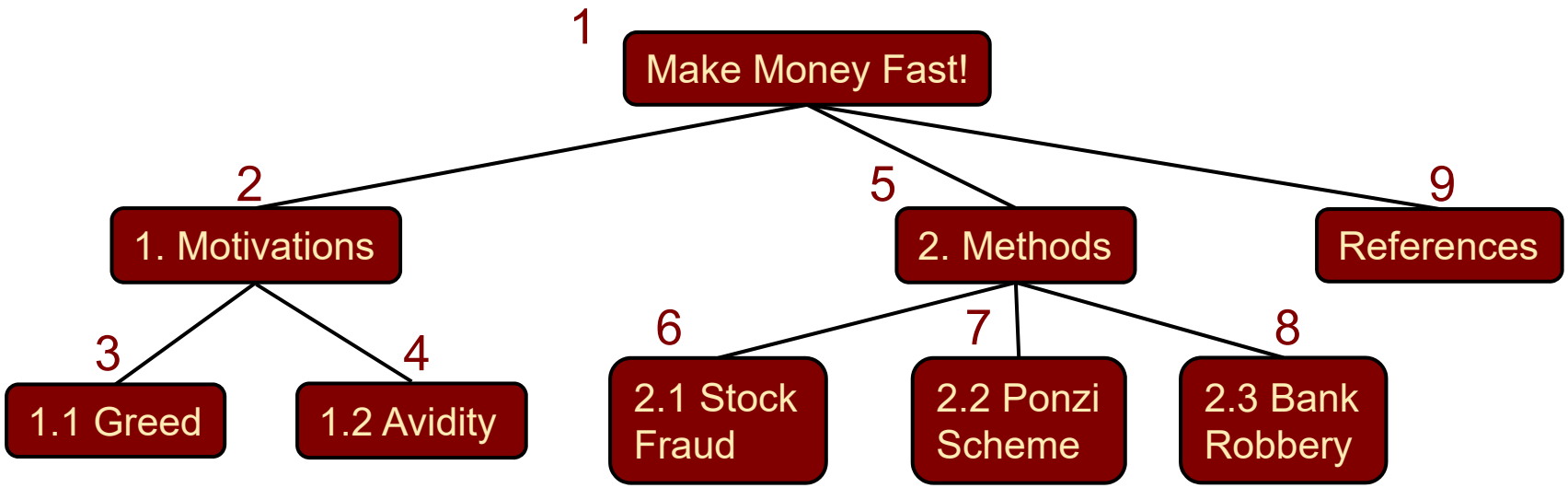
Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace(p, o)
  - Additional update methods may be defined by data structures implementing the Tree ADT

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants

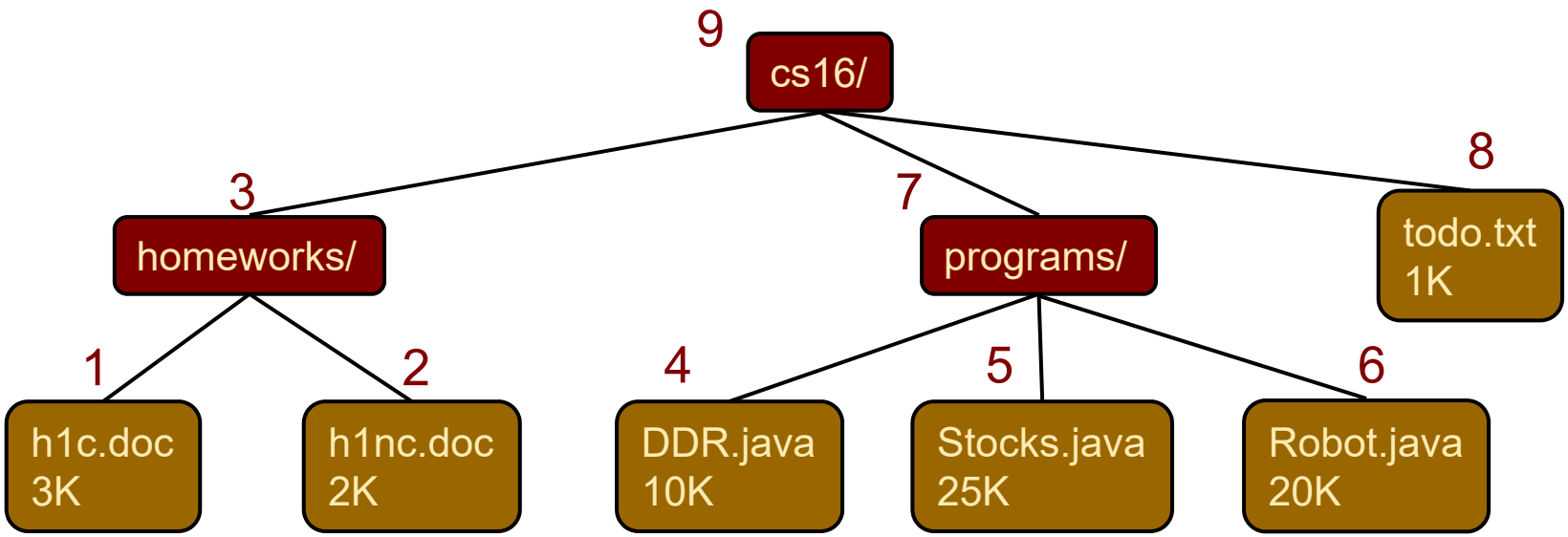
Algorithm ***preOrder***(***v***)
    visit(***v***)
    for each child ***w*** of ***v***
        preOrder (***w***)



Postorder Traversal

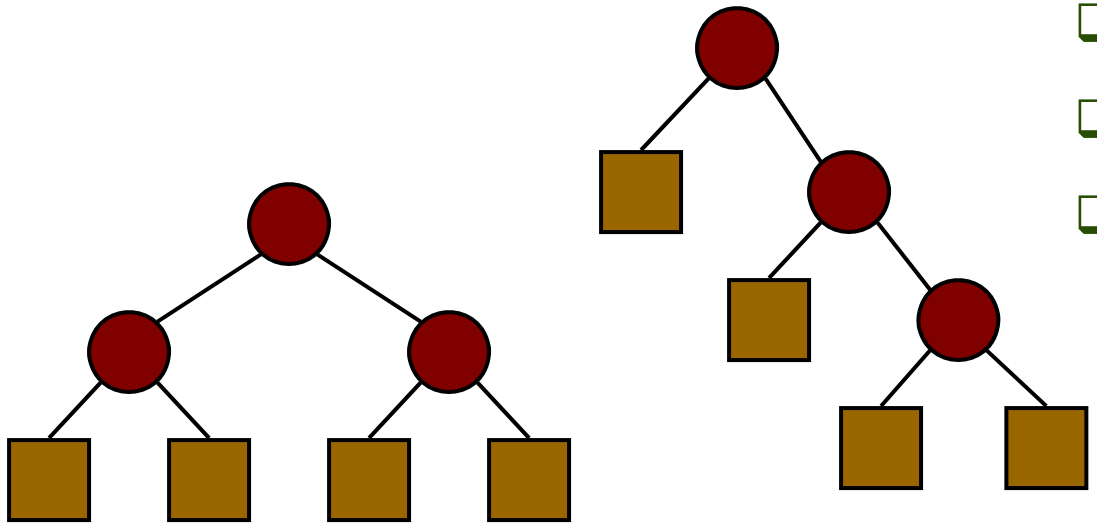
- In a postorder traversal, a node is visited after its descendants

```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```



Properties of Proper Binary Trees

- Notation
  - ***n*** number of nodes
  - ***e*** number of external nodes
  - ***i*** number of internal nodes
  - ***h*** height



- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2(n + 1) - 1$

BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position **left**(p)
  - position **right**(p)
  - boolean **hasLeft**(p)
  - boolean **hasRight**(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- **Priority Queues & Heaps**
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants

Priority Queue ADT

- A priority queue stores a collection of **entries**
- Each **entry** is a pair (key, value)
- Main methods of the Priority Queue ADT
  - **insert**(k, x) inserts an entry with key k and value x
  - **removeMin**() removes and returns the entry with smallest key
- Additional methods
  - **min**() returns, but does not remove, an entry with smallest key
  - **size**(), **isEmpty**()
- Applications:
  - Process scheduling
  - Standby flyers

Entry ADT

- An **entry** in a priority queue is simply a key-value pair
- Methods:
  - **key**(): returns the key for this entry
  - **value**(): returns the value for this entry

- As a Java interface:

```
/**
 * Interface for a key-value
 * pair entry
 **/
public interface Entry {
    public Object key();
    public Object value();
}
```

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator
- The primary method of the Comparator ADT:
  - **compare**(a, b):
    - Returns an integer *i* such that
      - $i < 0$ if $a < b$
      - $i = 0$ if $a = b$
      - $i > 0$ if $a > b$
      - an error occurs if *a* and *b* cannot be compared.

Sequence-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - **insert** takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - **removeMin** and **min** take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - **insert** takes $O(n)$ time since we have to find the right place to insert the item
  - **removeMin** and **min** take $O(1)$ time, since the smallest key is at the beginning

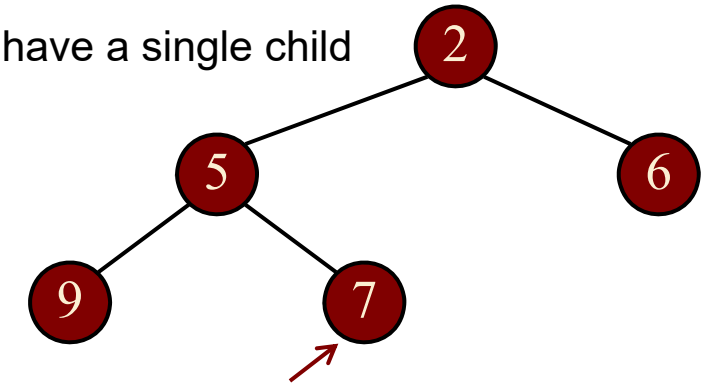
Is this tradeoff inevitable?

Heaps

- Goal:
  - O(log n) insertion
  - O(log n) removal
- Remember that O(log n) is almost as good as O(1)!
  - e.g., n = 1,000,000,000 → log n ≅ 30
- There are min heaps and max heaps. We will assume min heaps.

Min Heaps

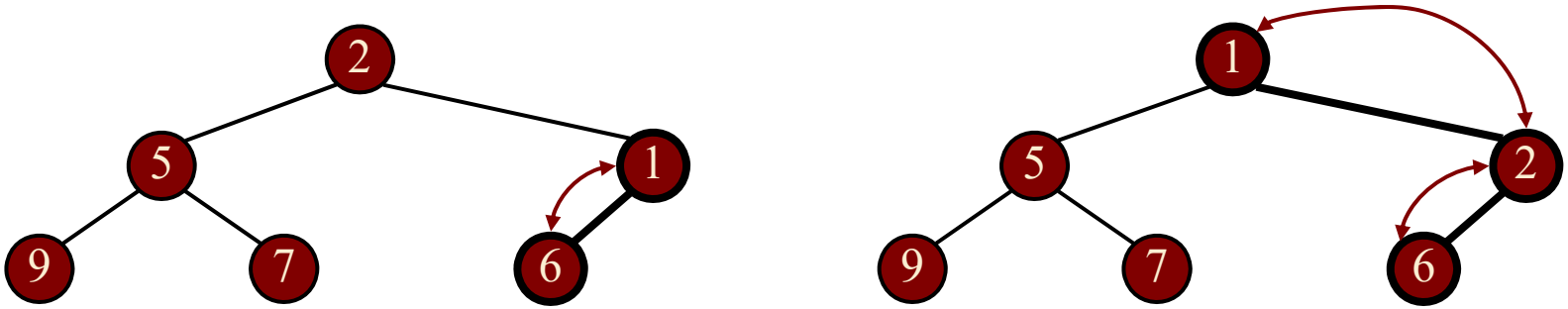
- A min heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Heap-order: for every internal node v other than the root
    - $key(v) \geq key(parent(v))$
  - (Almost) complete binary tree: let h be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
    - at depth h □ 1
      - the internal nodes are to the left of the external nodes
      - Only the rightmost internal node may have a single child



- **The last node of a heap is the rightmost node of depth *h***

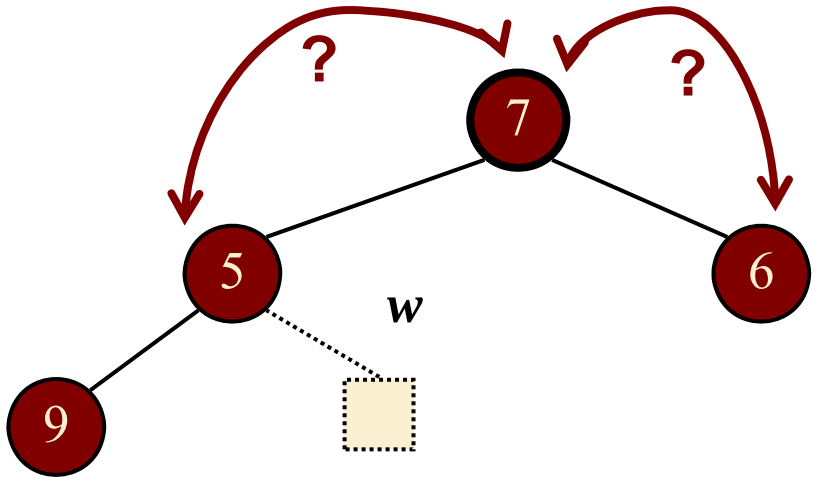
Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm **upheap** restores the heap-order property by swapping k along an upward path from the insertion node
- **Upheap** terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, **upheap** runs in $O(\log n)$ time



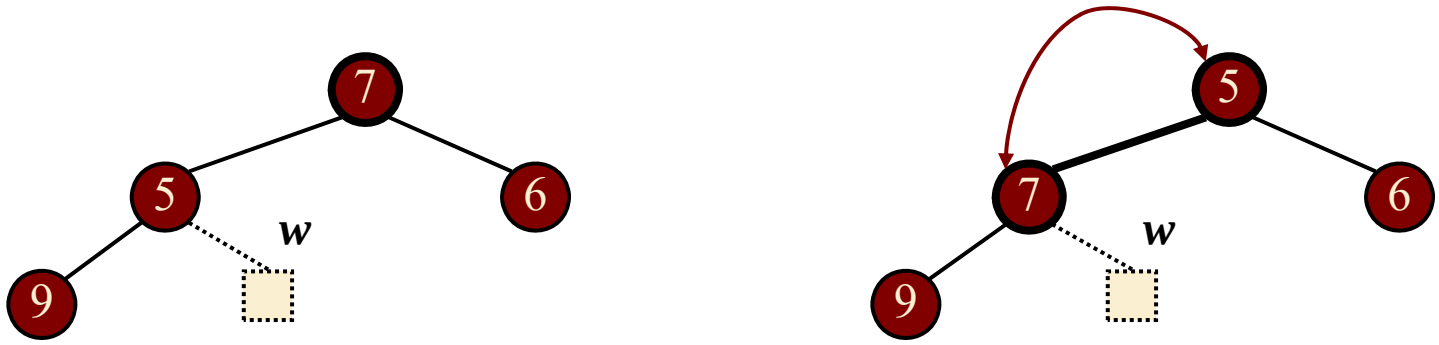
Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Note that there are, in general, many possible downward paths – which one do we choose?



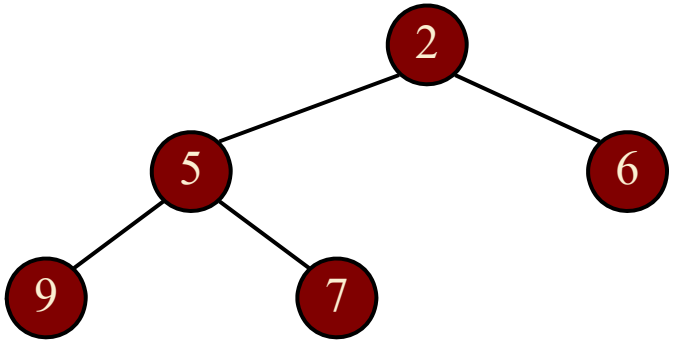
Downheap

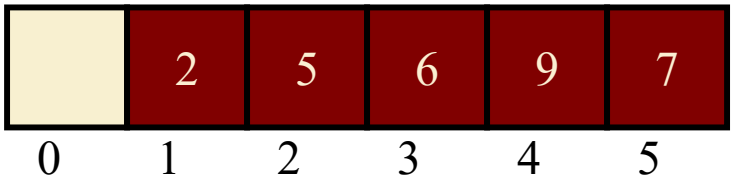
- We select the downward path through the **minimum-key** nodes.
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



Array-based Heap Implementation

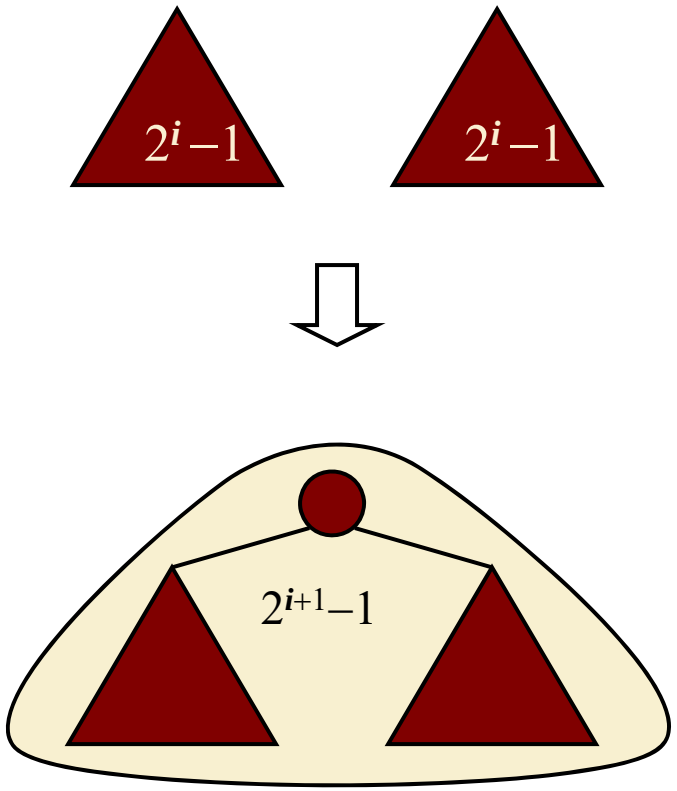
- We can represent a heap with n keys by means of an array of length $n + 1$
- Links between nodes are not explicitly stored
- The cell at rank 0 is not used
- The root is stored at rank 1.
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
  - the parent is at rank **floor**(i/2)
  - if 2i + 1 > n, the node has no right child
  - if 2i > n, the node is a leaf



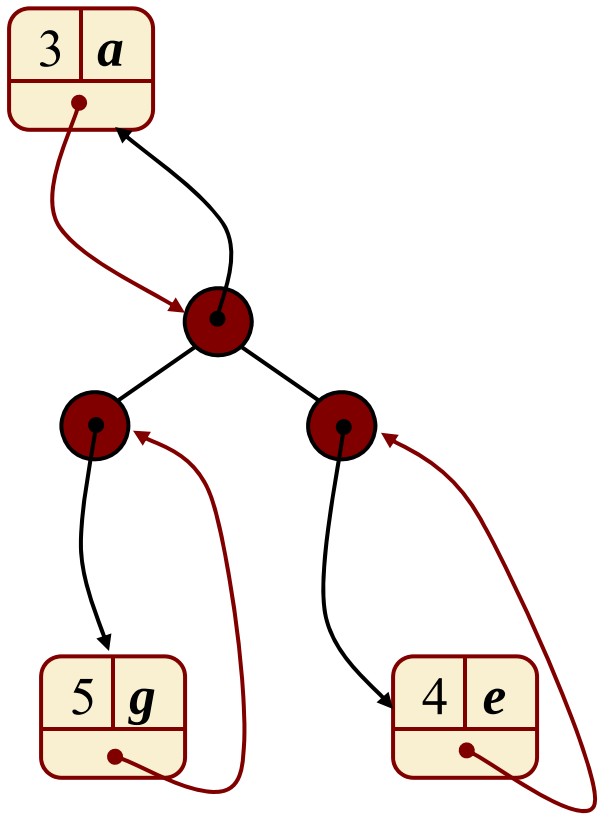


Bottom-up Heap Construction

- We can construct a heap storing n keys using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys
- Run time for construction is $O(n)$.



Adaptable Priority Queues



Additional Methods of the Adaptable Priority Queue ADT

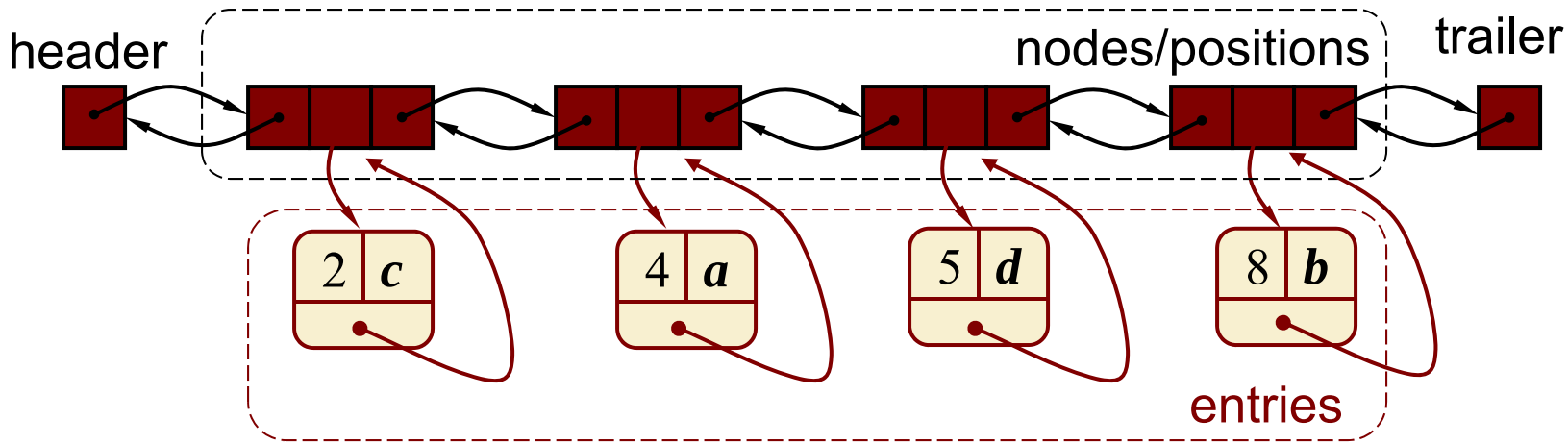
- remove(*e*): Remove from *P* and return entry *e*.
- replaceKey(*e*,*k*): Replace with *k* and return the old key; an error condition occurs if *k* is invalid (that is, *k* cannot be compared with other keys).
- replaceValue(*e*,*x*): Replace with *x* and return the old value.

Location-Aware Entries

- A locator-aware entry identifies and tracks the location of its (key, value) object within a data structure

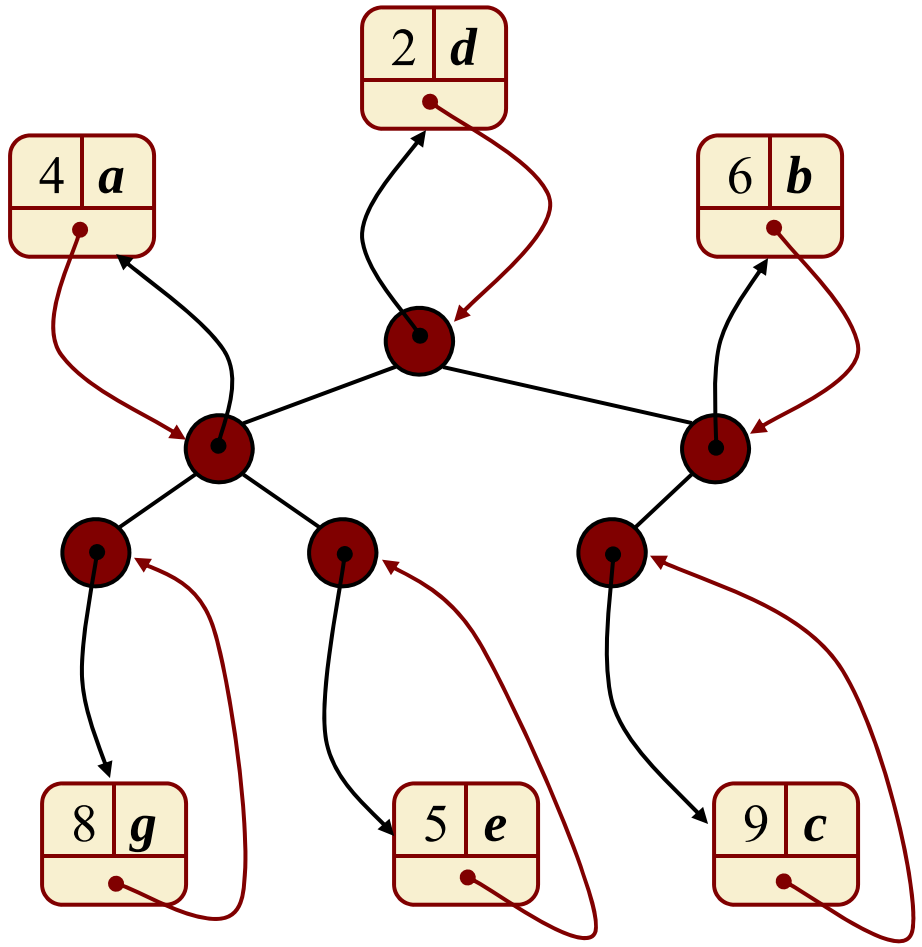
List Implementation

- A location-aware list entry is an object storing
  - key
  - value
  - position (or rank) of the item in the list
- In turn, the position (or array cell) stores the entry
- Back pointers (or ranks) are updated during swaps



Heap Implementation

- A location-aware heap entry is an object storing
  - key
  - value
  - position of the entry in the underlying heap
- In turn, each heap position stores an entry
- Back pointers are updated during entry swaps



Performance

- Times better than those achievable without location-aware entries are highlighted in red:

Method	**Unsorted List**	**Sorted List**	**Heap**
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(\log n)}$
replaceKey	$\boldsymbol{O(1)}$	$O(n)$	$\boldsymbol{O(\log n)}$
replaceValue	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$

Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
- **Maps, Hash Tables & Dictionaries**
- Iterative Algorithms & Loop Invariants

Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
- Applications:
  - address book
  - student-record database

Performance of a List-Based Map

- Performance:
  - put, get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for small maps

Hash Tables

- A hash table is a data structure that can be used to make map operations faster.
- While worst-case is still O(n), average case is typically O(1).

Polynomial Hash Codes

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0 a_1 \ldots a_{n-1}$$

  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$ at a fixed value z, ignoring overflows

  - Especially suitable for strings
  - Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
    - The following polynomials are successively computed, each from the previous one in $O(1)$ time

      $$p_0(z) = a_{n-1}$$

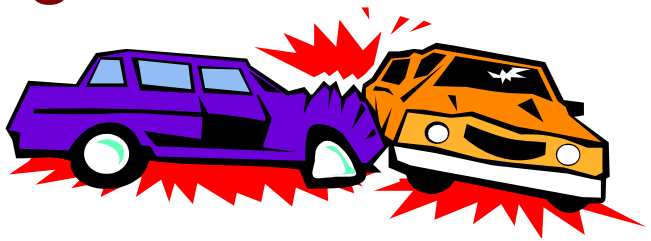
      $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$

  - We have $p(z) = p_{n-1}(z)$

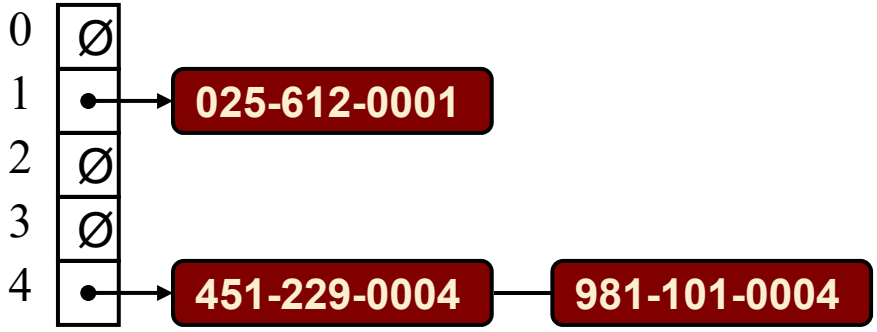
Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime (on the assumption that the differences between hash keys *y* are less likely to be multiples of primes).
- Multiply, Add and Divide (MAD):
  - $h_2(y) = [(ay + b) \bmod p] \bmod N$, where
    - p is a prime number greater than N
    - a and b are integers chosen at random from the interval [0, p – 1], with a > 0.

Collision Handling



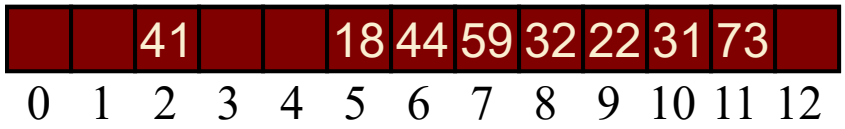
- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**:
  - Let each cell in the table point to a linked list of entries that map there
  - Separate chaining is simple, but requires additional memory outside the table



Open Addressing: Linear Probing

- **Open addressing**: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, so that future collisions cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Open Addressing: Double Hashing

- Double hashing is an alternative open addressing method that uses a **secondary hash function h'(k)** in addition to the primary hash function h(x).
- Suppose that the primary hashing i=h(k) leads to a collision.
- We then iteratively probe the locations
  $(i + jh'(k)) \bmod N$ for $j = 0, 1, \ldots, N - 1$
- The secondary hash function ***h'(k)*** cannot have zero values
- ***N*** is typically chosen to be prime.
- Common choice of secondary hash function h'(k):
  - $h'(k) = q - k \bmod q$, where
    - $q < N$
    - q is a prime
- The possible values for h'(k) are
  $1, 2, \ldots, \boldsymbol{q}$

Dictionary ADT

- The dictionary ADT models a searchable collection of key-element entries
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key **are** allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
- Dictionary ADT methods:
  - get(k): if the dictionary has at least one entry with key k, returns one of them, else, returns null
  - getAll(k): returns an iterable collection of all entries with key k
  - put(k, v): inserts and returns the entry (k, v)
  - remove(e): removes and returns the entry e. Throws an exception if the entry is not in the dictionary.
  - entrySet(): returns an iterable collection of the entries in the dictionary
  - size(), isEmpty()

A List-Based Dictionary

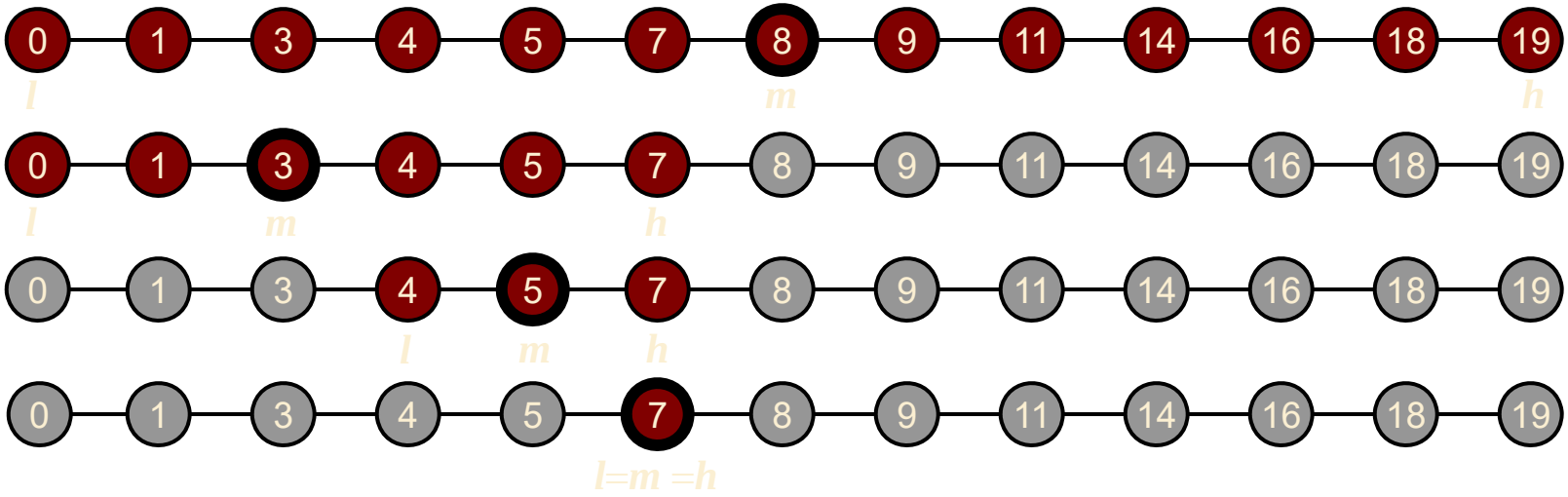
- A log file or audit trail is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- Performance:
  - insert takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - find and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Table Implementation

- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.

Ordered Maps and Dictionaries

- If keys obey a total order relation, can represent a map or dictionary as an ordered search table stored in an array.
- Can then support a fast find(k) using **binary search.**
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
  - Example: find(7)



Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- **Iterative Algorithms & Loop Invariants**

Loop Invariants

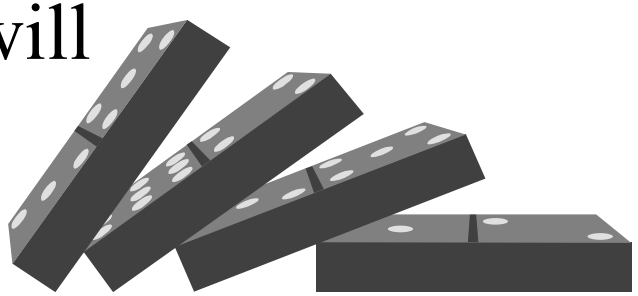
- Binary search can be implemented as an **iterative algorithm** (it could also be done recursively).
- **Loop Invariant:** An **assertion** about the current state useful for designing, analyzing and proving the correctness of iterative algorithms.

Establishing Loop Invariant

From the Pre-Conditions on the input instance we must establish the loop invariant.

Maintain Loop Invariant

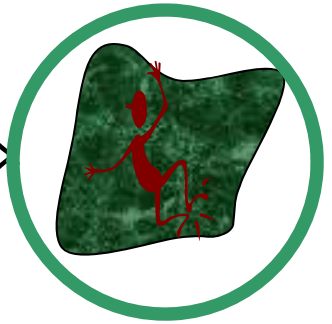
• By Induction the computation will always be in a safe location.



$\Rightarrow S(0)$

$\Rightarrow \forall i, S(i) \Rightarrow S(i+1)$

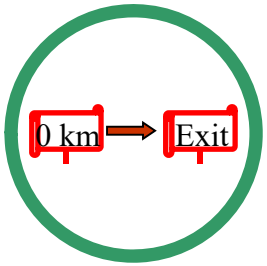
$\Rightarrow \forall i, S(i) \Rightarrow$



Ending The Algorithm

- Define Exit Condition

- Termination: With sufficient progress, the exit condition will be met.



- When we exit, we know
  - exit condition is true
  - loop invariant is true

  from these we must establish the post conditions.

Topics on the Midterm

- Data Structures & Object-Oriented Design
- Run-Time Analysis
- Linear Data Structures
- The Java Collections Framework
- Recursion
- Trees
- Priority Queues & Heaps
- Maps, Hash Tables & Dictionaries
- Iterative Algorithms & Loop Invariants