CSE214

FINAL'S REVIEW

ALGORITHMS: ANALYSIS, SOLUTION PATTERNS, AND SORTING

BIG-OH NOTATION

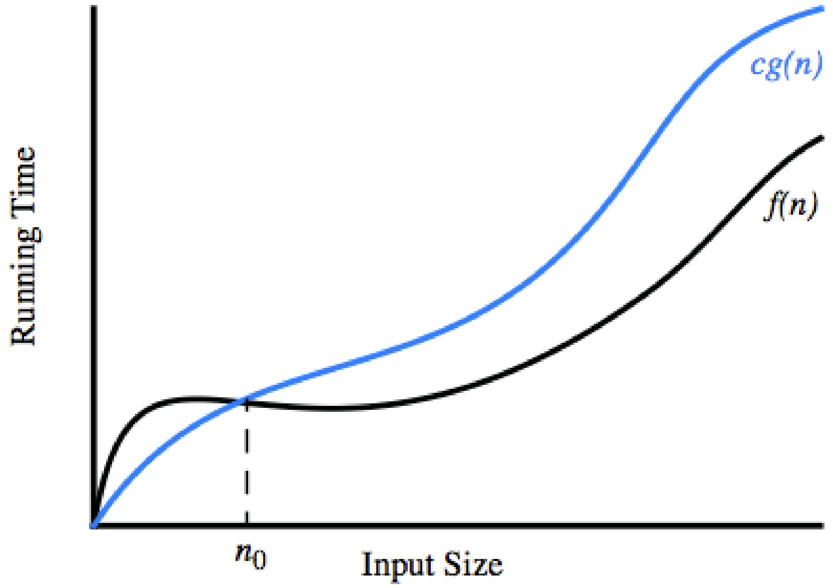
Focus on the growth rate of the running time as a function of the input size *n*, taking a "big-picture" approach.

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \quad \text{for} \quad n \geq n_0.$$

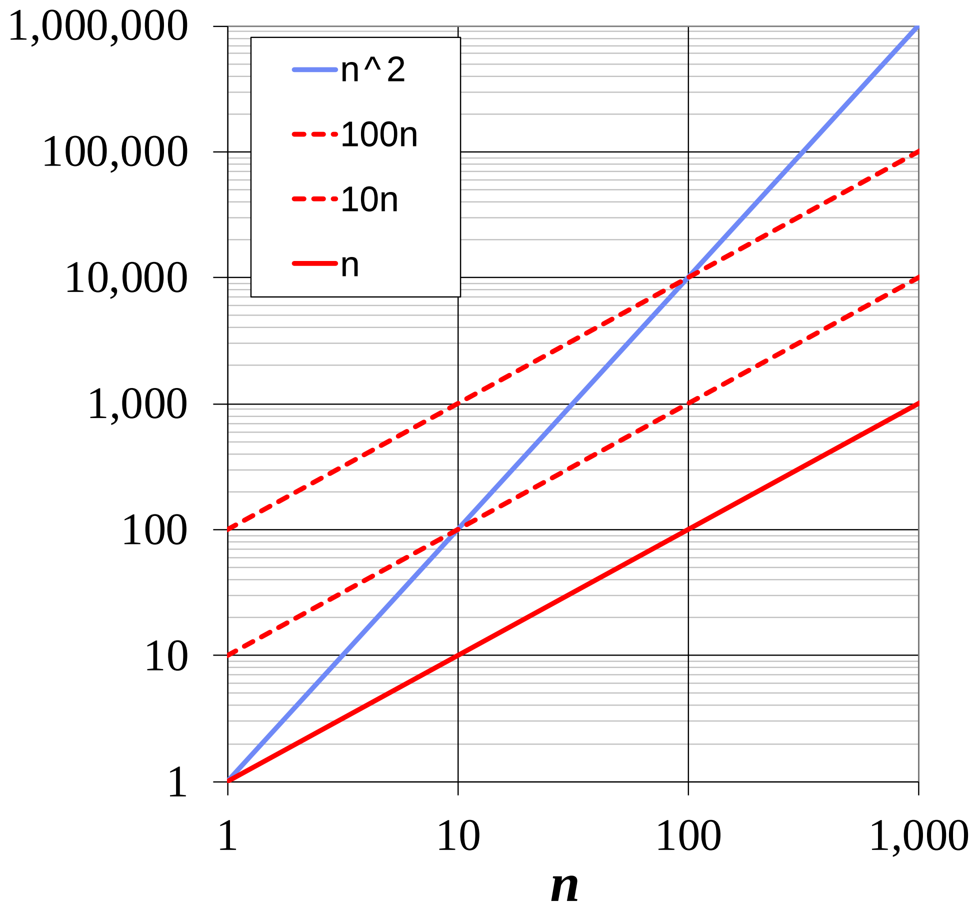
This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "$f(n)$ is ***big-Oh*** of $g(n)$."

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



BIG-OH EXAMPLE

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality c annot be satisfied since c must be a constant



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

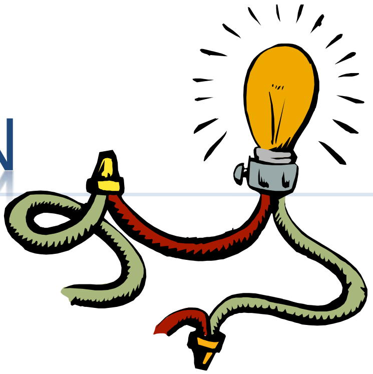
  $$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

big-Theta

- f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

  $$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

INTUITION FOR ASYMPTOTIC NOTATION



big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically greater than or equal to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)

ADVANCE TOPIC: COMPARISON OF THE STRATEGIES

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations (amortization)
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e., $T(n)/n$

THE RECURSION PATTERN EXAMPLE

- **Recursion**: when a method calls itself
- Classic example – the factorial function:
  n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition: $$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$
- As a Java method:

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );     // argument must be nonnegative
  else if (n == 0)
    return 1;                                  // base case
  else
    return n * factorial(n−1);                 // recursive case
}
```

CONTENT OF A RECURSIVE METHOD

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

BINARY SEARCH

Search for an integer in an ordered list

```
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                          // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                         // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);     // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);    // recur right of the middle
  }
}
```

TYPES OF RECURSION

- ***Linear recursion :*** If a recursive call starts at most one other.
- ***Binary recursion***: If a recursive call may start two others.
- ***Multiple recursion:*** If a recursive call may start three or more others.

Terminology reflects the structure of the recursion trace, not the asymptotic analysis of the running time.

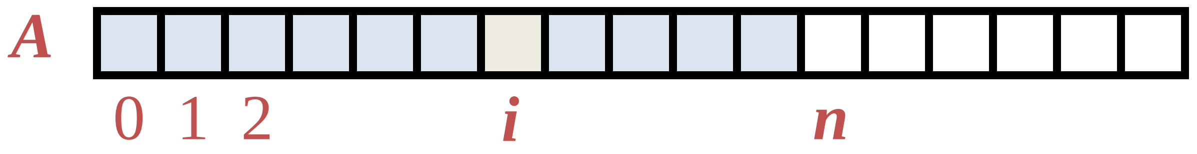
Recursion is important: Make sure you can trace the recursion code and figure it's time complexity.
Know everything about the recursion we have talked about!.

BASE DATASTRUCTURES

Base Data-structure questions will be integrated with the questions in the advance datastructures (ex> how dequeue() works in a queue when implemented with singly-linked list.)

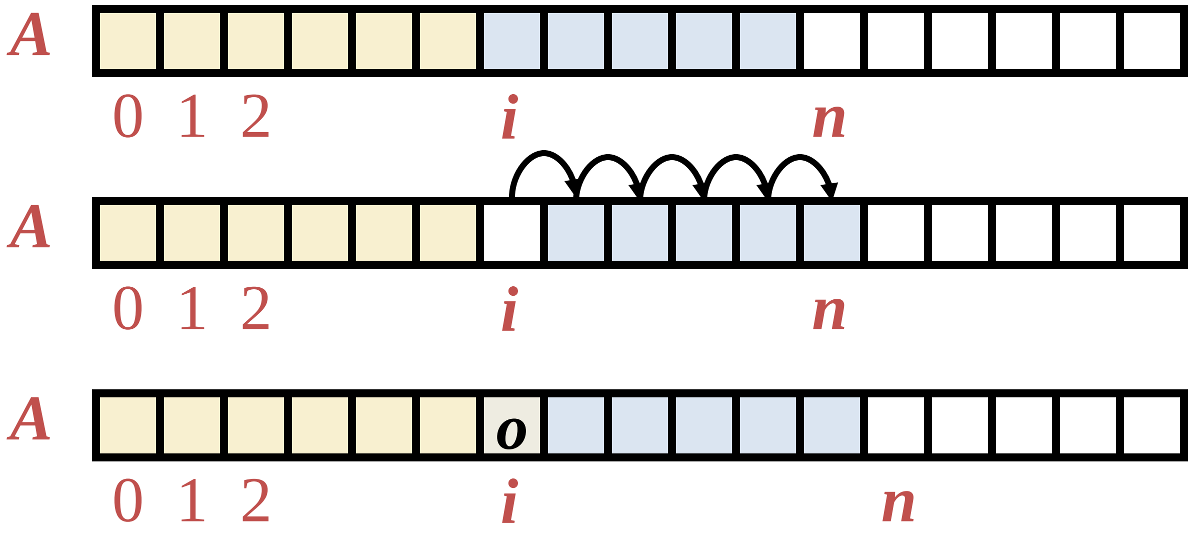
ARRAY LISTS

- An obvious choice for implementing the **list ADT** is to use an array, **A**, where **A[i]** stores (a reference to) the element with index **i**.
- With a representation based on an array **A**, the get(**i**) and set(**i**, **e**) methods are easy to implement by accessing **A[i]** (assuming **i** is a legitimate index).



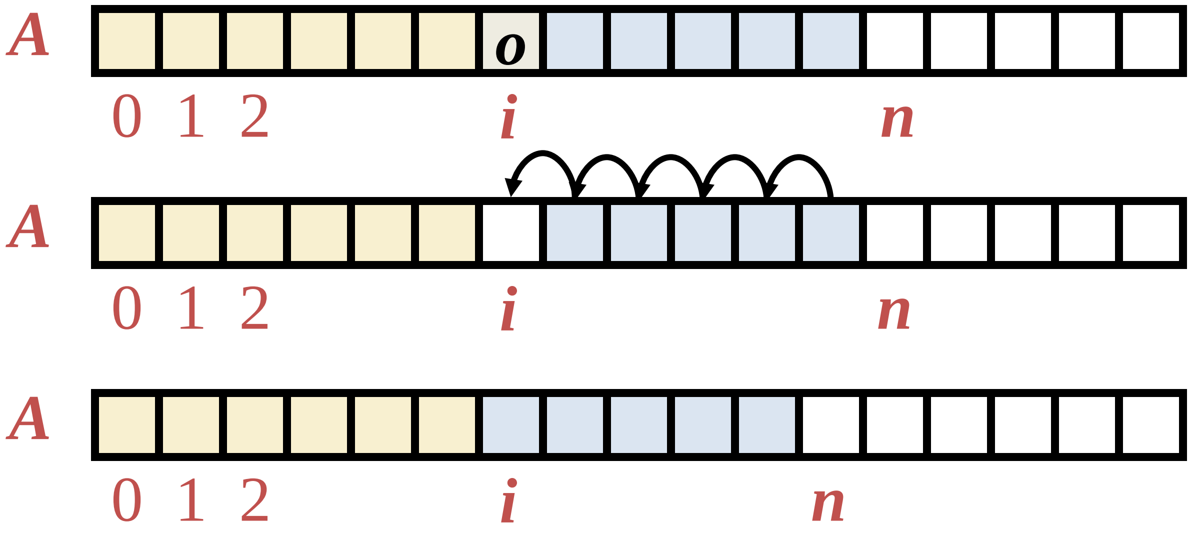
INSERTION

- In an operation $\boldsymbol{add}(\boldsymbol{i}, \boldsymbol{o})$, we need to make room for the new element by shifting forward the $\boldsymbol{n} - \boldsymbol{i}$ elements $\boldsymbol{A}[\boldsymbol{i}], \ldots, \boldsymbol{A}[\boldsymbol{n} - 1]$
- In the worst case ($\boldsymbol{i} = 0$), this takes $\boldsymbol{O}(\boldsymbol{n})$ time



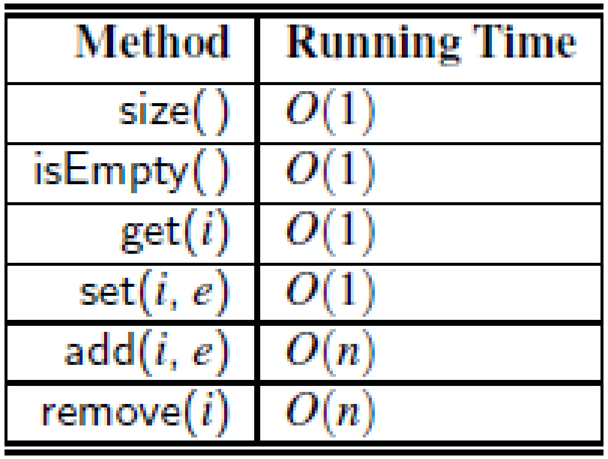
ELEMENT REMOVAL

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



PERFORMANCE OF ***ARRAY LIST***

- In an array-based implementation of a list (*array list*):
  - The space used by the data structure is $O(n)$
  - Indexing the element at i takes $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one …



Method	Running Time
size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

DYNAMIC ARRAY:

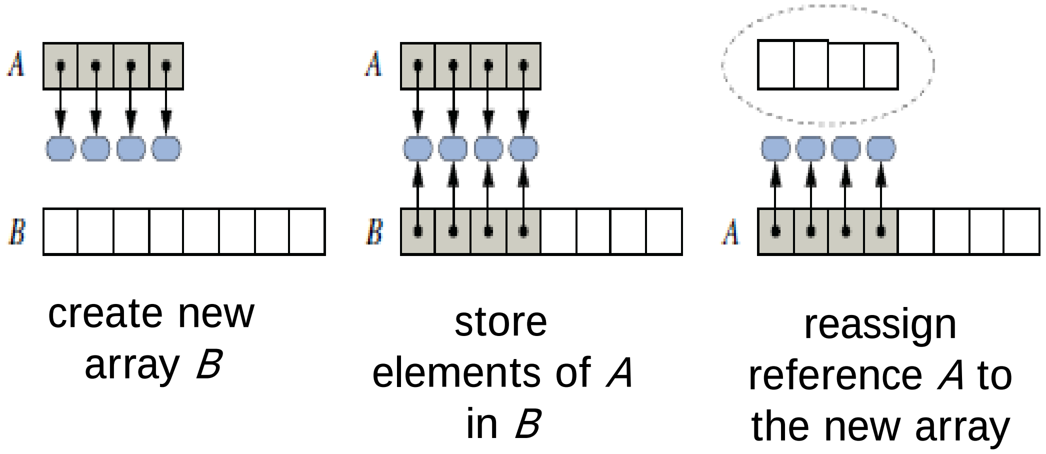
- Let push(o) be the operation that adds element o at the end of the list
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

```
Algorithm push(o)
  if $t = S.length - 1$ then
    $A \leftarrow$ new array of
          size …
    for $i \leftarrow 0$ to $n-1$ do
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$
```

IMPLEMENTING A DYNAMIC ARRAY

- Provide means to "grow" the array *A*

1. Allocate a new array *B* with larger capacity.
2. Set *B*[*k*]=*A*[*k*], for *k*=0, . . . ,*n*−1, where *n* denotes current number of items.
3. Set *A* = *B*, that is, we henceforth use the new array to support the list.
4. Insert the new element in the new array.


A
B
create new array *B*
A
B
store elements of *A* in *B*
A
reassign reference *A* to the new array

INCREMENTAL STRATEGY ANALYSIS

- Over n push operations, we replace the array $k = n/c$ times, where c is a constant
- The total time $T(n)$ of a series of n push operations is proportional to

Actual push op.

$$\boxed{n} + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of a push operation is $O(n)$

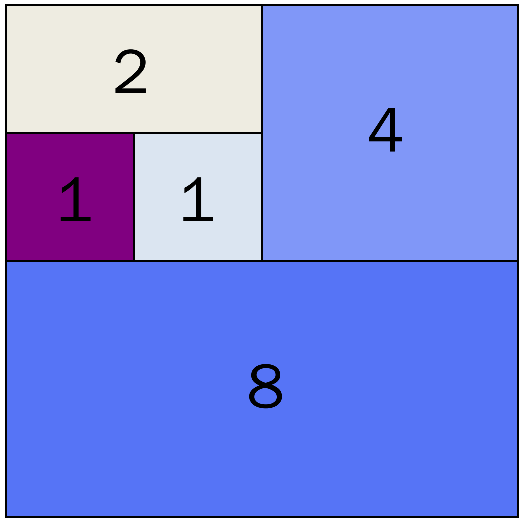
DOUBLING STRATEGY ANALYSIS

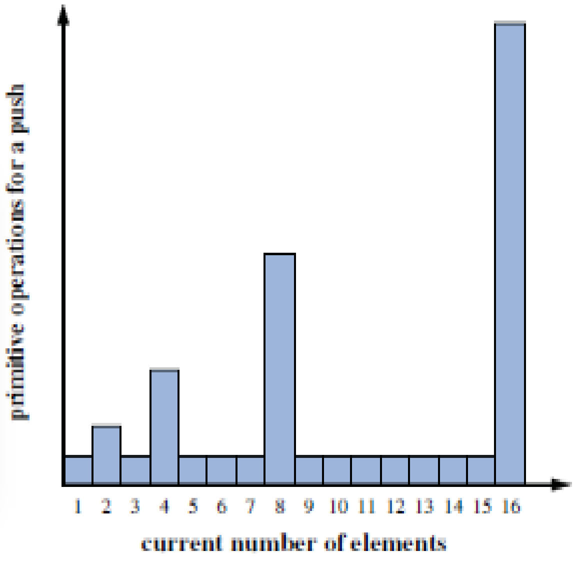
- We replace the array $k = \log_2 n$ times (2^{k+1} -1=**n**; solve for k)
- The total time $T(n)$ of a series of n push operations is proportional to

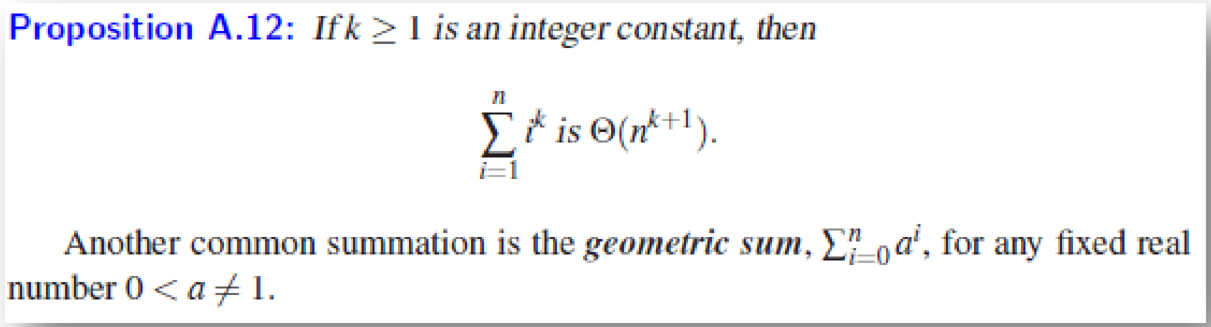
  $$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
  $$n + 2^{k+1} - 1 =$$
  $$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$

geometric series







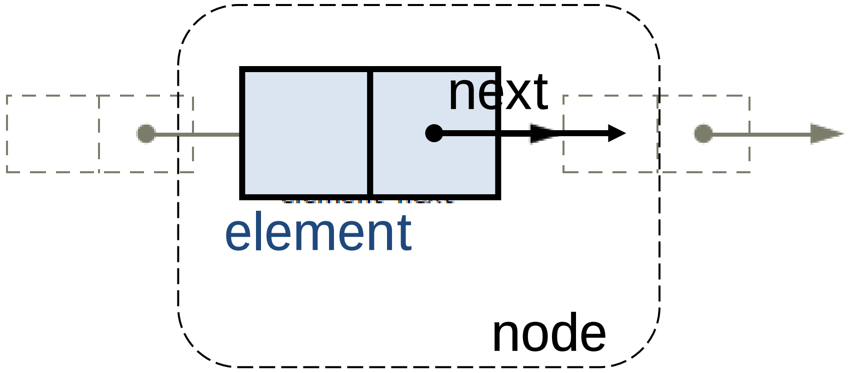
Proposition A.12: *If $k \geq 1$ is an integer constant, then*

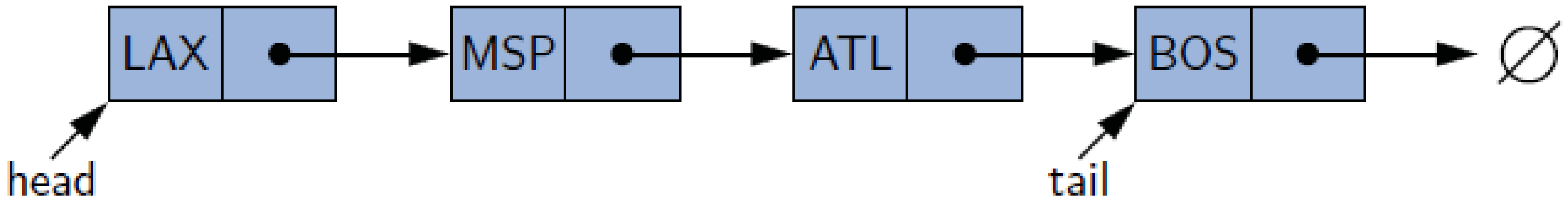
$$\sum_{i=1}^{n} i^k \text{ is } \Theta(n^{k+1}).$$

Another common summation is the *geometric sum*, $\sum_{i=0}^{n} a^i$, for any fixed real number $0 < a \neq 1$.

SINGLY LINKED LIST

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





INSERTING AT THE HEAD

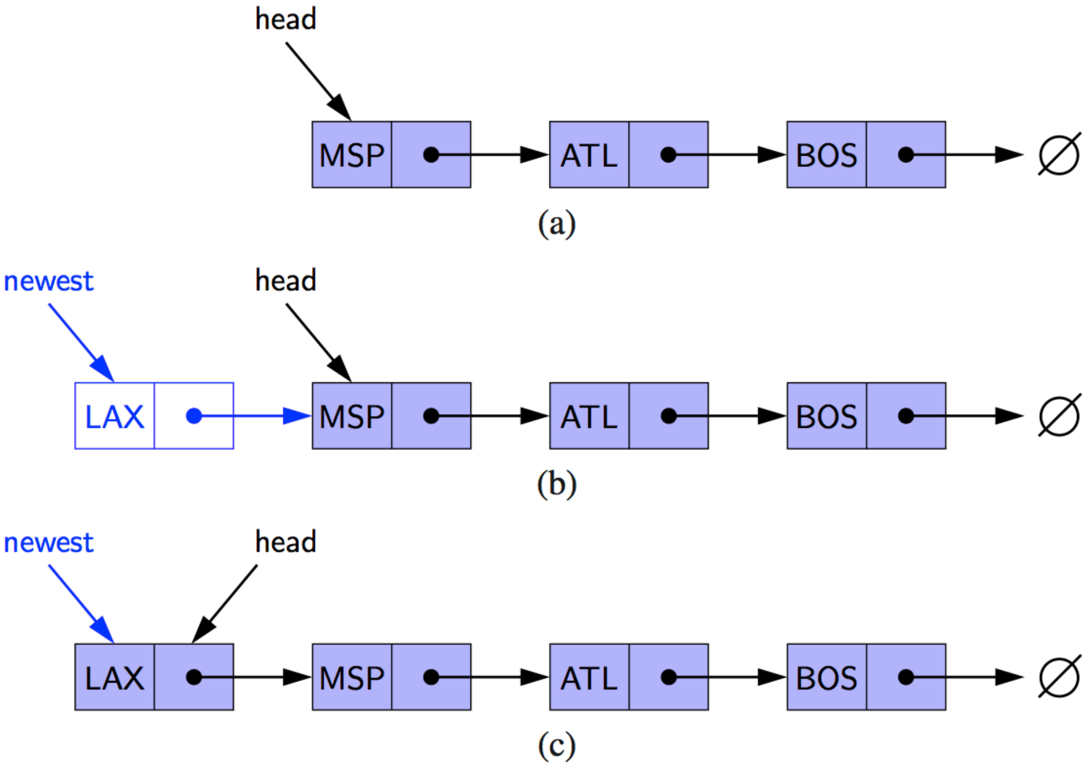
```
Algorithm addFirst(e):
    newest = Node(e)        {create new node instance storing reference to element e}
    newest.next = head      {set new node's next to reference the old head node}
    head = newest           {set variable head to reference the new node}
    size = size + 1         {increment the node count}
```

Allocate new node

Insert new element

Have new node point to old head

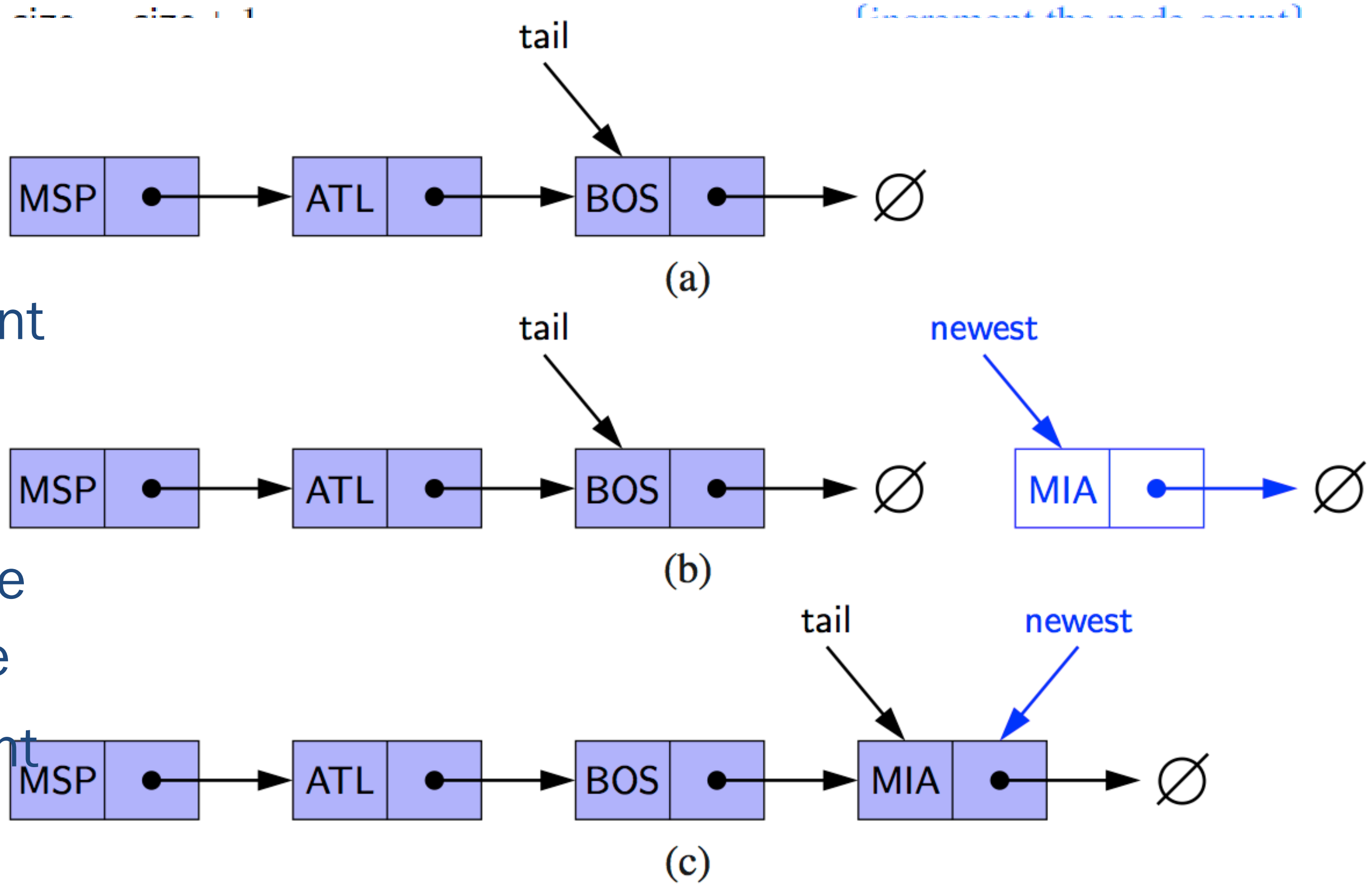
Update head to point to new node



INSERTING AT THE TAIL

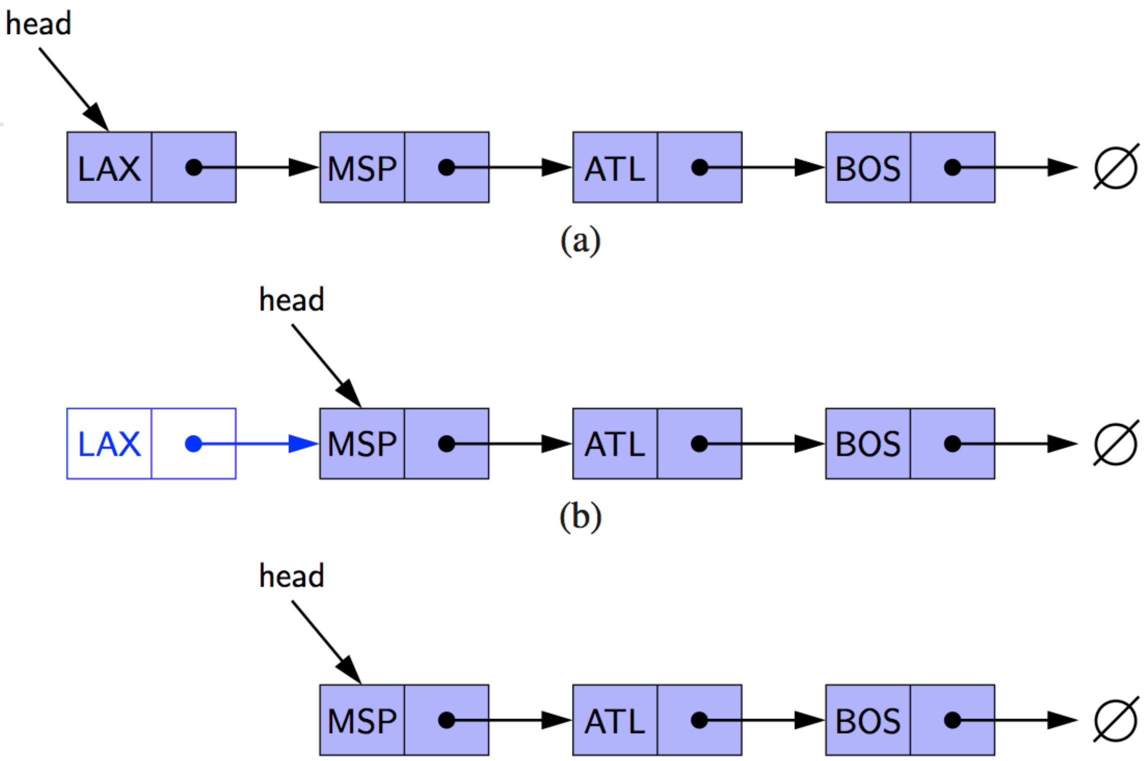
```
Algorithm addLast(e):
    newest = Node(e)  {create new node instance storing reference to element e}
    newest.next = null          {set new node's next to reference the null object}
    tail.next = newest                {make old tail node point to new node}
    tail = newest              {set variable tail to reference the new node}
    size = size + 1                          {increment the node count}
```

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



REMOVING AT THE HEAD

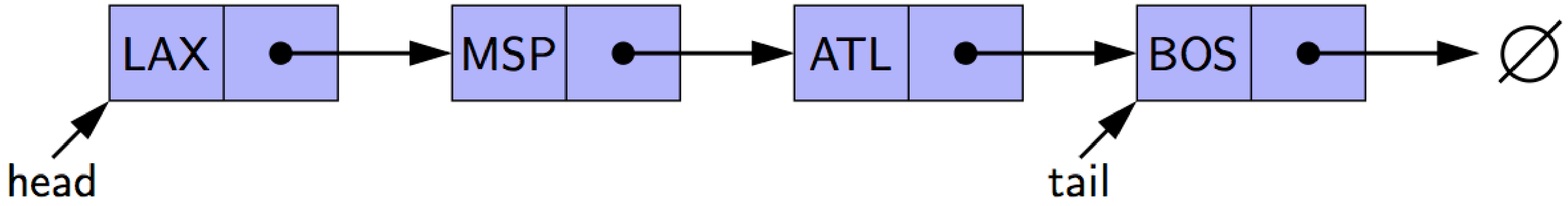
- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



```
public E removeFirst( ) {                    // removes and returns the first element
  if (isEmpty( )) return null;               // nothing to remove
  E answer = head.getElement( );
  head = head.getNext( );                    // will become null if list had only one node
  size−−;
  if (size == 0)
    tail = null;                             // special case as list is now empty
  return answer;
}
}
```

REMOVING AT THE TAIL

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



CIRCULARLY LINKED LIST

- A singularly linked list in which the next reference of the tail node is set to refer back to the head of the list (rather than null).
- Supports all of the public behaviors of our SinglyLinkedList class and one additional update method

rotate(): Moves the first element to the end of the list.

- Nodes store:
  - element
  - link to the next node

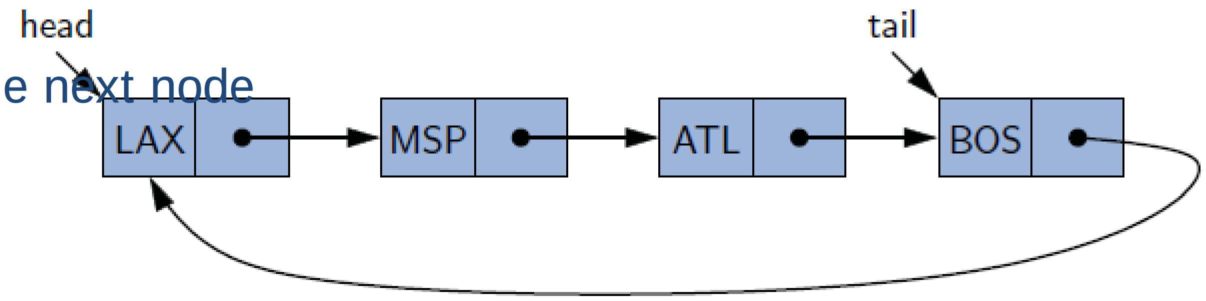
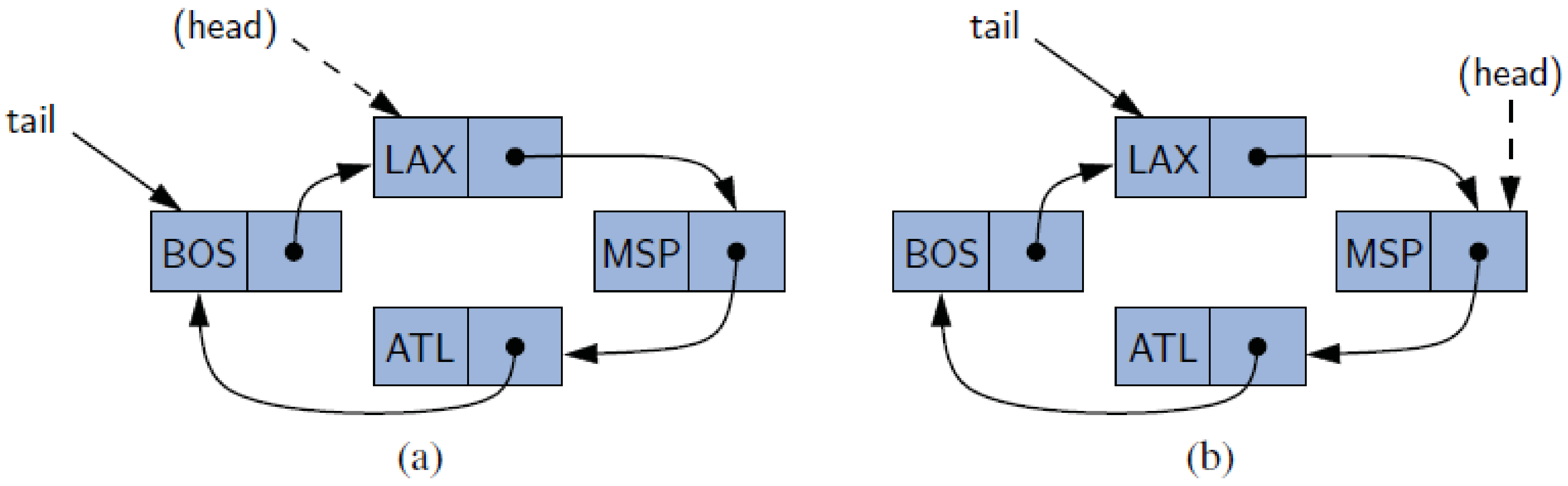


Figure 3.16: Example of a singly linked list with circular structure.

ROTATE() ON A CIRCULARLY LINKED LIST

We do not move any nodes or elements, we simply advance the tail reference to point to the node that follows it (the implicit head of the list).

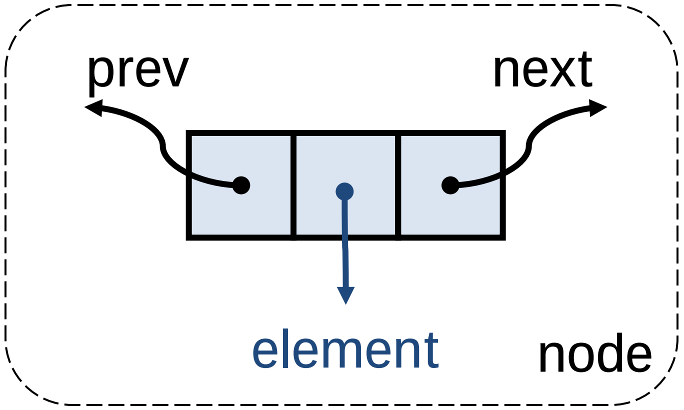
implicit head: tail.getNext( ).

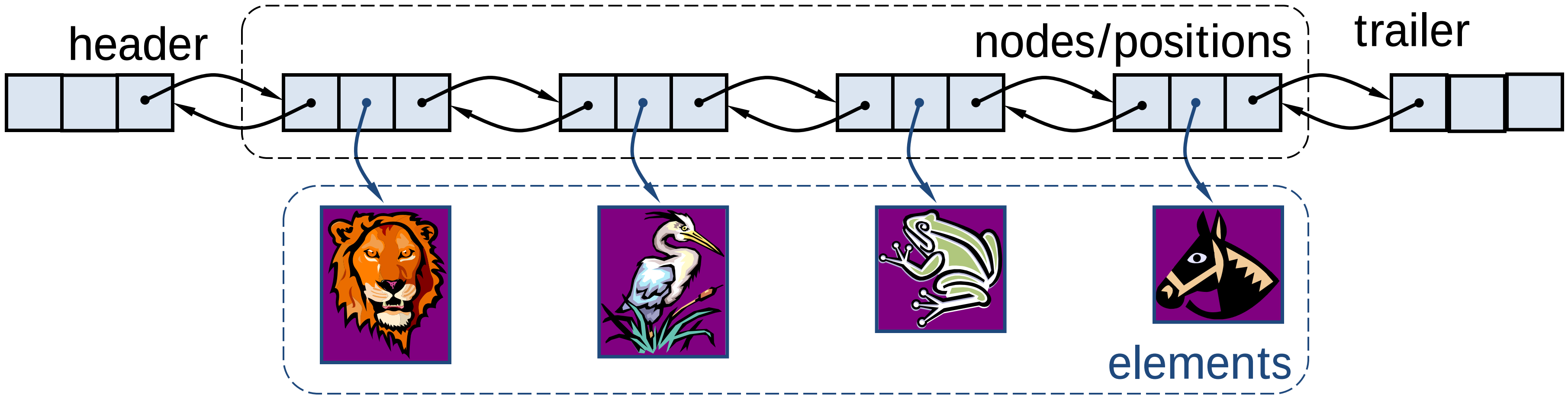


```
public void rotate() {          // rotate the first element to the back of the list
  if (tail != null)               // if empty, do nothing
    tail = tail.getNext();        // the old head becomes the new tail
}
```

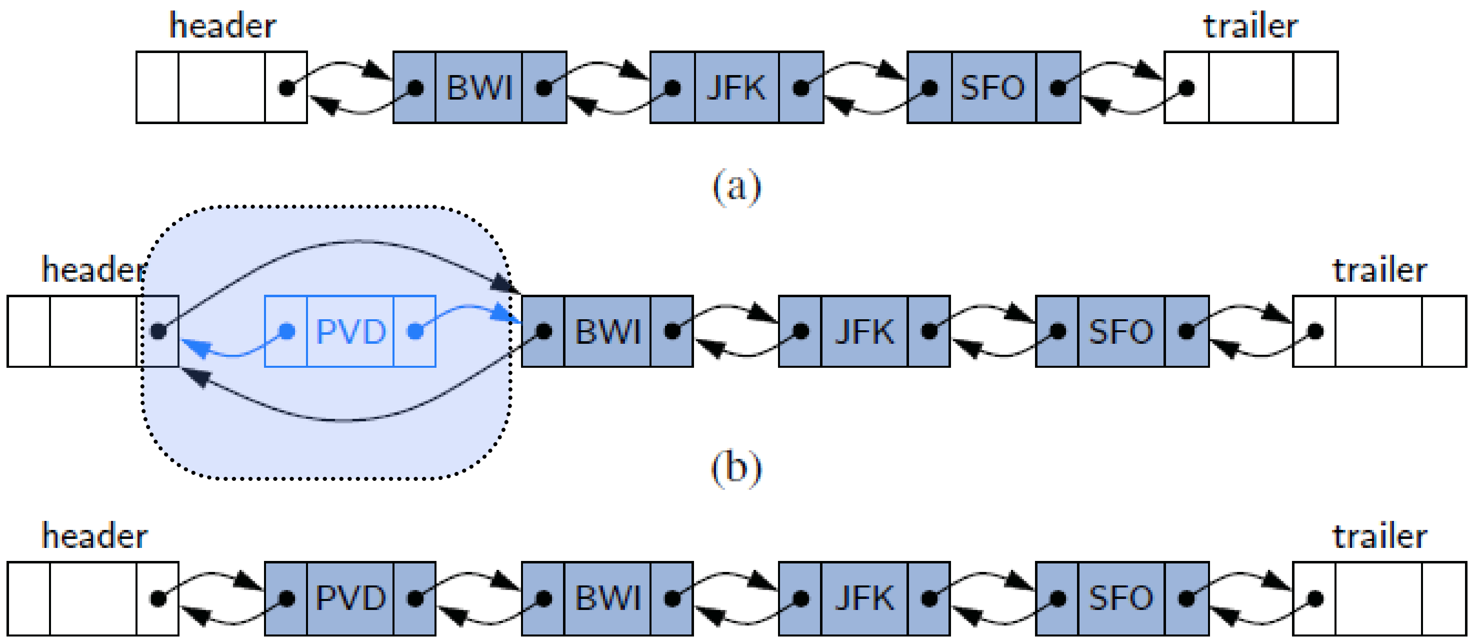
DOUBLY LINKED LIST

- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

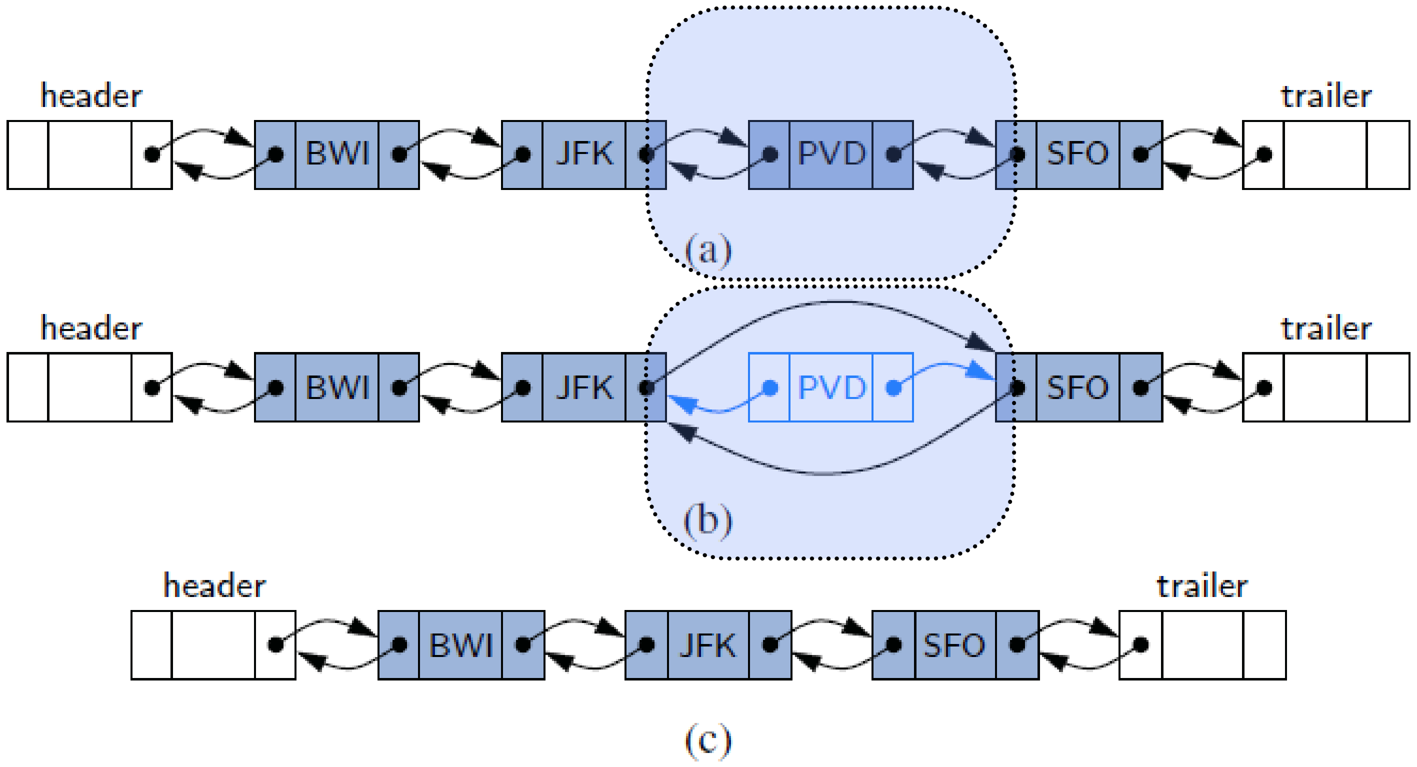




INSERTION IN DOUBLY LINKED LIST

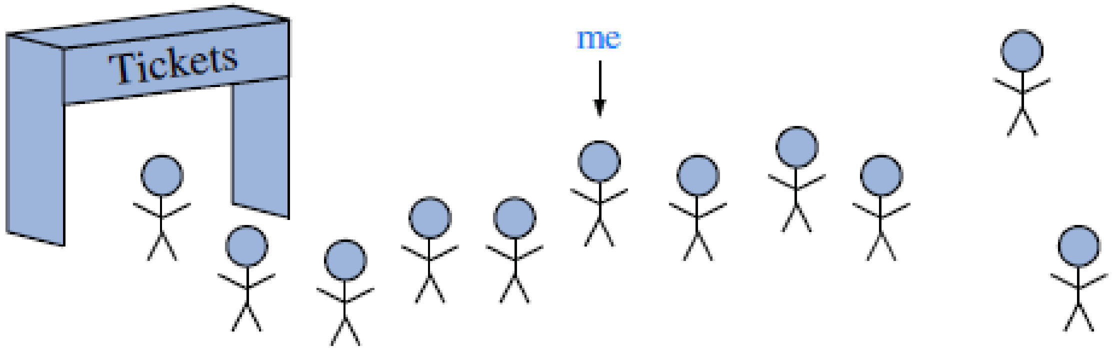


DELETION IN DOUBLY LINKED LIST



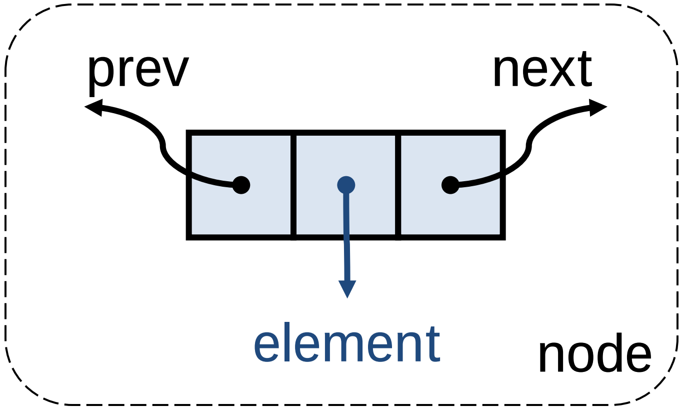
POSITIONAL LISTS

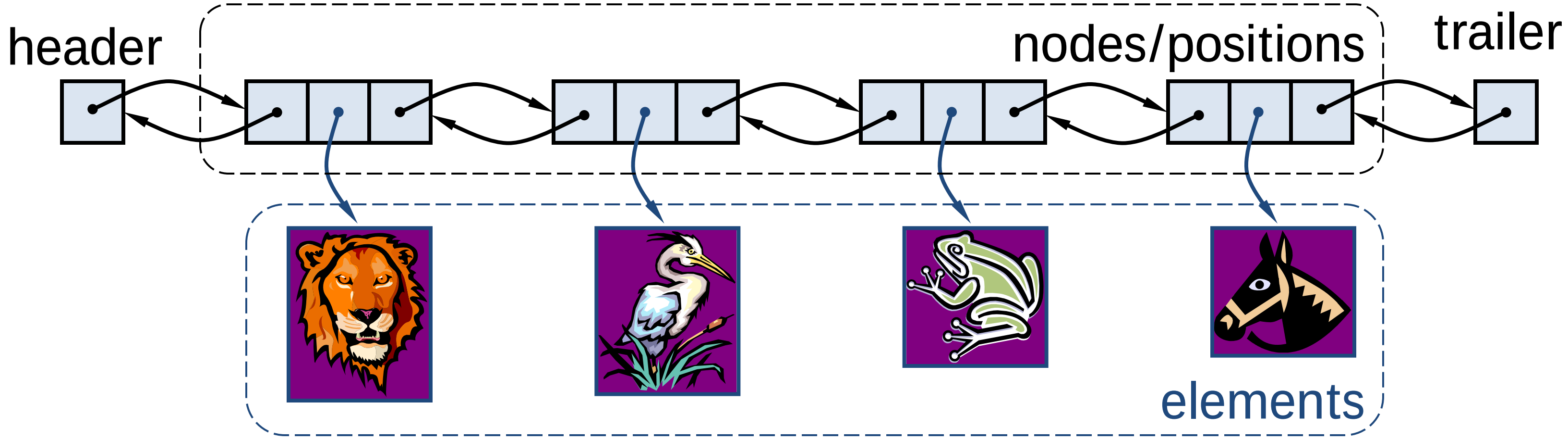
- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position ***p*** is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - P.getElement( ): Return the element stored at position p.



POSITIONAL LIST IMPLEMENTATION USING DOUBLY LIKED LIST

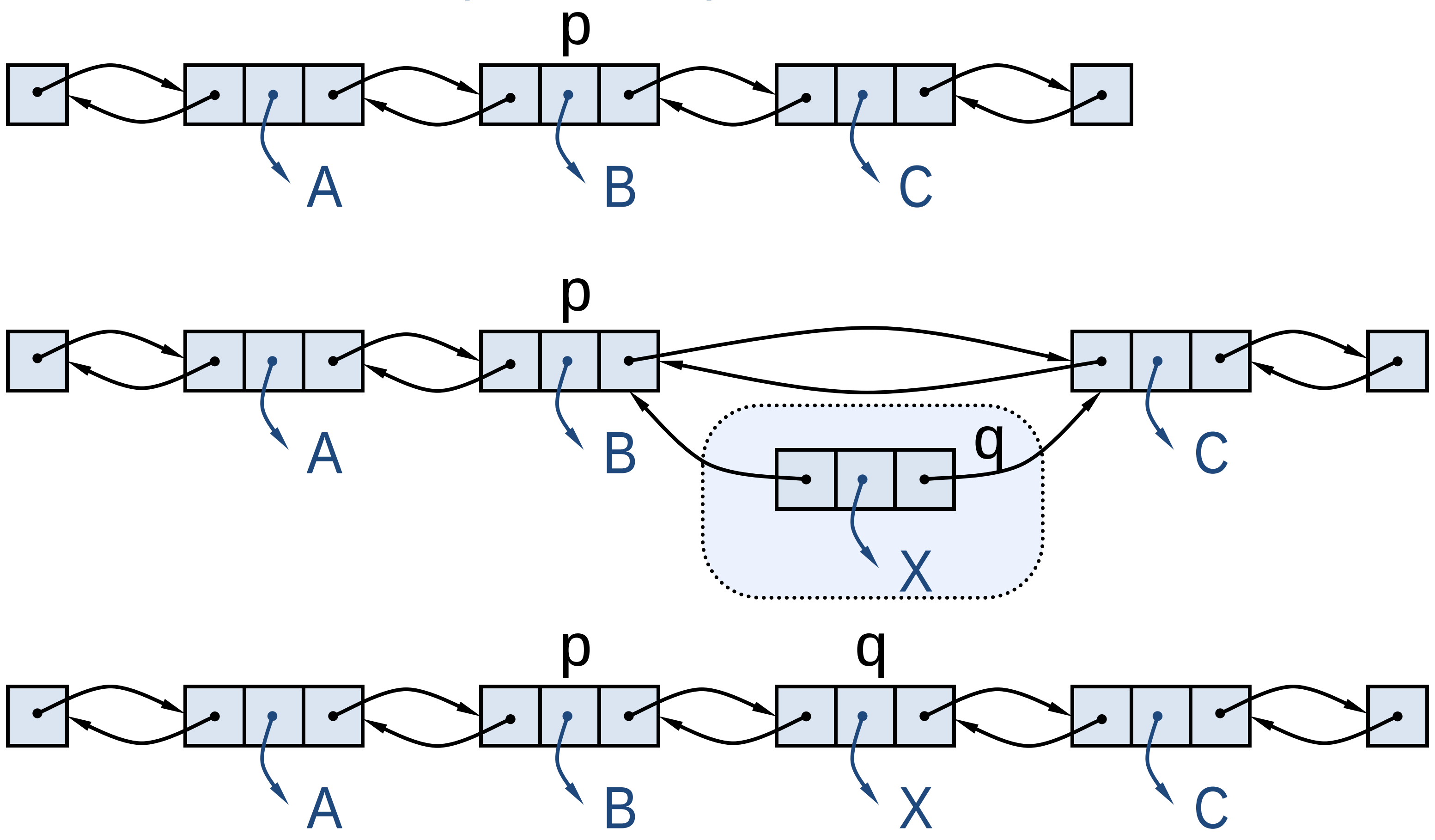
- The most natural way to implement a positional list is with a **doubly-linked list.**
- NOTE: Not the same as the DoublyLinkedList class in Ch3
  - Difference in the management of the positional abstraction





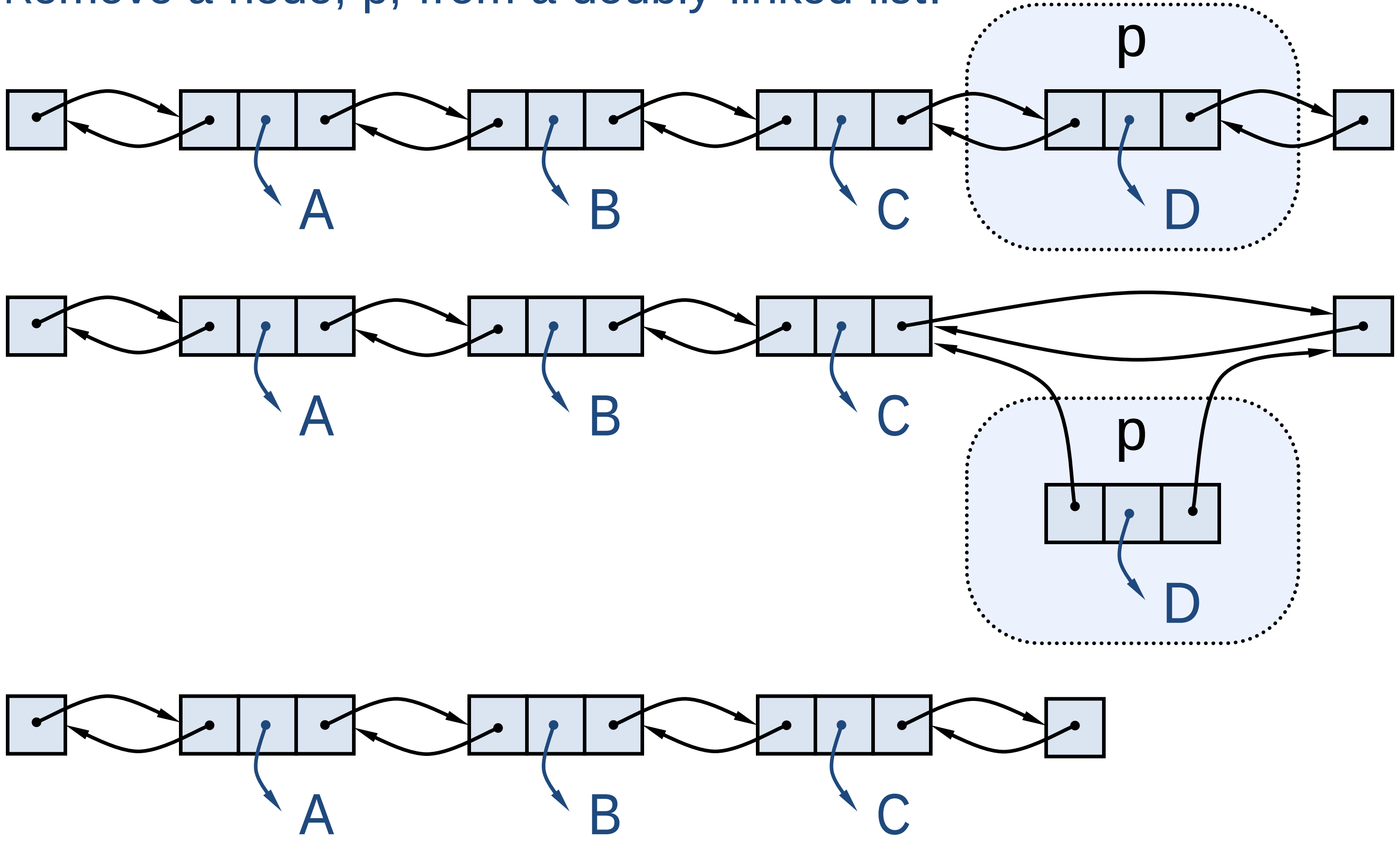
INSERTION

- Insert a new node, q, between p and its successor.



DELETION

- Remove a node, p, from a doubly-linked list.



BASIC DATASTRUCTURES

STACKS

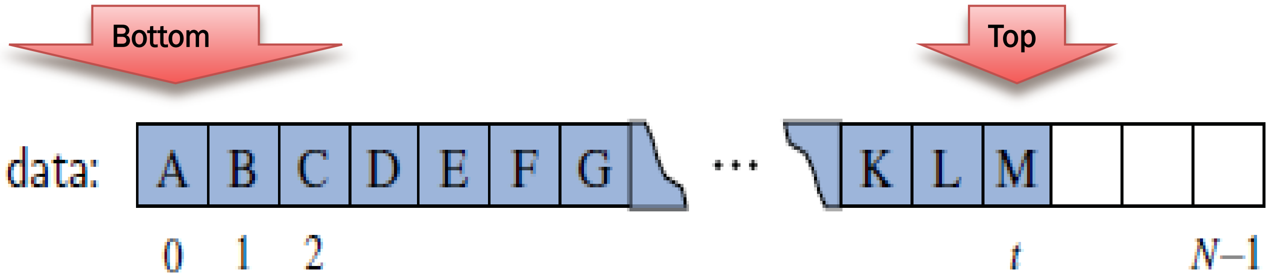
- Main stack operations:
  - **push(object):** inserts an element
  - **object pop():** removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean 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 (t) keeps track of the index of the top element

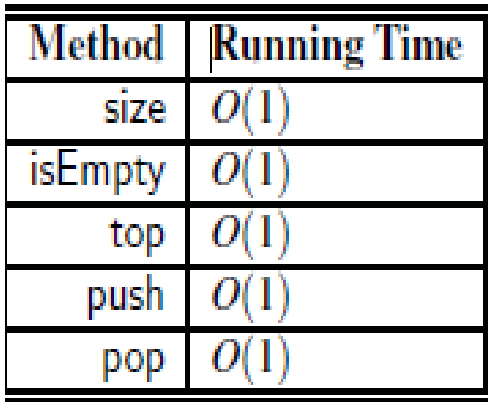
```
Algorithm size()
    return $t + 1$

Algorithm pop()
    if isEmpty() then
        return null
    else
        $t \leftarrow t - 1$
        return $S[t + 1]$
```



PERFORMANCE & LIMITATIONS OF ARRAY-BASED STACK

- Performance
  - Let n be the number of elements ir the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$



Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed (fixed size array)
  - Trying to push a new element into a full stack causes an implementation-specific exception

IMPLEMENTING A STACK WITH A SINGLY LINKED LIST

- The linked-list approach has memory usage that is always proportional to the number of actual elements currently in the stack, and without an arbitrary capacity limit
- Q: What the best choice for the top of the stack: the front or back of the list?
  - With the top of the stack stored at the front of the list, all methods execute in constant time.

ADAPTING SINGLE LINKED LIST ON STACK ADT

- We will adapt SinglyLinkedList class of Section 3.2.1 to define a new LinkedStack class

SinglyLinkedList is named *list* as a private field, and uses the following correspondences:

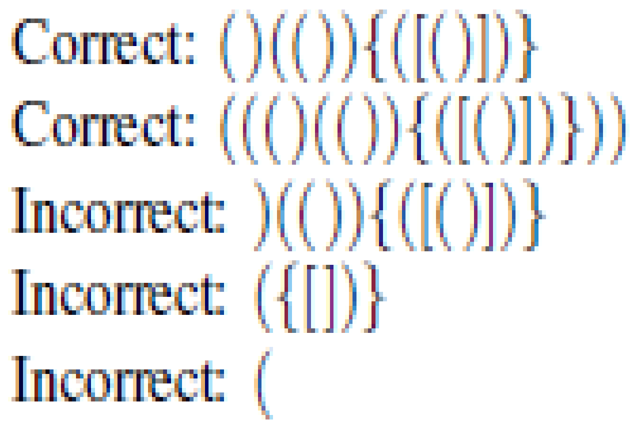
Stack Method	*Singly Linked List Method*
size()	list.size()
isEmpty()	list.isEmpty()
push(*e*)	list.addFirst(*e*)
pop()	list.removeFirst()
top()	list.first()

```
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedStack() { }                  // new stack relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(E element) { list.addFirst(element); }
  public E top() { return list.first(); }
  public E pop() { return list.removeFirst(); }
}
```

EXAMPLE: MATCHING PARENTHESES

- Consider arithmetic expressions that may contain various pairs of grouping symbols:
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”

[(5+*x*)−(*y*+*z*)



Correct: ()(()){([()])}

Correct: ((()(()){([()])}))

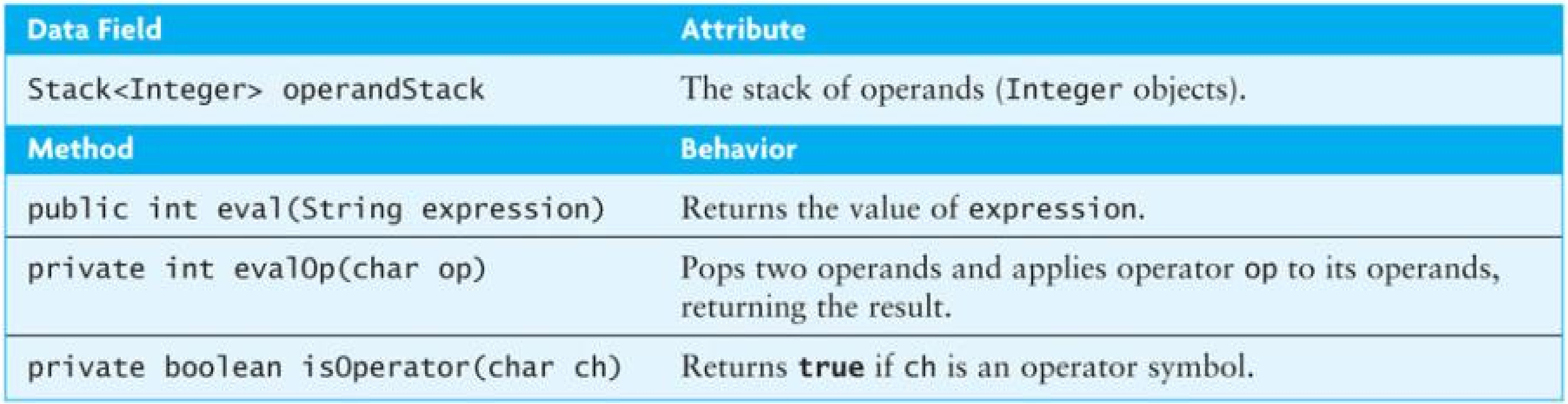
Incorrect:)(()){([()])}

Incorrect: ({[])}

Incorrect: (

EVALUATING POSTFIX EXPRESSIONS

- Write a class that evaluates a postfix expression
- Use the space character as a delimiter between tokens



Data Field	Attribute
`Stack<Integer> operandStack`	The stack of operands (`Integer` objects).
Method	**Behavior**
`public int eval(String expression)`	Returns the value of `expression`.
`private int evalOp(char op)`	Pops two operands and applies operator `op` to its operands, returning the result.
`private boolean isOperator(char ch)`	Returns **`true`** if `ch` is an operator symbol.

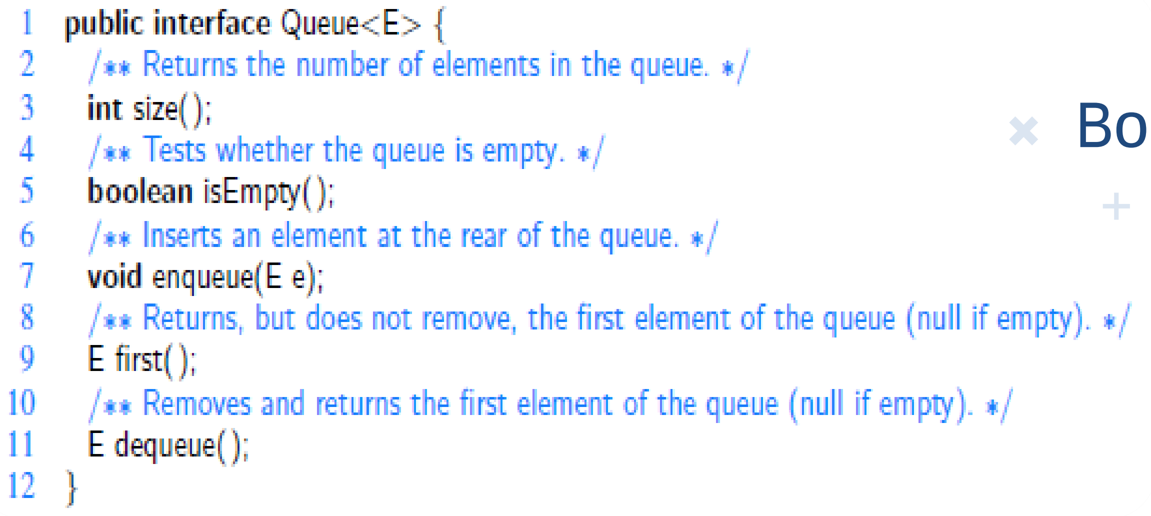
QUEUE

- The Queue ADT stores arbitrary objects
- The queue, like the stack, is a widely used data structure
- A queue differs from a stack in one important way Insertions and deletions follow
  - A stack is LIFO list – *Last-In, First-Out*
  - while a queue is FIFO list, *First-In, First-Out*
- Insertions are at the rear of the queue and removals are at the front of the queue

Know about application of queues and their base operations.

THE QUEUE ADT

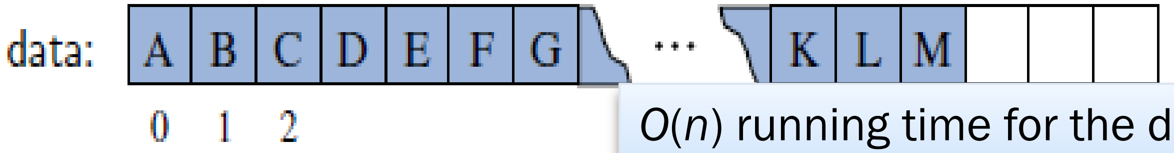
- Main queue operations:
  - **enqueue(object):** inserts an element at the <u>end</u> of the queue
  - **object dequeue():** removes and returns the element at the <u>front</u> of the queue
- Auxiliary queue operations:
  - object **first**(): returns the element at the front without removing it
  - integer **size**(): returns the number of elements stored
  - boolean **isEmpty**(): indicates whether no elements are stored
- Boundary cases:
  - Attempting the execution of dequeue or first on an empty queue returns null



```
public interface Queue<E> {
  /** Returns the number of elements in the queue. */
  int size();
  /** Tests whether the queue is empty. */
  boolean isEmpty();
  /** Inserts an element at the rear of the queue. */
  void enqueue(E e);
  /** Returns, but does not remove, the first element of the queue (null if empty). */
  E first();
  /** Removes and returns the first element of the queue (null if empty). */
  E dequeue();
}
```

ARRAY-BASED QUEUE 1

Using an array to store elements of a queue, such that the first element inserted, “A”, is at cell 0, the second element inserted, “B”, at cell 1, and so on.

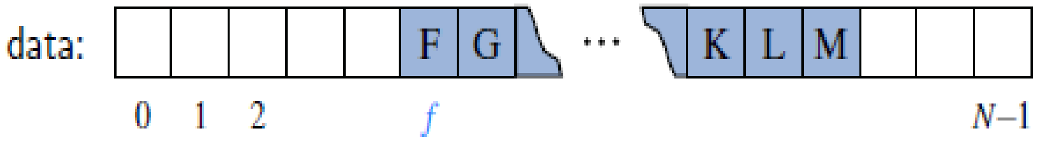


O(*n*) running time for the dequeue method.

ARRAY-BASED QUEUE 2

- Replace a dequeued element in the array with a null reference
- A variable to keep track of the front
  - *f* index of the front element

O(1) deque operation but, if we repeatedly let the front of the queue drift rightward over time, the back of the queue would reach the end of the underlying array even when there are fewer than *N* elements currently in the queue.



ARRAY-BASED QUEUE 3: CIRCULAR ARRAY

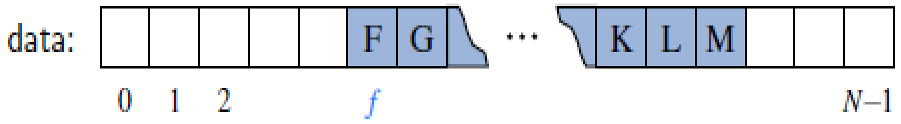
- Use an array of size N in a circular fashion
- Two variables keep track of the front and size

  f index of the front element

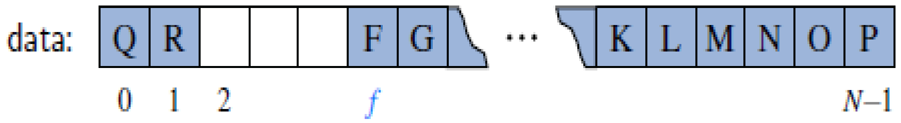
  sz number of stored elements

- How to store additional elements in such a configuration:
  - When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

normal configuration



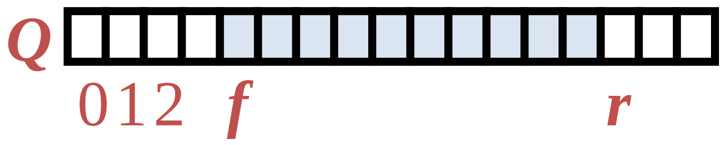
wrapped-around configuration

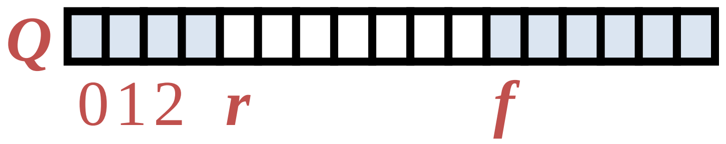


QUEUE OPERATIONS (CONT.): **ENQUEUE**

- Operation **enqueue** throws an exception if the array is full
- This exception is implementation-dependent

```
Algorithm enqueue(o)
  if size() = N – 1 then
    throw IllegalStateException
  else
    r ← (f + sz) mod N
    Q[r] ← o
    sz ← (sz + 1)
```



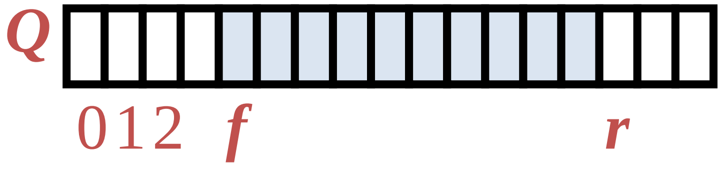


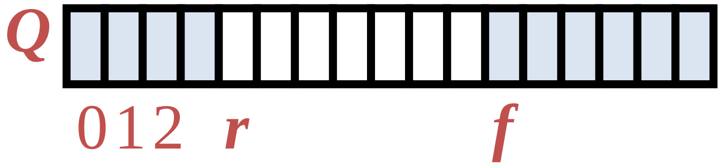
avail = (f + sz) % data.length;

QUEUE OPERATIONS (CONT.): **DEQUEUE**

- Note that operation **dequeue** returns NULL if the queue is empty

```
Algorithm dequeue()
  if isEmpty() then
    return null
  else
    o ← Q[f]
    f ← (f + 1) mod N
    sz ← (sz – 1)
    return o
```



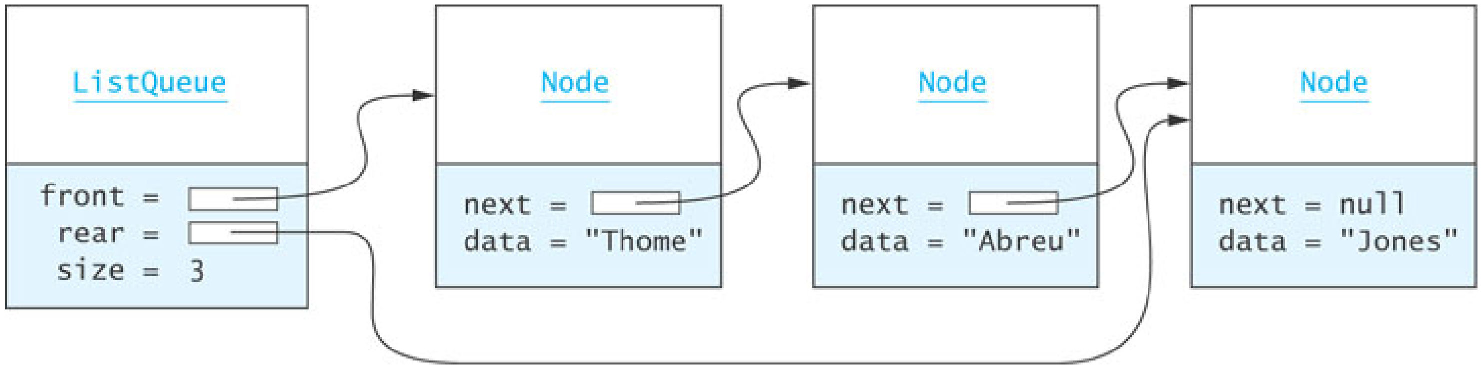


f = (f+1) % data.length

IMPLEMENTING A QUEUE WITH A SINGLY LINKED LIST

- Supporting worst-case O(1)-time for all operations, and without any artificial limit on the capacity
- Orientation for Queue using singly linked list
  - Align the front of the queue with the front of the list,
  - Align the back of the queue with the tail of the list,

  (because the only update operation that singly linked lists support at the back end is an insertion)



DOUBLE-ENDED QUEUE: DEQUE

- A deque (pronounced "deck") is short for double-ended queue
- A double-ended queue allows insertions and removals from both ends
- The deque abstract data type is more general than both the stack and the queue ADTs.

SETS

- Consider another part of the Collection hierarchy: the Set interface
- A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.
  - Elements of a set are like keys of a map, but without any auxiliary values.

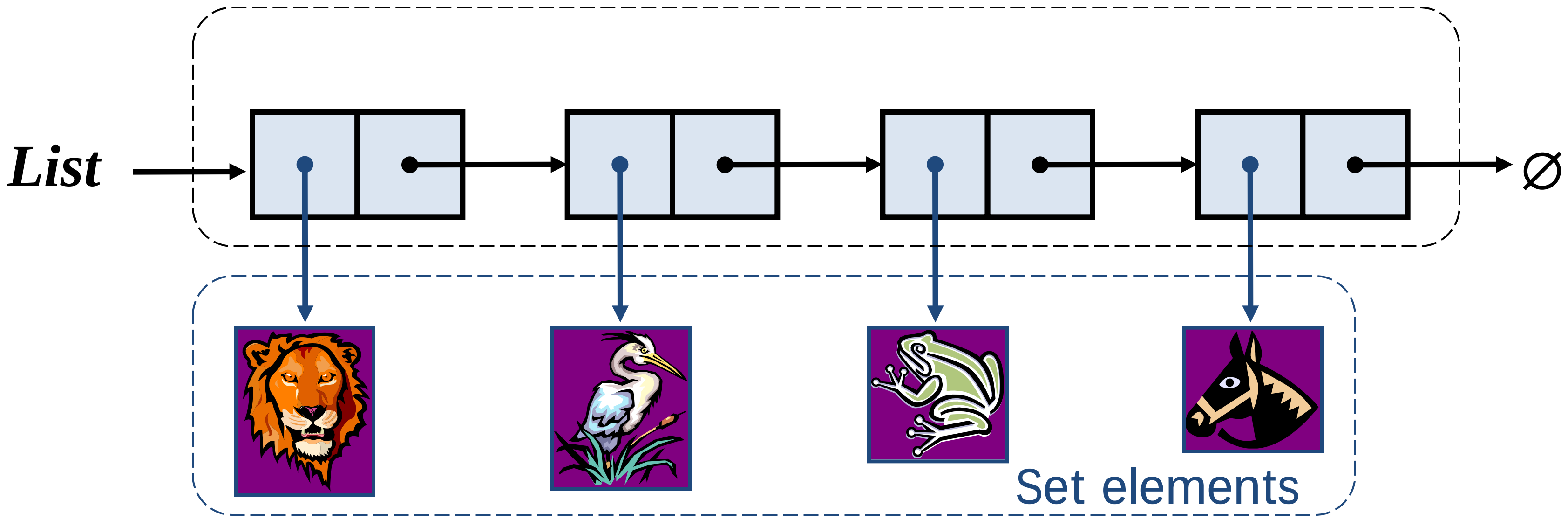
add(e):	Adds the element e to S (if not already present).
remove(e):	Removes the element e from S (if it is present).
contains(e):	Returns whether e is an element of S.
iterator( ):	Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

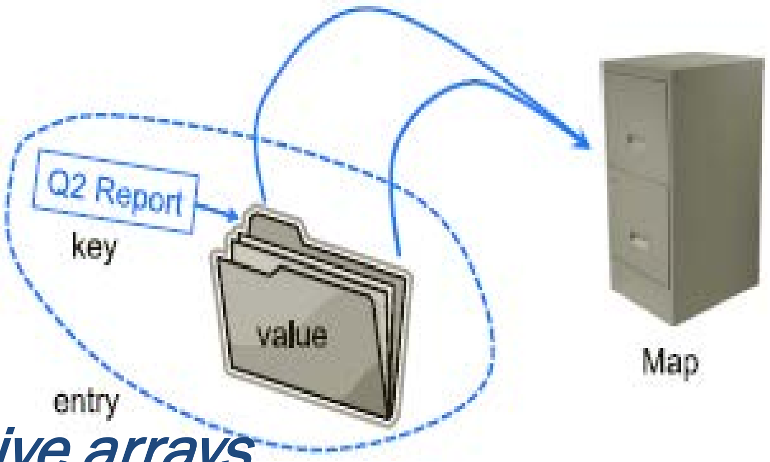
$$\begin{aligned} S \cup T &= \{e\colon\ e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e\colon\ e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e\colon\ e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

STORING A SET IN A LIST

- We can implement a set with a list
- The space used is $O(n)$



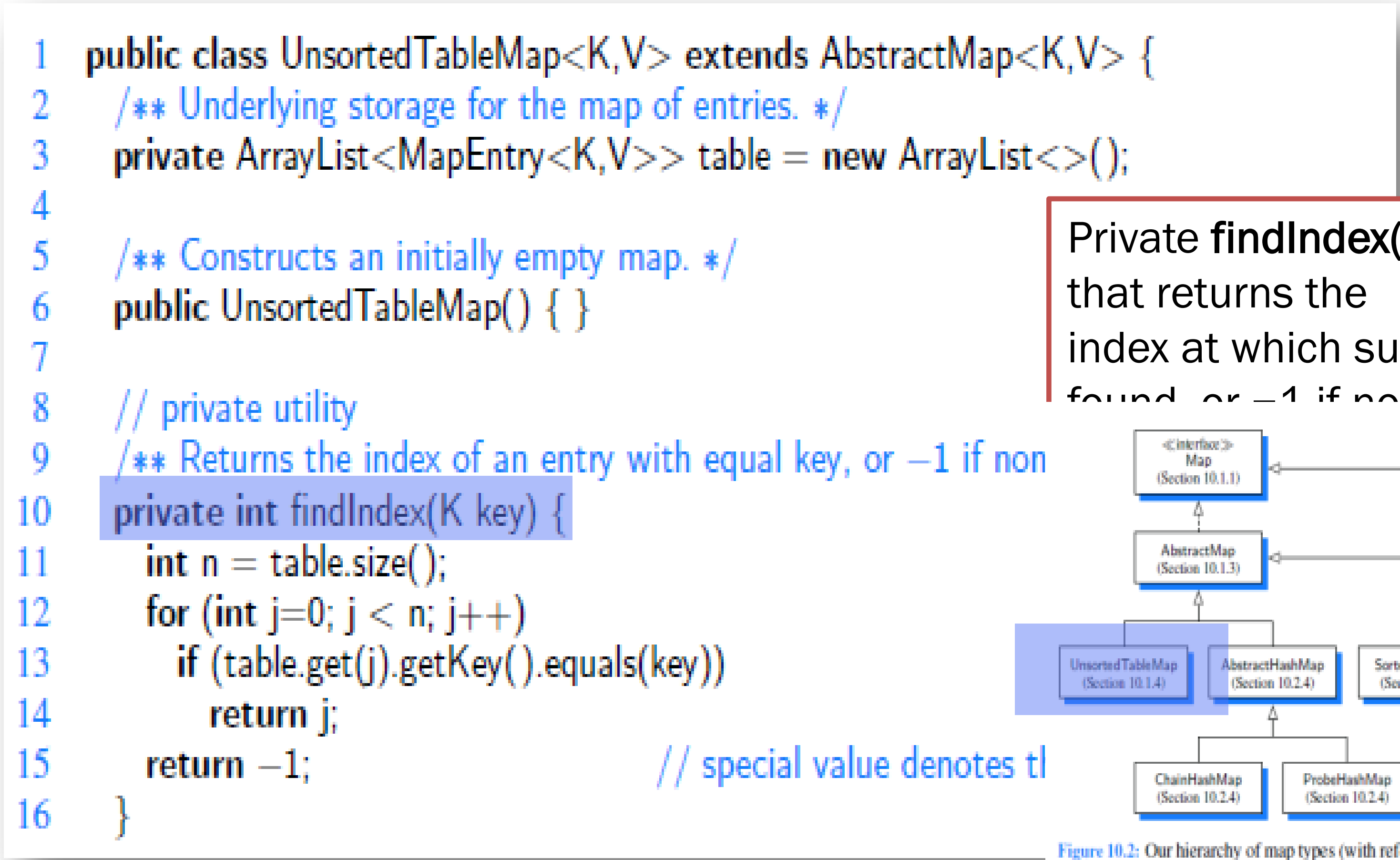
MAPS



- A **map** models a searchable collection of **key-value** pairs (*k*,*v*), which we call ***entries***
  - Keys are required to be unique
- Maps are also known as ***associative arrays***,
  - entry's key serves somewhat like an index into the map, in that it assists the map in efficiently locating the associated entry.
- Unlike a standard array, a **key** of a map need not be numeric, and is does not directly designate a position within the structure.
- The `Map` is related to the `Set,` mathematically, a `Map` is a set of ordered pairs whose elements are known as the key and the value
- The main operations are for searching, inserting, and deleting items

A SIMPLE UNSORTED MAP IMPLEMENTATION: UNSORTEDTABLEMAP

The use of the AbstractMap class with a very simple concrete implementation of the map ADT that relies on storing key-value pairs in arbitrary order within a Java **ArrayList.**



```
public class UnsortedTableMap<K,V> extends AbstractMap<K,V> {
  /** Underlying storage for the map of entries. */
  private ArrayList<MapEntry<K,V>> table = new ArrayList<>();

  /** Constructs an initially empty map. */
  public UnsortedTableMap() { }

  // private utility
  /** Returns the index of an entry with equal key, or −1 if non
  private int findIndex(K key) {
    int n = table.size();
    for (int j=0; j < n; j++)
      if (table.get(j).getKey().equals(key))
        return j;
    return −1;                    // special value denotes tl
  }
```

Private **findIndex(key**) method that returns the index at which such an entry is found, or −1 if no such entry is

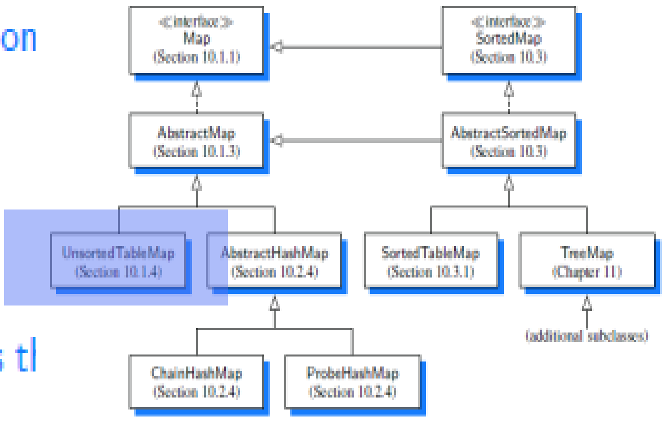

«interface» Map (Section 10.1.1)
«interface» SortedMap (Section 10.3)
AbstractMap (Section 10.1.3)
AbstractSortedMap (Section 10.3)
UnsortedTableMap (Section 10.1.4)
AbstractHashMap (Section 10.2.4)
SortedTableMap (Section 10.3.1)
TreeMap (Chapter 11)
(additional subclasses)
ChainHashMap (Section 10.2.4)
ProbeHashMap (Section 10.2.4)

Figure 10.2: Our hierarchy of map types (with references to where they are defined).

SIMPLE IMPLEMENTATION OF A SORTED MAP

Sorted search table: Store the map's entries in an array list *A* so that they are in increasing order of their keys.

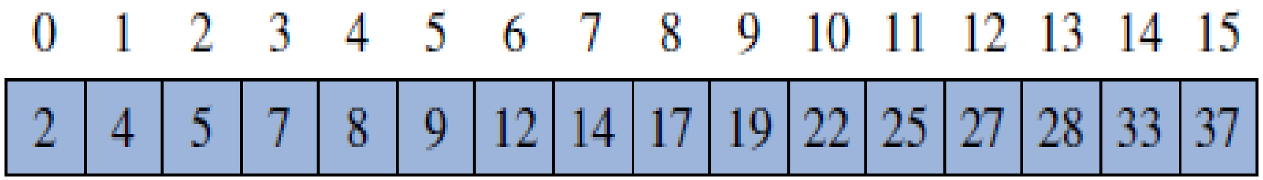


Figure 10.8: Realization of a map by means of a sorted search table. We show only the keys for this map, so as to highlight their ordering.

- the sorted search table has a space requirement that is $O(n)$.
- array-based implementation allows us to use the ***binary search*** algorithm for a variety of efficient operations.

FINDINDEX METHOD

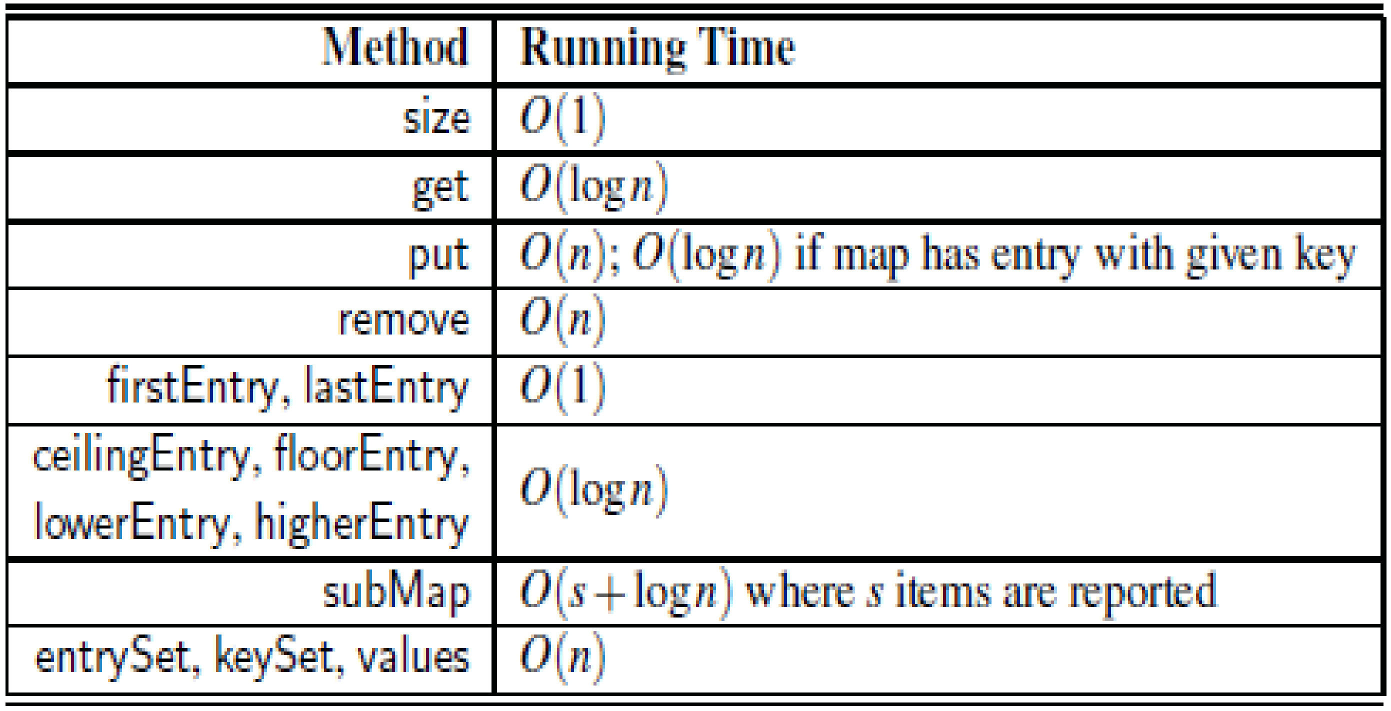
findIndex method uses the **recursive binary search algorithm**,

- returns the *index* of the leftmost entry in the search range having key greater than or equal to *k*;
- if no entry in the search range has such a key, we return the index just beyond the end of the search range.

$\Rightarrow$ If an entry has the target key, the search returns the index of that entry. (Recall that keys are unique in a map.)

$\Rightarrow$ If the key is absent, the method returns the index at which a new entry with that key would be inserted

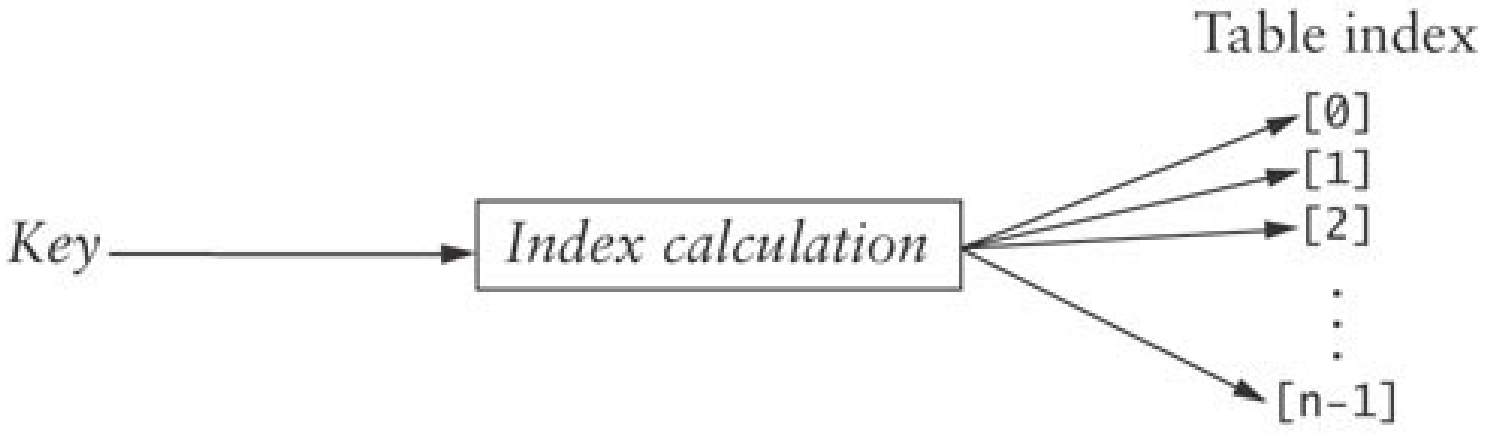
ANALYSIS OF OUR SORTEDTABLEMAP



Method	Running Time
size	$O(1)$
get	$O(\log n)$
put	$O(n)$; $O(\log n)$ if map has entry with given key
remove	$O(n)$
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$ where s items are reported
entrySet, keySet, values	$O(n)$

HASH CODES AND INDEX CALCULATION

- The basis of hashing is to transform the item's key value into an integer value (its *hash code*) which is then transformed into a table index



HASH FUNCTIONS

- The goal of a ***hash function***, *h*, is to map each key *k* to an integer in the range [0, *N* − 1], where *N* is the capacity of the bucket array for a hash table.
- A hash function is usually specified as the composition of two functions:
  - **Hash code** (independent of hash table size – allow generic implementation):
    h_1: keys $\rightarrow$ integers
  - **Compression function** (dependent of hash table size):
    h_2: integers $\rightarrow$ $[0, N - 1]$

Hash Codes: Bit Representation as an Integer

- Integer cast:
  - Java relies on 32-bit hash codes
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - Partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum (or exclusive-or) the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and & double in Java)

Both not good for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0,x_1,\ldots,x_{n-1})$, where the order of the x_i's is significant. Ex> "stop", "tops", "pots", and "spot".

POLYNOMIAL HASH CODES

A **polynomial hash code** takes into consideration the positions of the x_i's by using multiplication by different powers as a way to spread out the influence of each component across the resulting hash code.

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

    $$x_0a^{n-1} + x_1a^{n-2} + \cdots + x_{n-2}a + x_{n-1}.$$

    where a!=1 is a nonzero constant, ignoring overflows
- Especially suitable for strings
  - (33, 37, 39, and 41 are particularly good choices for *a* when working with character strings that are English words.)

- Polynomial $p(a)$ 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

    $p_0(a) = x_{n-1}$

    $p_i(a) = x_{n-i-1} + ax_{i-1}(a)$

    $(i = 1, 2, \ldots, n-1)$
- We have $p(a) = p_{n-1}(a)$

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \cdots + a(x_2 + a(x_1 + ax_0))\cdots)).$$

CYCLIC-SHIFT HASH CODES

- A variant of the polynomial hash code replaces multiplication by *a* with a cyclic shift of a partial sum by a certain number of bits.
  - Ex.> 5-bit cyclic shift of the 32-bit 00111101100101101010100010101000 is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in 10110010110101010001010100000111.

```
static int hashCode(String s) {
  int h=0;
  for (int i=0; i<s.length(); i++) {
    h = (h << 5) | (h >>> 27);              // 5-bit cyclic shift of the running sum
    h += (int) s.charAt(i);                 // add in next character
  }
  return h;
}
```

Shift	Collisions	
	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

CYCLIC-SHIFT HASH CODES

- A variant of the polynomial hash code replaces multiplication by *a* with a cyclic shift of a partial sum by a certain number of bits.
  - Ex.> 5-bit cyclic shift of the 32-bit 00111101100101101010100010101000 is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in 10110010110101010001010100000111.

```
static int hashCode(String s) {
  int h=0;
  for (int i=0; i<s.length(); i++) {
    h = (h << 5) | (h >>> 27);          // 5-bit cyclic shift of the running sum
    h += (int) s.charAt(i);             // add in next character
  }
  return h;
}
```

Shift	Collisions	
	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

COMPRESSION FUNCTIONS

- **Compression function** maps integer hash code i into an integer in the range of [0, N-1]

- A good compression function: probability any two different keys collide is 1/*N*.
  - If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is 1/*N*.

- Methods:
  - Multiplication method
  - MAD method

COMPRESSION FUNCTION: DIVISION METHOD

- Function:

  $h_2(i) = i \bmod N$ (i: the hash code)
- The size N of the hash table is usually chosen to be a prime
  - Prime numbers are shown to helps "spread out" the distribution of hashed values.
  - Example: if we insert keys with hash codes {200,205,210,215,220,...,600} into a bucket array of size 100, then each hash code will collide with three others. But if we use a bucket array of size 101, then there will be no collisions.
  - The reason has to do with number theory and is beyond the scope of this course
- Choosing N to be a prime number is not always enough
  - If there is a repeated pattern of hash codes of the form $pN + q$ fo r several different *prime numbers, p*, then there will still be collis ions.

COMPRESSION FUNCTION: MAD METHOD

- *Multiply-Add-and-Divide* (MAD) Method:

$$h_2(i) = [(ai + b) \bmod p] \bmod N$$

  - where *N* is the size of the hash table, *p* is a prime number larger than *N*, and *a* and *b* are integers chosen at random from the interval [0, *p* − 1], with *a* > 0.
- MAD is chosen in order to eliminate repeated patterns in the set of hash codes and get us closer to having a “good” hash function

COLLISION-HANDLING SCHEMES

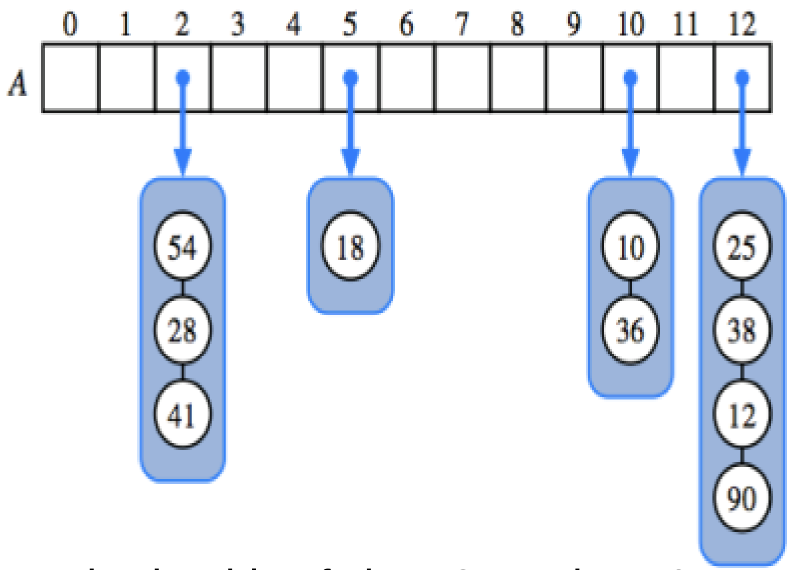
- The main idea of a hash table is to take a bucket array, *A*, and a hash function, *h*, and use them to implement a map by storing each entry (*k*,*v*) in the "bucket" *A*[*h*(*k*)].
- Even with a good hash function, collisions happen, i.e., two distinct keys, k_1 and k_2, such that $h(k_1) = h(k_2)$.
- Collisions
  - Prevents us from simply inserting a new entry (*k*,*v*) directly into the bucket *A*[*h*(*k*)]
  - Complicates our procedure for insertion, search, and deletion operations.
- Collision handling schemes:
  - Separate Chaining
  - Open Addressing
    - Linear Probing and Variants of Linear Probing

COLLISION-HANDLING SCHEMES: SEPARATE CHAINING

- Separate Chaining Scheme: have each bucket *A*[*j*] store its own secondary container, holding all entries (*k*,*v*) such that *h*(*k*) = *j*.

- Advantage: simple implementations of map operations

- Disadvantage: requires the use of an auxiliary data structure to hold entries with colliding keys



A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is *h*(*k*) = *k* mod 13. Values omitted.

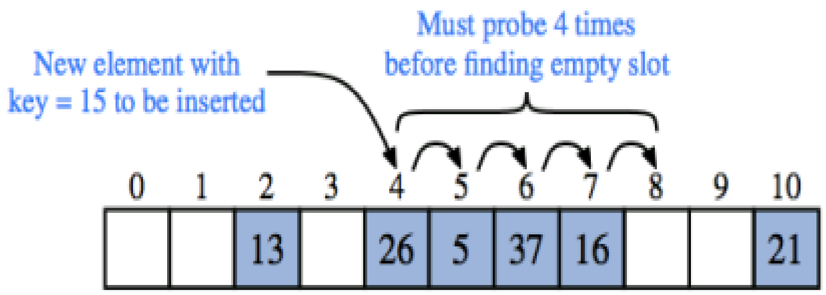
COLLISION-HANDLING SCHEMES: OPEN ADDRESSING

- Open Addressing: store each entry directly in a table slot.
  - This approach saves space because no auxiliary structures are employed
  - Requires a bit more complexity to properly handle collisions.
- Open addressing requires
  - Load factor is always at most 1
  - Entries are stored directly in the cells of the bucket array itself.

- EX> Linear Probing and Its Variants

COLLISION-HANDLING SCHEMES: LINEAR PROBING

- Deletion Scheme:
  - Cannot simply remove a found entry from its slot in the array
  - EX> after the insertion of key 15, if the entry with key 37 were trivially deleted, a subsequent search for 15 would fail because that search would start by probing at index 4, then index 5, and then index 6, at which an empty cell is found.


New element with key = 15 to be inserted
Must probe 4 times before finding empty slot
0 1 2 3 4 5 6 7 8 9 10
13 26 5 37 16 21

  - Resolve by replacing a deleted entry with a special "defunct" sentinel object.
    - Modify search algorithm so that the search for a key *k* will skip over cells containing the defunct
    - The put should remember a defunct locations during the search for *k*, and put the new entry (*k*,*v*), if no existing entry is found beyond it.

COLLISION-HANDLING SCHEMES: VARIANTS OF LINEAR PROBING

- Linear probing tends to cluster the entries of a map into contiguous runs, which may even overlap causing searches to slow down considerably.
- Avoiding Clustering with variant of Linear Probing:
  - **Quadratic Probing:** iteratively tries the buckets $A[(h(k)+ f(i)) \bmod N]$, for $i = 0,1,2,\ldots$, where $f(i) = i^2$, until finding an empty bucket.
  - **Double Hashing**: choose a secondary hash function, h', and if h maps some key k to a bucket $A[h(k)]$ that is already occupied (no clustering effect)

PROBLEMS WITH QUADRATIC PROBING

- Quadratic probing strategy complicates the removal o peration.
- It does avoid the kinds of clustering patterns that occ ur with linear probing but still suffers from secondary clustering
  - **Secondary Clustering:** set of filled array cells still has a non uniform pattern, even if we assume that the original hash c odes are distributed uniformly.
- Calculation of next index ((h(k)+ f(i)) mod N) is time-consuming, involving multiplication, addition, and modulo division

DOUBLE HASHING

- Open addressing strategy that does not cause clustering of the kind prod uced by linear probing or the kind pr oduced by quadratic probing
- Double hashing uses a secondary hash function $h'(k)$ and handles collisions by placing an item in the first available cel l of the series

  $(h(k) + i*h'(k)) \bmod N$

  for $i = 0, 1, \ldots, N \square 1$
- The table size <u>N must be a prime </u>to all ow probing of all the cells
- The secondary hash function $h'(k)$ cannot have zero values
- Common choice of compressio n function for the secondary h ash function:

  $h'(k) = q - (k \bmod q)$

  for <u>some prime $q < N$</u>.
- The possible values for $h'(k)$ ar e

  $1, 2, \ldots, q$

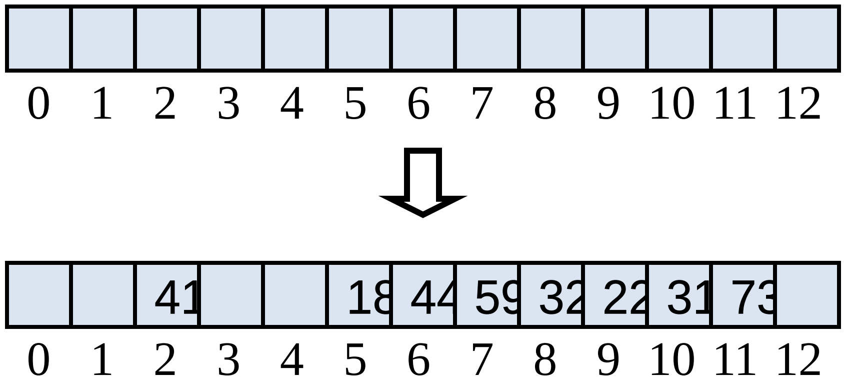
MAP WITH SEPARATE CHAINING

- To represent each bucket for separate chaining, we use an instance of the simpler UnsortedTableMap class.
- Entire hash table is then represented as a fixed-capacity array *A* of thevsecondary maps.
- Each cell, *A*[*h*], is initially a null reference;
- We only create a secondary map when an entry is first hashed to a particular bucket.

MAP WITH 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, causing future collisions to 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



PERFORMANCE OF HASH TABLES VERSUS SORTED ARRAY AND BINARY SEARCH TREE

- The number of comparisons required for a binary search of a sorted array is O(log *n*)
  - A sorted array of size 128 requires up to 7 probes (2^7 is 128) which is more than for a hash table of any size that is 90% full
  - A binary search tree performs similarly
- Insertion or removal

hash table	O(1) expected; worst case O(*n*)
unsorted array	O(*n*)
binary search tree	O(log *n*); worst case O(*n*)

WHAT IS A TREE

In computer science, a tree is an abstract model of hierarchical structure (a type of nonlinear data structure)

- Trees consists of nodes with a parent-child relation
- Trees also provide a natural or ganization for data,
  - Organization charts
  - File systems
  - Programming environment s

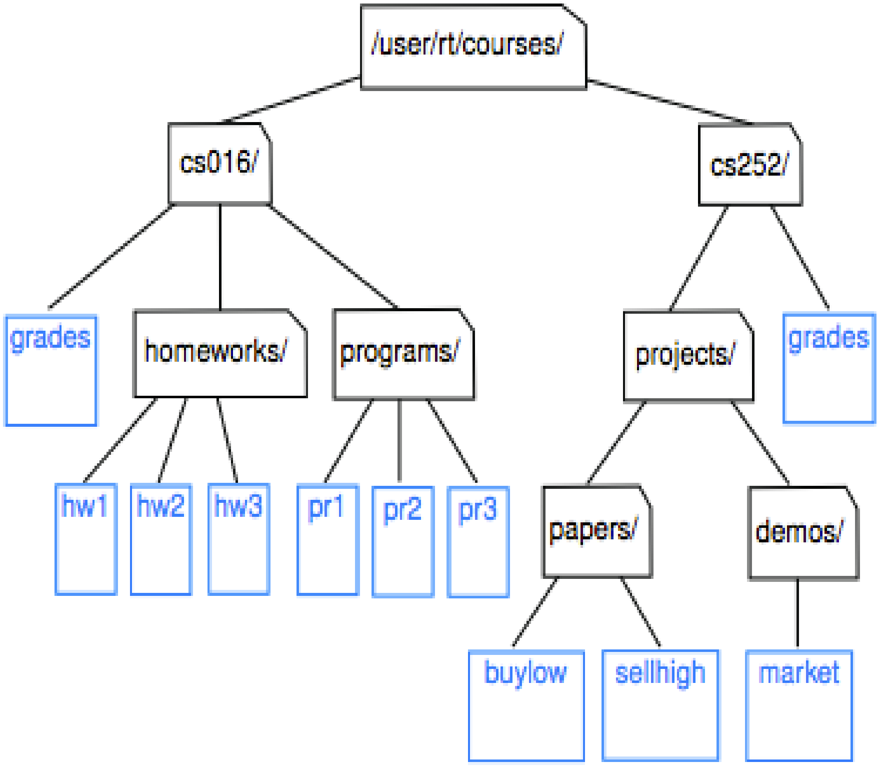
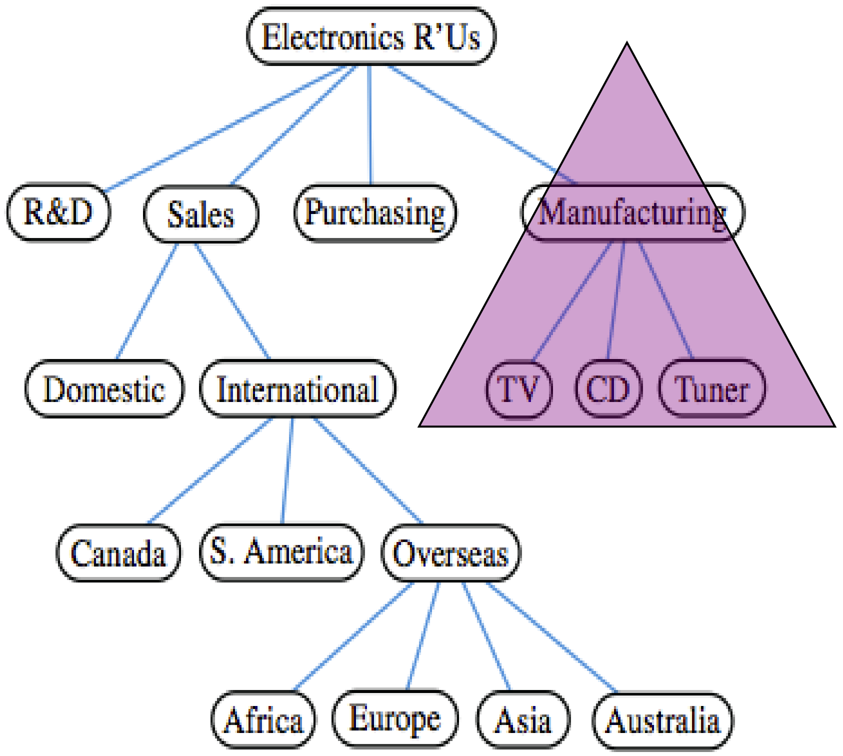


Figure 8.3: Tree representing a portion of a file system.

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.
- Subtree: tree consisting of a node and its descendants



organization of a fictitious corporation.

TREE TERMINOLOGY CONT

- Siblings: two nodes that are children of the same parent
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Edge: a pair of nodes (*u*,*v*) such that *u* is the parent of *v*, or vice versa.
- *Path:* a sequence of nodes such that any two consecutive nodes in the sequence form an edge

BINARY TREES

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- (Alternative recursive definition) A set of nodes T is a binary tree if either of the following is true
  - T is empty
  - Its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary trees

    (T_L = left subtree; T_R = right subtree)

TREE ADT

- We use positions as an abstraction for a node of a tree
- A position object for a tree supports the method:
  - getElement(): Returns the element stored at this position.
- ***Accessor methods*** for navigating through positions of a tree *T*
  - root(): Returns the position of the root of the tree (or null if empty).
  - parent(*p*): Returns the position of the parent of position *p* (or null if *p* is the root).
  - children(*p*): Returns an iterable collection containing the children of position *p* (if any).
  - numChildren(*p*): Returns the number of children of position *p*.

TREE ADT CONT.

- ***Query methods,*** **which are often used with** conditionals statements:
  - isInternal(*p*): Returns true if position *p* has at least one child.
  - isExternal(*p*): Returns true if position *p* does not have any children.
  - isRoot(*p*): Returns true if position *p* is the root of the tree.
- General methods, unrelated to the specific structure of the tree:
  - size(): Returns the number of positions (and hence elements) that are contained in the tree.
  - isEmpty(): Returns true if the tree does not contain any positions (and thus no elements).
  - iterator(): Returns an iterator for all elements in the tree (so that the tree itself is Iterable).
  - positions(): Returns an iterable collection of all positions of the tree.
- Additional update methods may be defined by data structures implementing the Tree ADT. (Discussed later)

A TREE INTERFACE IN JAVA

Methods for a Tree interface:

```
/** An interface for a tree where nodes can have an arbitrary number of children. */
public interface Tree<E> extends Iterable<E> {
  Position<E> root( );
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
                                    throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isExternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size( );
  boolean isEmpty( );
  Iterator<E> iterator( );
  Iterable<Position<E>> positions( );
}
```

Accessor methods

Query methods

General method s

COMPUTING DEPTH

- Let *p* be a position within tree *T*. The ***depth*** of *p* is the number of ancestors of *p*, other than *p* itself.
- The depth of *p* can also be recursively defined as follows:
  - If *p* is the root, then the depth of *p* is 0.
  - Otherwise, the depth of *p* is one plus the depth of the parent of *p*.

```
/** Returns the number of levels separating Position p from the root. */
public int depth(Position<E> p) {
  if (isRoot(p))
    return 0;
  else
    return 1 + depth(parent(p));
}
```

COMPUTING HEIGHT CONT

```
/** Returns the height of the subtree rooted at Position p. */
public int height(Position<E> p) {
  int h = 0;                                    // base case if p is external
  for (Position<E> c : children(p))
    h = Math.max(h, 1 + height(c));
  return h;
}
```

$O(n)$ worst-case time

- The overall height of a nonempty tree can be computed by sending the root of the tree as a parameter.

- Assuming that children(p) executes in $O(c_p + 1)$ time, where c_p denotes the number of children of p. Algorithm height(p) spends $O(c_p + 1)$ time at each position p to compute the maximum, and its overall running time is

$$O(\sum_p (c_p + 1)) = O(n + \sum_p c_p).$$

Let T be a tree with n positions, and let c_p denote the number of children of a position p of T. Then, summing over the positions of T, $\sum_p c_p = n - 1$.

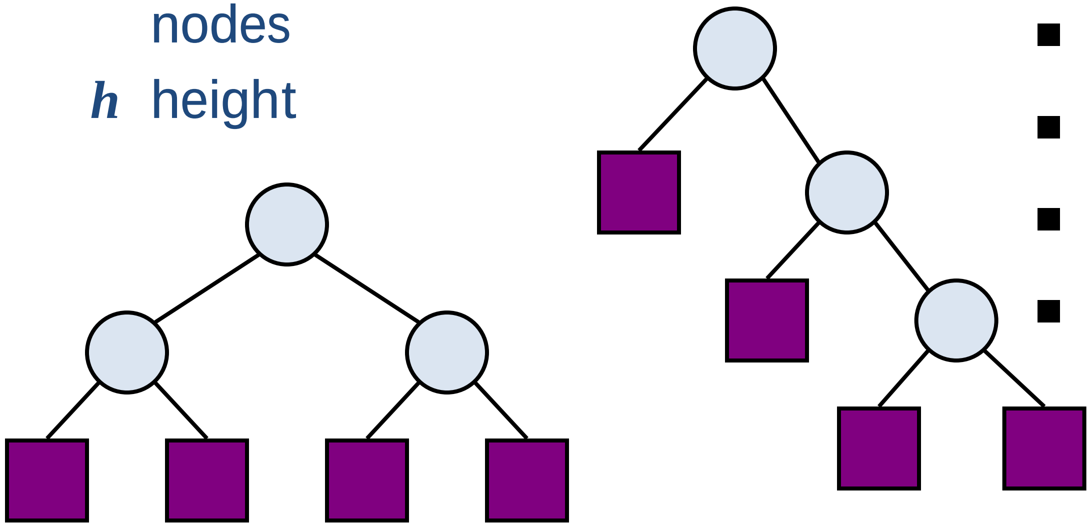
BINARY TREES

- A ***binary tree*** is an ordered tree with the following properties:
  - Every node has at most two children.
  - Each child node is labeled as being either a ***left child*** or a ***right child***.
  - A left child precedes a right child in the order of children of a node.
- The subtree rooted at a left or right child of an internal node *v* is called a ***left subtree*** or ***right subtree***, respectively, of *v*.
- A binary tree is ***proper (full)*** if each node has either zero or two children.
  - Every internal node has exactly two children.
- A binary tree that is not proper is ***improper***
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

PROPERTIES OF PROPER BINARY TREES

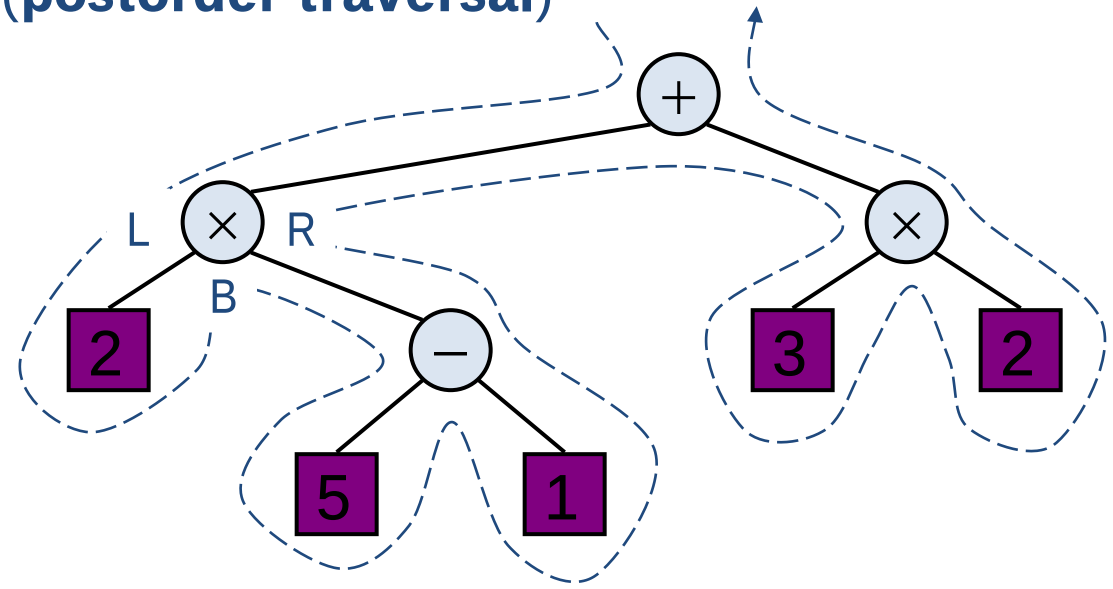
- Notation
  - n number of nodes
  - e number of externa l nodes
  - i number of internal nodes
  - h height

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



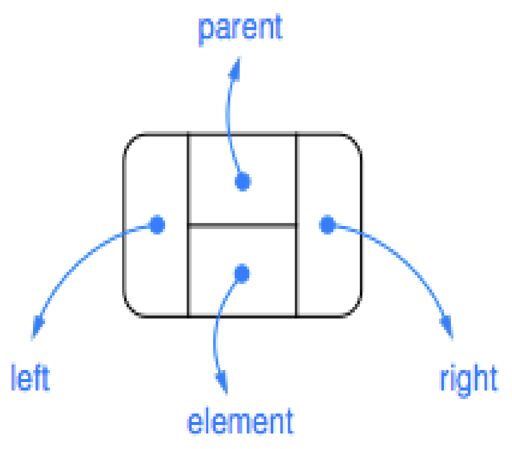
EULER TOUR TRAVERSAL

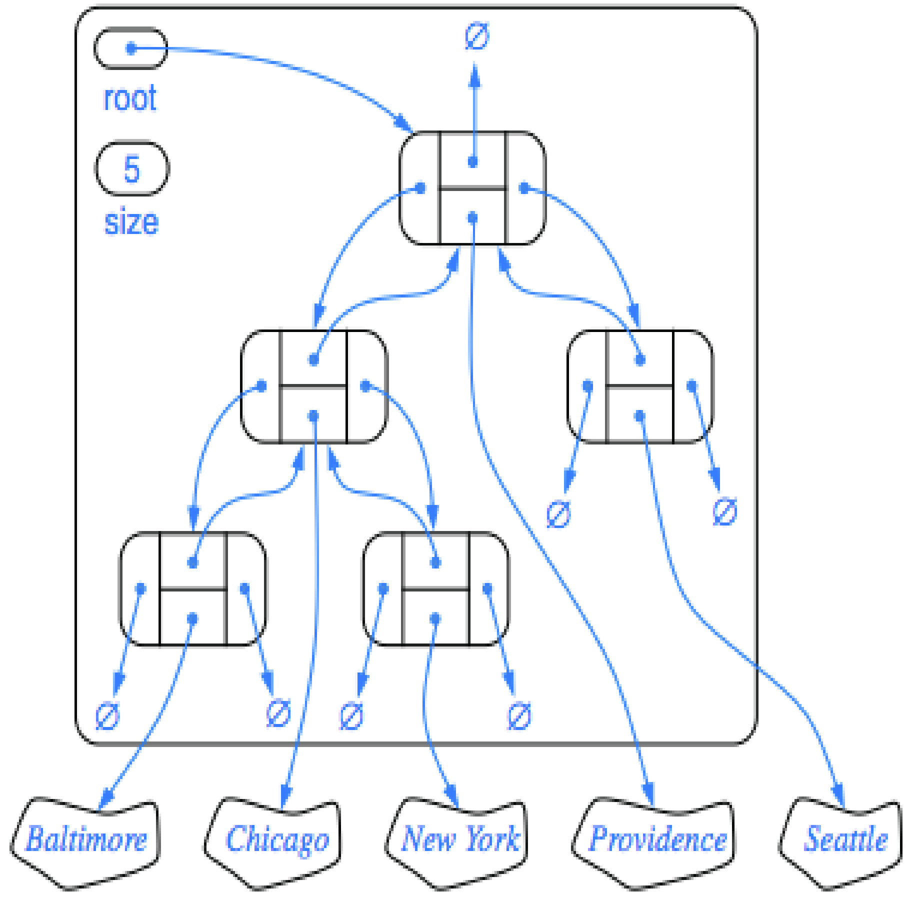
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (**preorder traversal**)
  - from below (**inorder traversal**)
  - on the right (**postorder traversal**)



LINKED STRUCTURE FOR BINARY TREES

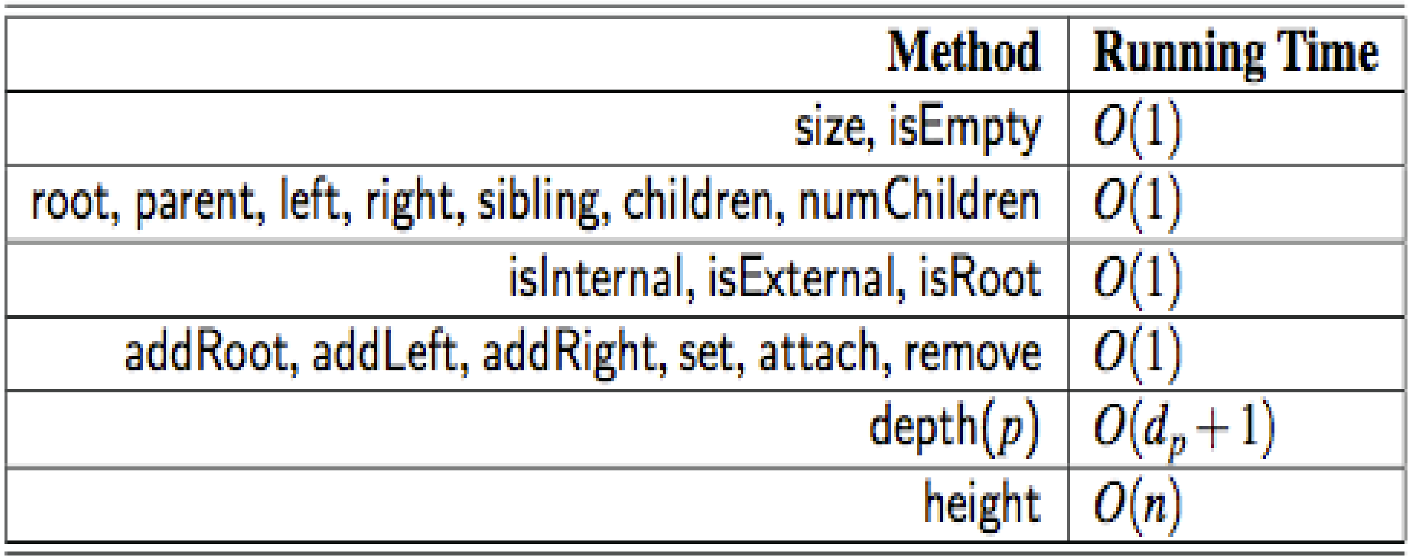
linked structure, with a node that maintains references to the element stored at a position *p* and to the nodes associated with the children and parent of *p*.





(a) (b)

PERFORMANCE OF THE LINKED BINARY TREE IMPLEMENTATION

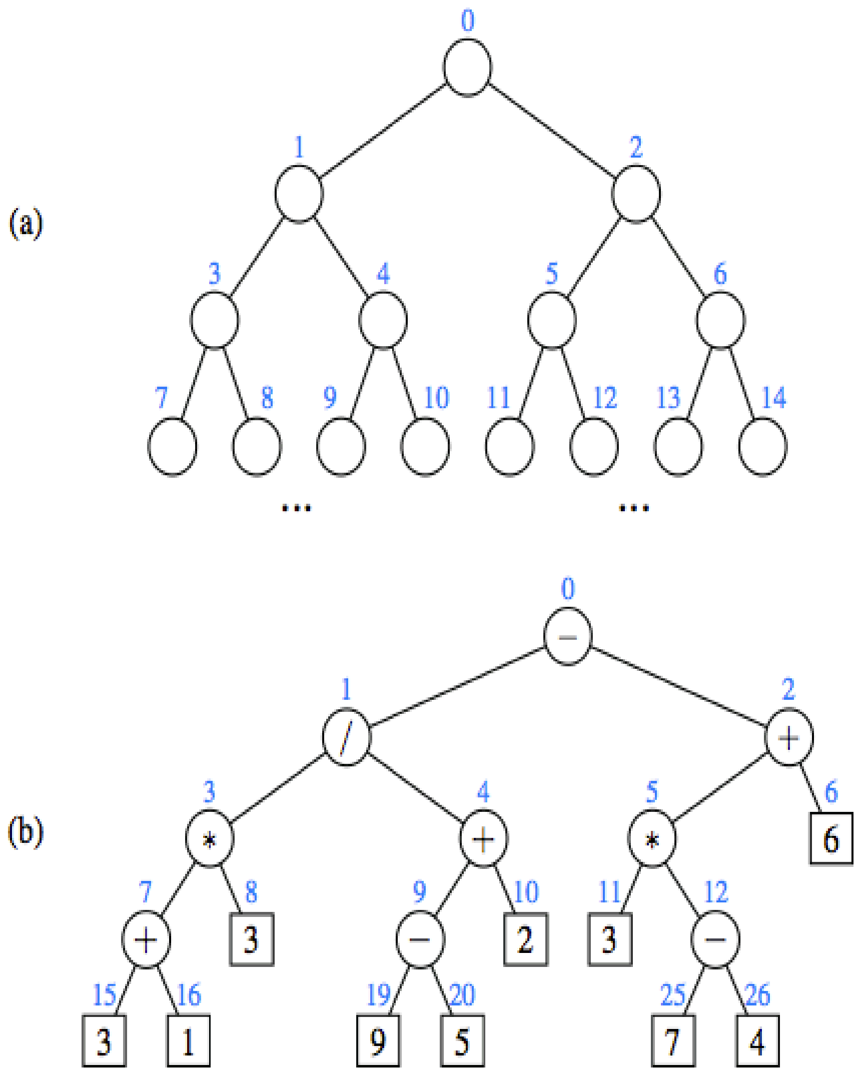


Method	Running Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p+1)$
height	$O(n)$

Running times for the methods of an *n*-node binary tree implemented with a linked structure. The space usage is $O(n)$. d_p is depth of node.

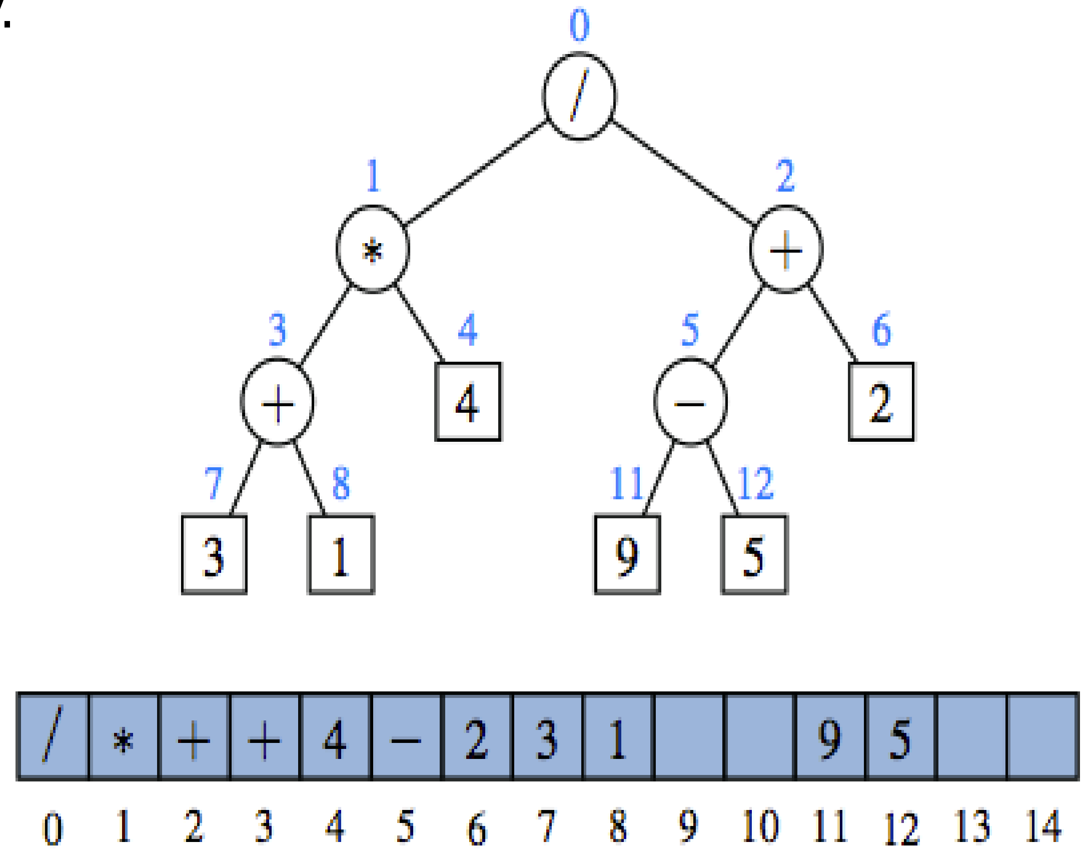
ARRAY-BASED REPRESENTATION OF A BINARY TREE

- Utilize the way of numbering the positions of T.
- For every position *p* of *T* , let *f(p)* be the of integer defined as follows.
  - If *p* is the root of *T*, then *f*(*p*)=0.
  - If *p* is the left child of position *q*, then *f*(*p*) = 2*f*(*q*)+1.
  - If *p* is the right child of position *q*, then *f*(*p*) = 2*f*(*q*)+2.
- *f* is known as ***level numbering*** of the positions in a binary tree *T* , for it numbers the positions on each level of *T* in increasing order from left to right.



ARRAY-BASED REPRESENTATION OF A BINARY TREE 2

an array-based structure *A*, with the element at position *p* of *T* stored at index *f*(*p*) of the array.

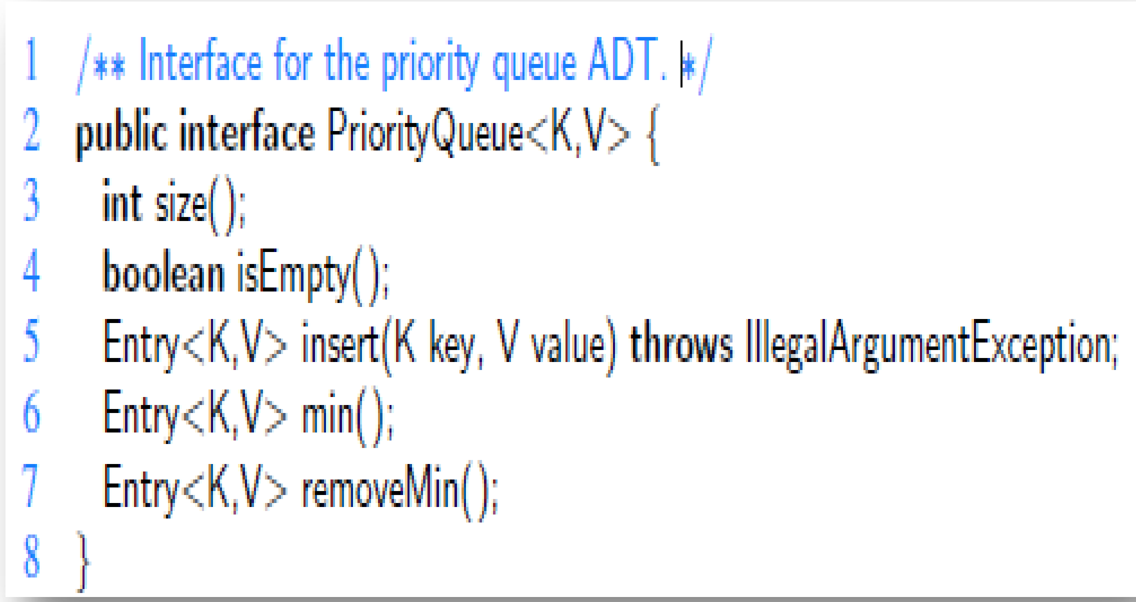


PRIORITY QUEUE

- Queue ADT is a collection of objects that are added and removed according to the ***first-in, first-out*** (***FIFO***) principle.
- However, sometimes a FIFO policy does not suffice.
  - Ex> "first come, first serve" policy might seem reasonable, but other priorities also come into play.
- A **priority queue** is a data structure for storing prioritized elements that allows arbitrary insertion, and allows the removal of the element that has **first priority** (*minimal* key).
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

PRIORITY QUEUE ADT

- A priority queue stores a collection of entries
- Each entry is a pair
  (key, value)
- Priority is stored in the key

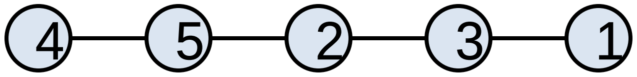


```
/** Interface for the priority queue ADT. */
public interface PriorityQueue<K,V> {
  int size();
  boolean isEmpty();
  Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
  Entry<K,V> min();
  Entry<K,V> removeMin();
}
```

- Main methods
  - insert(k, v): inserts an entry with key k and value v
  - removeMin(): removes and returns the entry with smallest key, or null if the the priority queue is empty
- Additional methods
  - min(): returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size(), isEmpty()

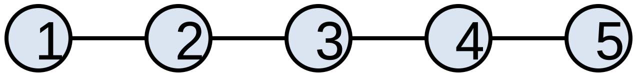
SEQUENCE-BASED PRIORITY QUEUE

- 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

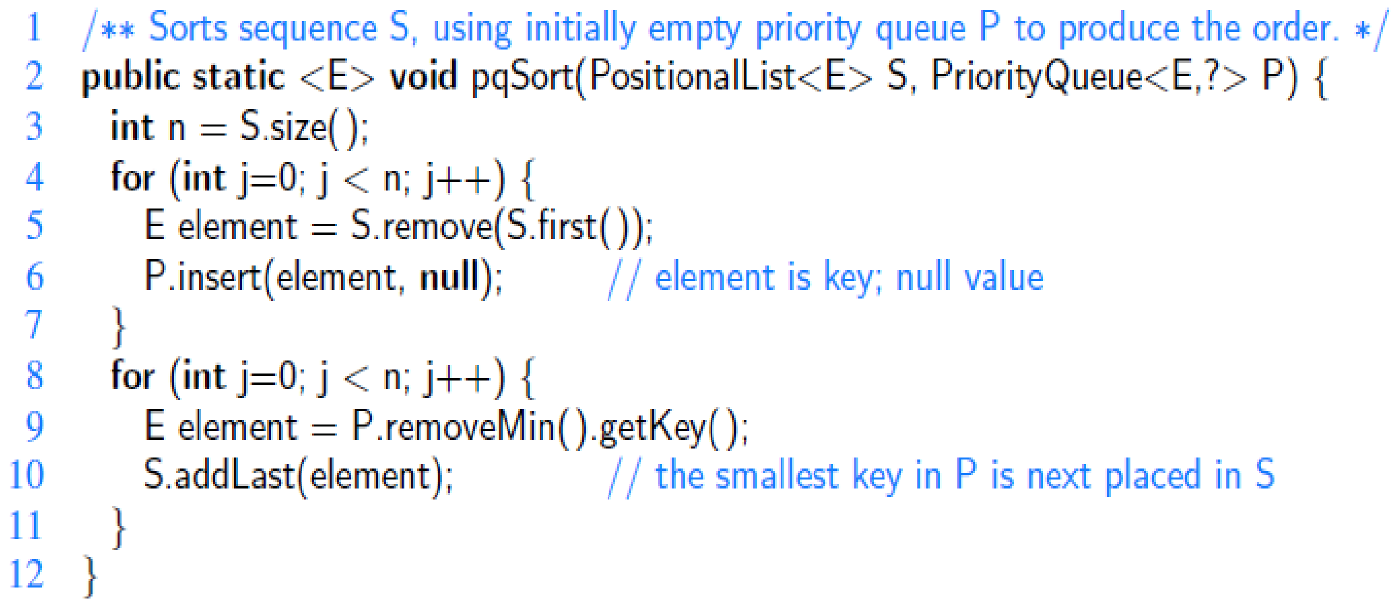
- Implementation with a sorted list



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

PRIORITY QUEUE SORTING “SCHEME”

- We can use a priority queue to sort a list of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation



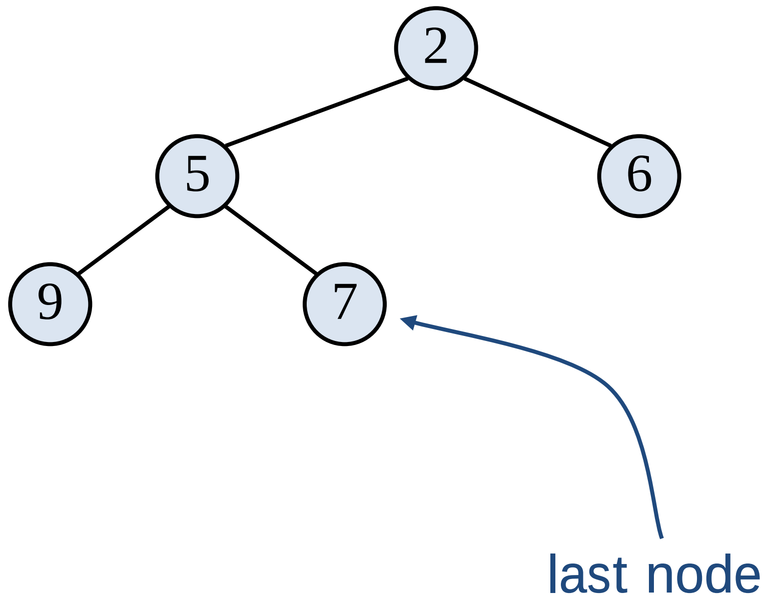
```
/** Sorts sequence S, using initially empty priority queue P to produce the order. */
public static <E> void pqSort(PositionalList<E> S, PriorityQueue<E,?> P) {
  int n = S.size();
  for (int j=0; j < n; j++) {
    E element = S.remove(S.first());
    P.insert(element, null);          // element is key; null value
  }
  for (int j=0; j < n; j++) {
    E element = P.removeMin().getKey();
    S.addLast(element);               // the smallest key in P is next placed in S
  }
}
```

- The [illegible] ns, including selection-sort, insertion-sort, and heap-sort

HEAPS

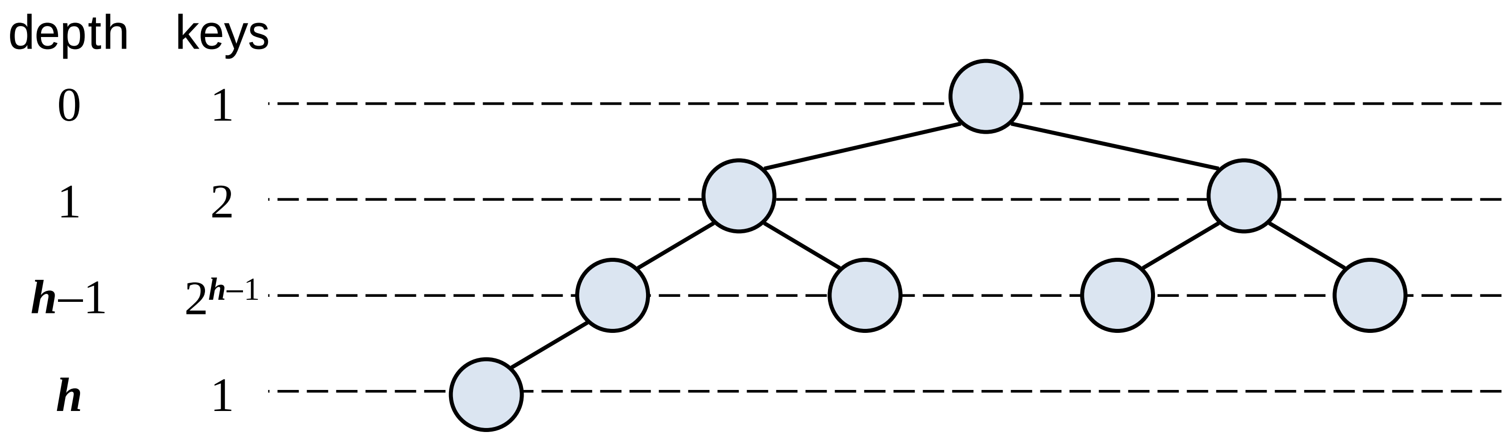
- A binary heap is a binary tree storing keys at its nodes and satisfying the following properties:
- (min) Heap-Order: for every internal node v other than the root, ***key*(*v*) ≥ *key*(*parent*(*v*))**
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \dots, h - 1$, there are 2^i nodes of depth i
  - at depth $h - 1$, the internal nodes are to the left of the external nodes

- The last node of a heap is the rightmost node of maximum depth



Height of a Heap

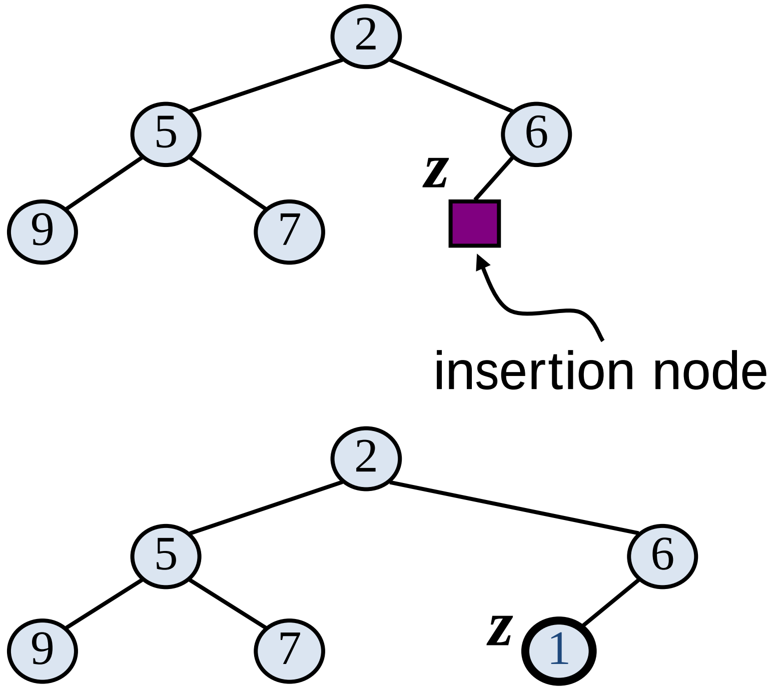
- Theorem: A heap T storing n entries has height $h = \lfloor \log n \rfloor$.
- Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \dots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$



INSERTION INTO A HEAP

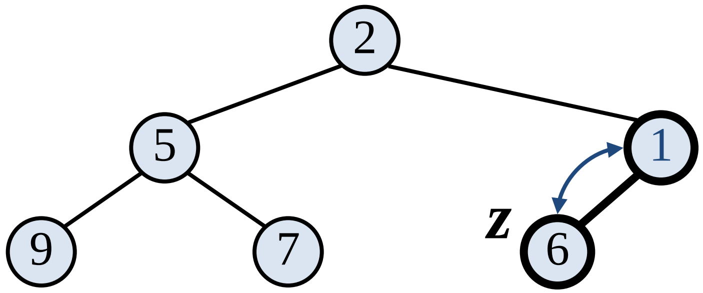
- Method *insert(k, v)* of the priority queue ADT corresponds to the insertion of a key ***k*** to the heap

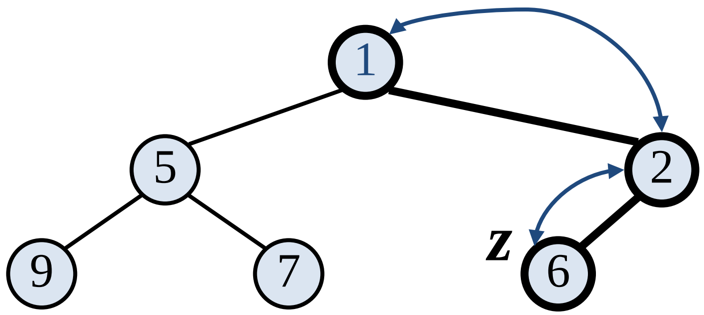
- The insertion algorithm consists of three steps to maintain the ***complete binary tree property***,
  - Find the insertion node ***z*** (the n ew last node)
  - Store ***k*** at ***z***
  - Restore the heap-order property



UPHEAP

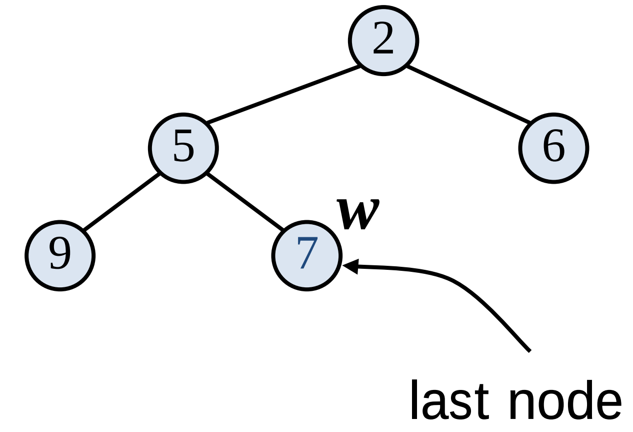
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k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

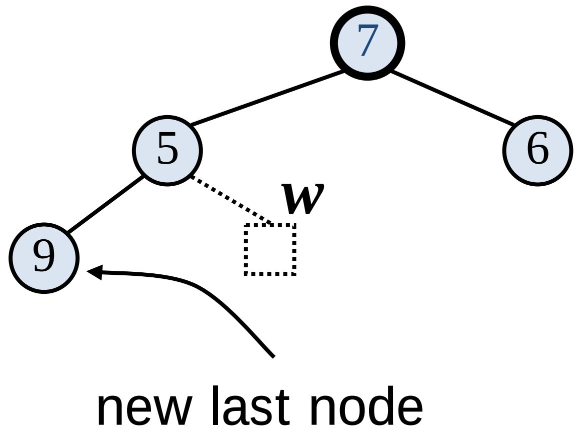




REMOVAL FROM A HEAP

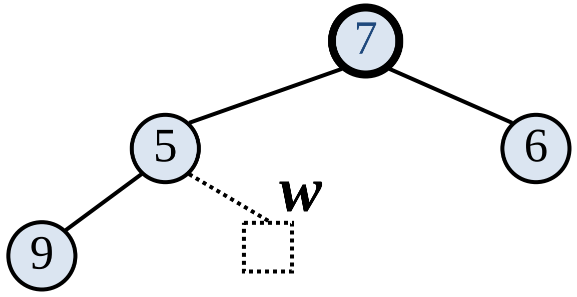
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)

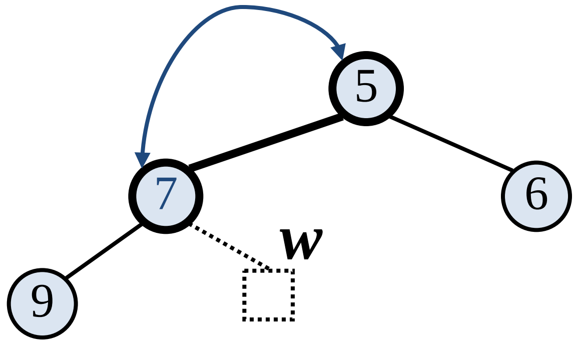




Downheap

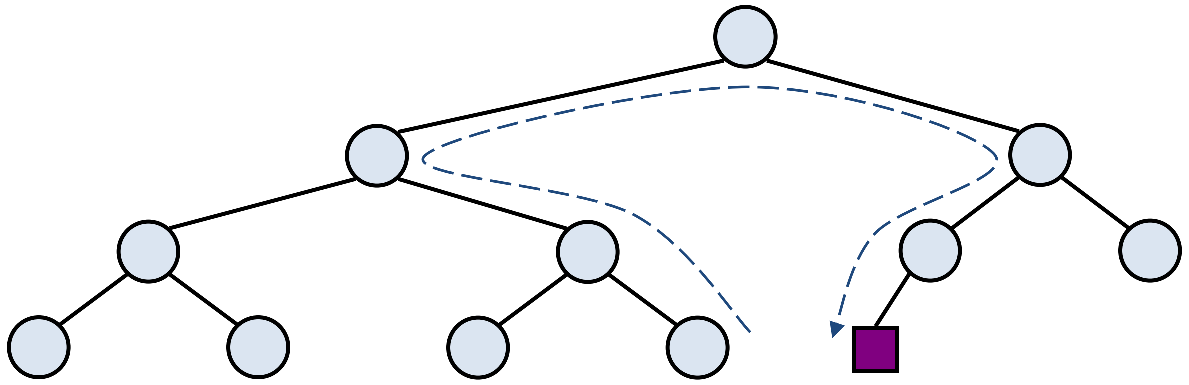
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Upheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time





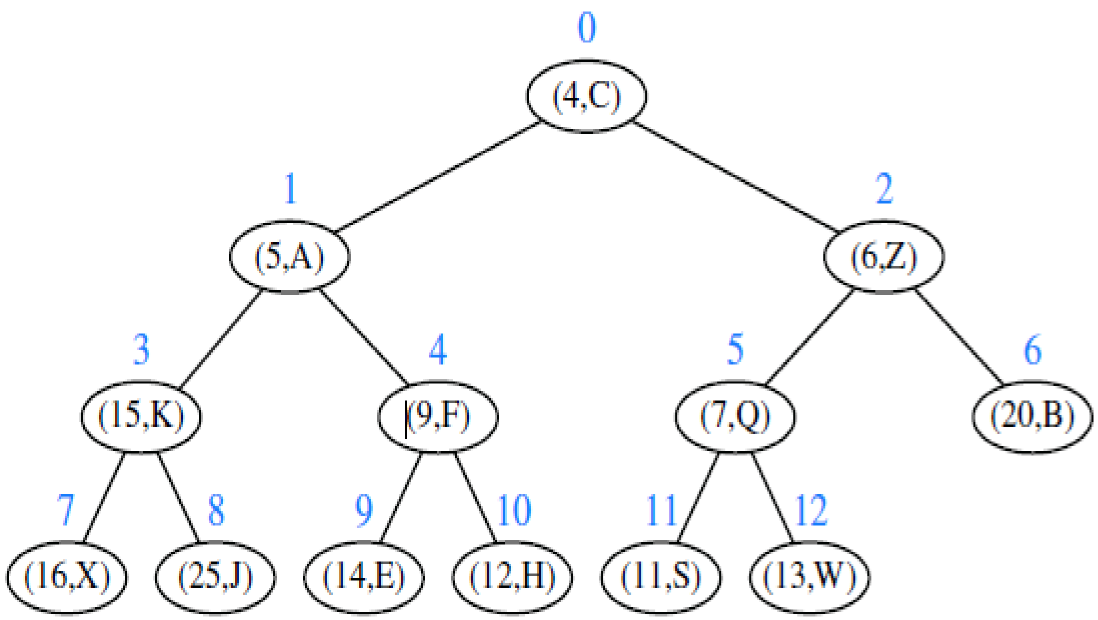
UPDATING THE LAST NODE

- The last node is the rightmost node at the bottom level of the tree, or as the leftmost position of a new level
- The last node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



ARRAY-BASED HEAP IMPLEMENTATION

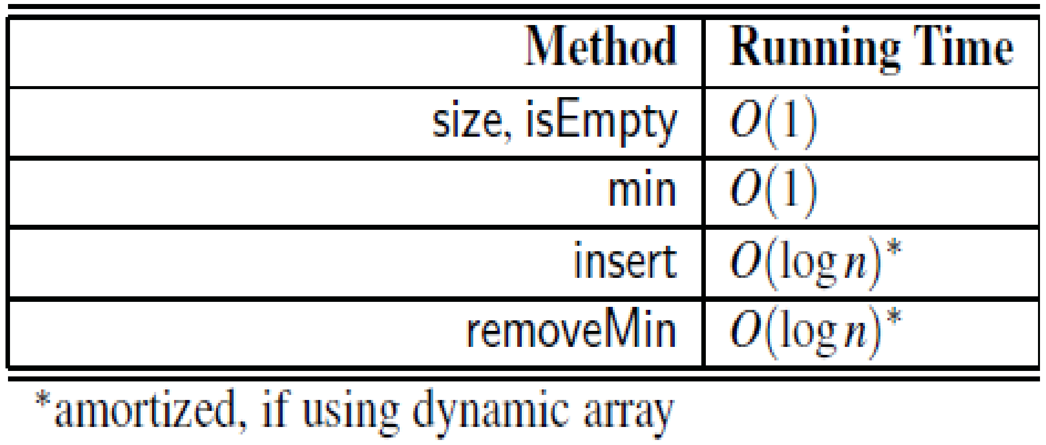
- We can represent a heap with *n* keys by means of an array of length *n*
- For the node at rank *i*
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Methods insert and removeMin depend on locating the last position of a h eap (in heap of size *n*, the last position at index $n-1$.)
  - **insert** corresponds to inserting at rank $n + 1$
  - **removeMin** corresponds to removing at rank *n*
- Space usage of an array-based representation of a complete binary tree with n nodes is O(n),
- Time bounds of methods for adding or removing elements become **amortized.** (occasional resizing of array needed)
- Yields in-place heap-sort



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

ANALYSIS OF A HEAP-BASED PRIORITY QUEUE

Assuming that two keys can be compared in *O*(1) time and that the heap *T* is implemented with an array-based or linked-based tree representation.



Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

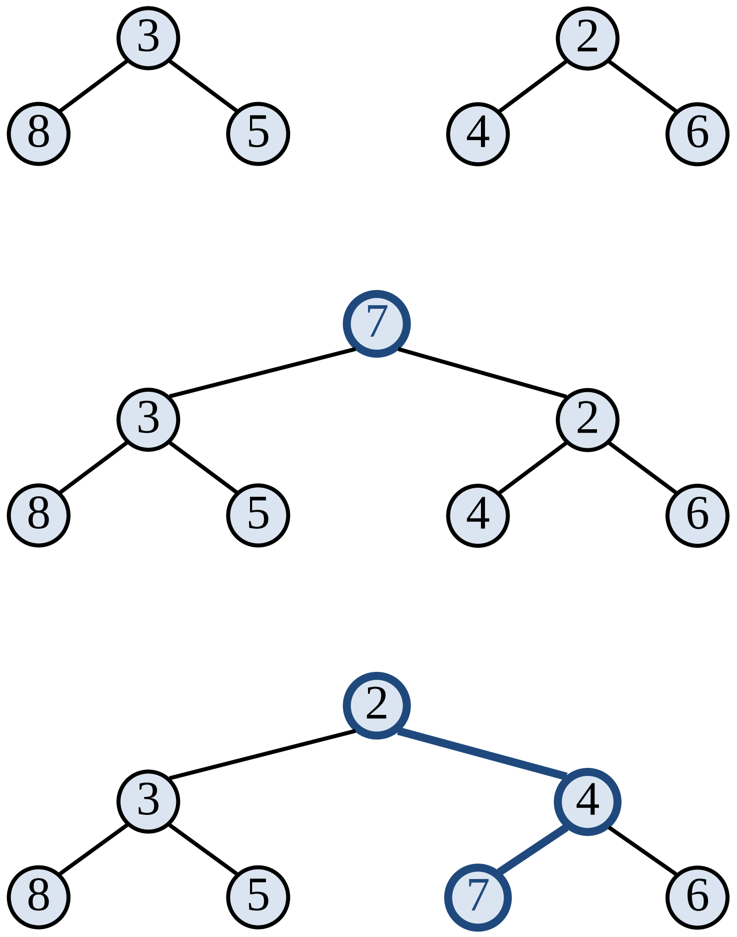
*amortized, if using dynamic array

BOTTOM-UP HEAP CONSTRUCTION

- If we start with an initially empty heap, n successive calls to the insert operation will run in $O(n\log n)$ time in the worst case.
- However, if all n key-value pairs to be stored in the heap are given in advance, such as during the first phase of the heap-sort algorithm, there is an alternative ***bottom-up*** construction method that runs in $O(n)$ time.
- we describe this bottom-up heap construction assuming the number of keys, n, is an integer such that $n = 2^{h+1} - 1$.
  - That is, the heap is a complete binary tree with every level being full, so the heap has height $h = \log(n+1) - 1$.

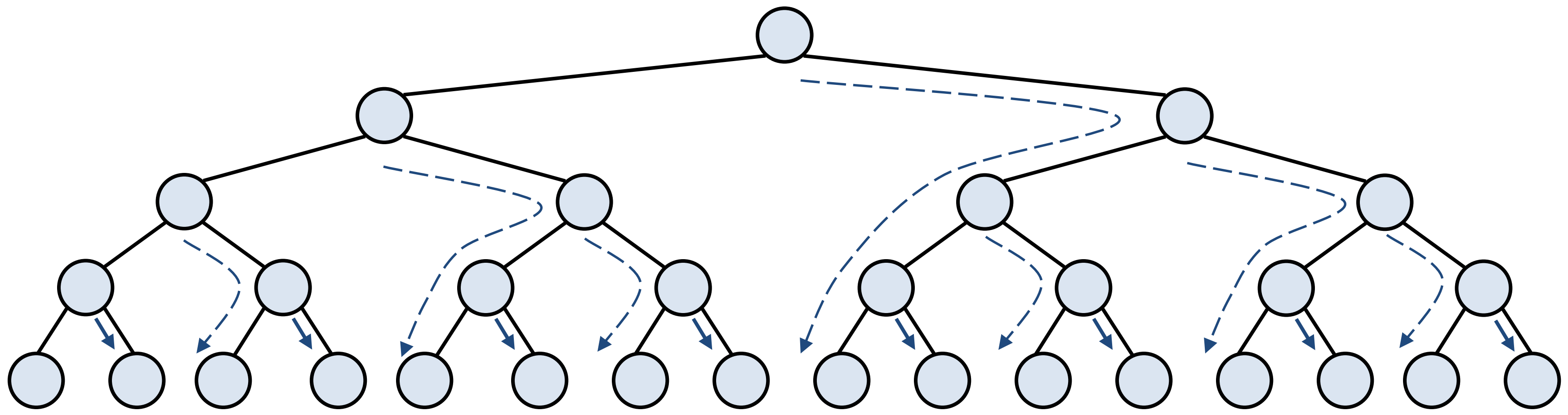
MERGING TWO HEAPS

- We are given two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



ANALYSIS

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort

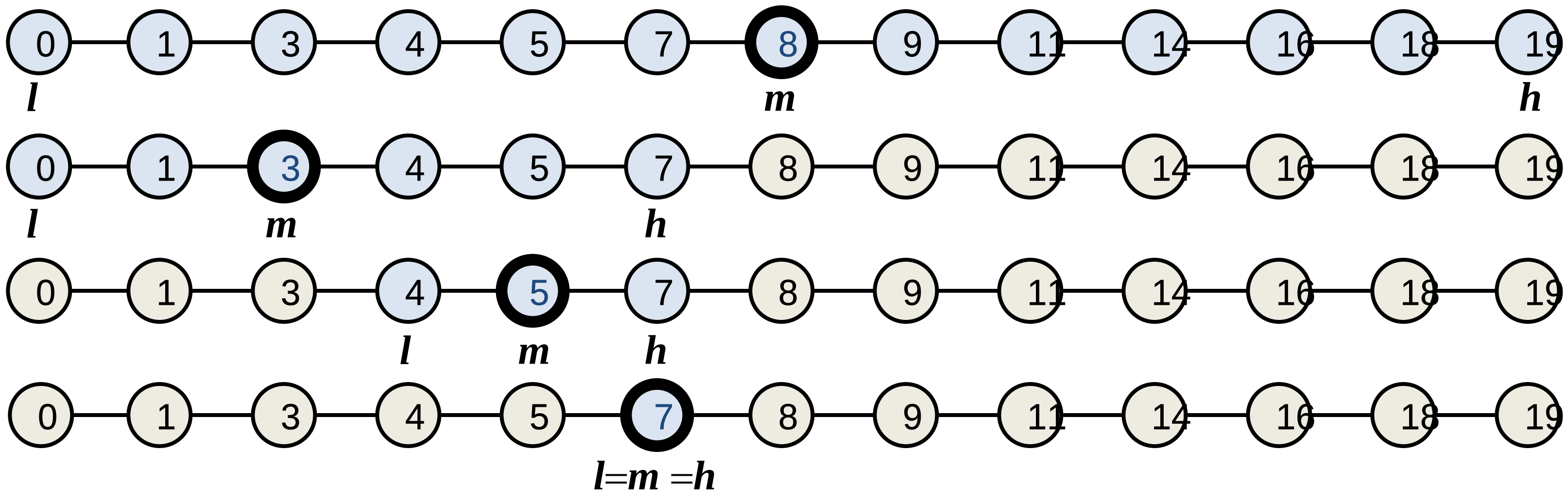


ORDERED MAPS

- Keys are assumed to come from a total order.
- Items are stored in order by their keys
- This allows us to support nearest neighbor queries :
  - Item with largest key less than or equal to k
  - Item with smallest key greater than or equal to k

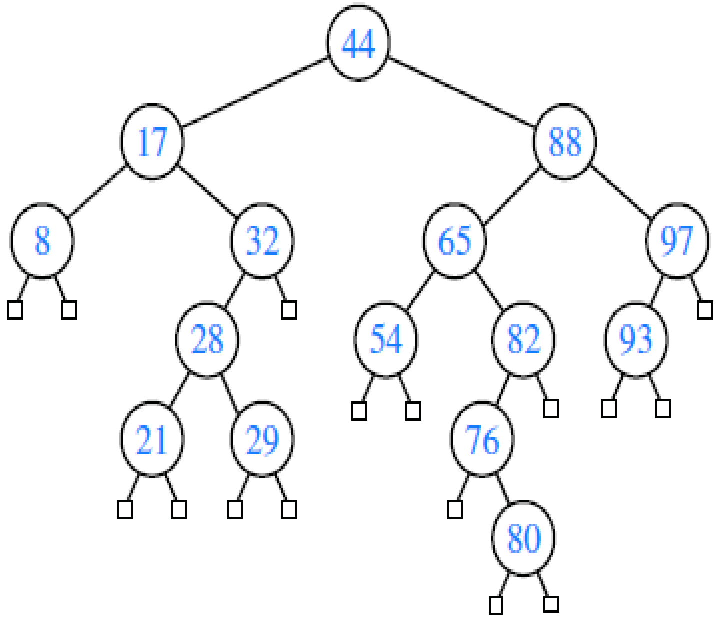
BINARY SEARCH

- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - similar to the high-low children's game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)



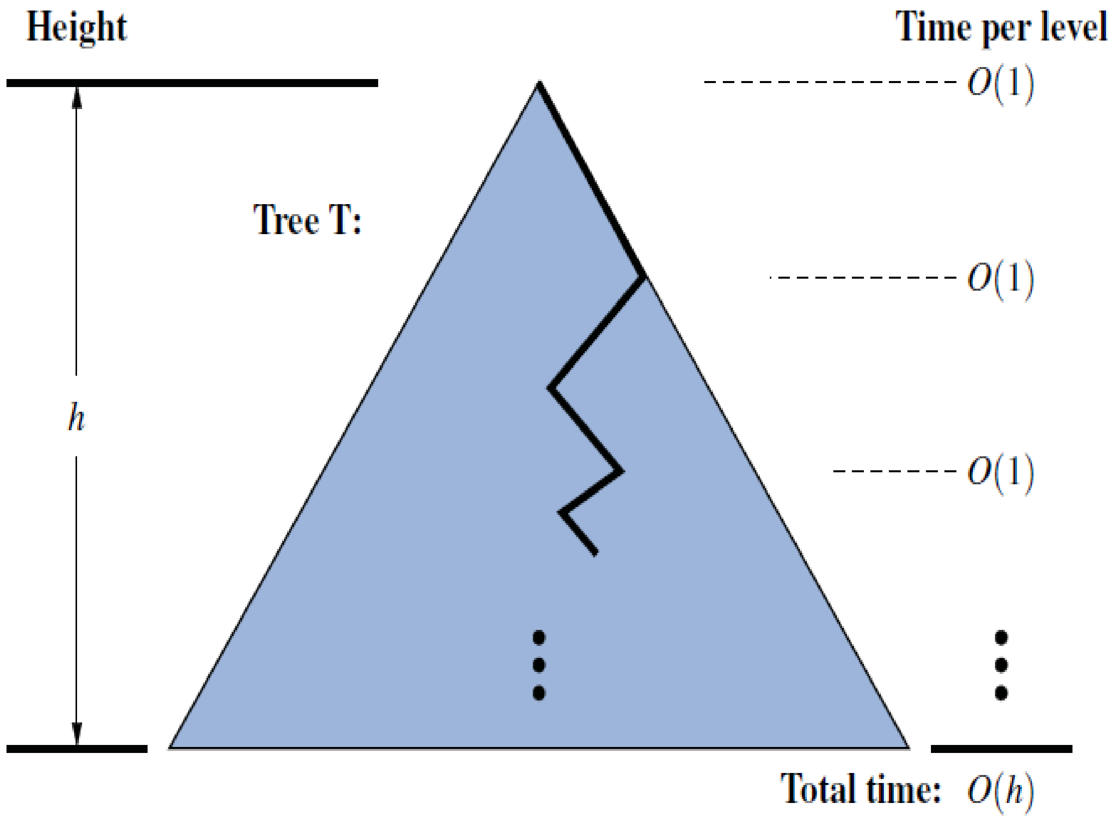
BINARY SEARCH TREES

- We define binary search tree as a proper binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items
  - We use the leaves as "placeholders" (sentinels)
  - Represented as **null** references in practice,
- An inorder traversal of a binary search trees visits the keys in increasing order



ANALYSIS OF BINARY TREE SEARCHING

- Algorithm TreeSearch is recursive and executes a constant number of primitive operations for each recursive call.



executes in time $O(h)$

We'll talk about various strategies to maintain an upper bound of $O(\log n)$ on the height soon

INSERTION

- To perform operation **put(k, o),** we search for key k (using TreeSearch)
- insertions, which always occur at a leaf).
- Assume a proper binary tree supports the following update operation
  - expandExternal(*p*, *e*): Stores entry e at the external position *p*, and expands *p* to be internal, having two new leaves as children.

```
Algorithm TreeInsert($k$, $v$):
    Input: A search key $k$ to be associated with value $v$
    $p$ = TreeSearch(root(), $k$)
    if $k$ == key($p$) then
        Change $p$'s value to ($v$)
    else
        expandExternal($p$, ($k$, $v$))
```

executes in time $O(h)$

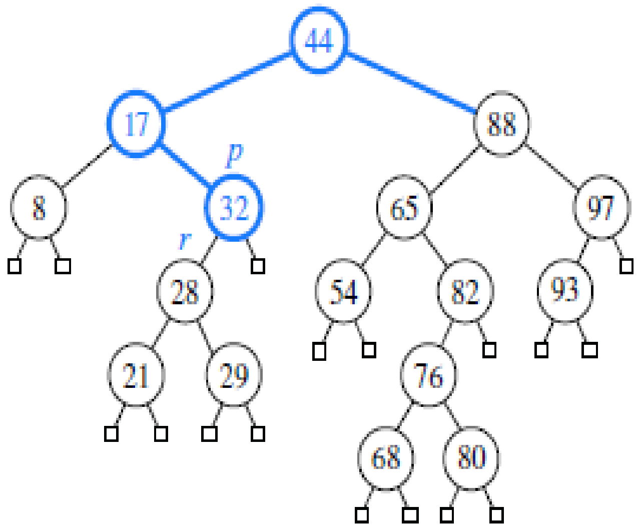
DELETION

- Deleting an entry from a binary search tree might happen anywhere in the tree
- To perform operation remove(k), we search for key k by calling TreeSearch(root( ), *k*) to find the position *p* storing an entry with key equal to *k* (if any).
  - If search returns an external node, then there is no entry to remove.
  - Otherwise,
    - at most one of the children of position *p* is internal,
    - Or position *p* has two internal children

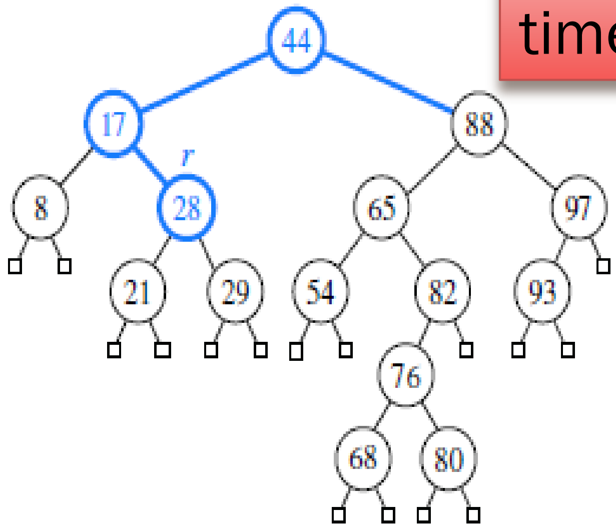
DELETION CONT.

- Deletion when at most one of the children of position *p* is internal.
  - Let position *r* be a child of *p* that is internal (or an arbitrary child, if both are leaves).
  - Remove *p* and the leaf that is *r*'s sibling, while promoting *r* upward to take the place of *p*.

executes in time $O(h)$



before the deletion of 32

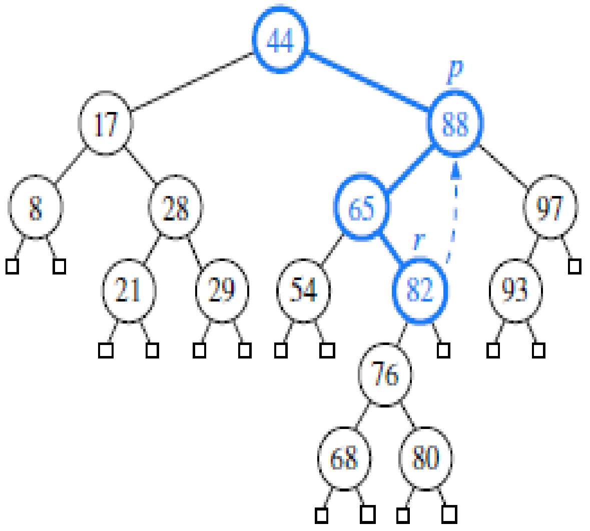


after the deletion of 32

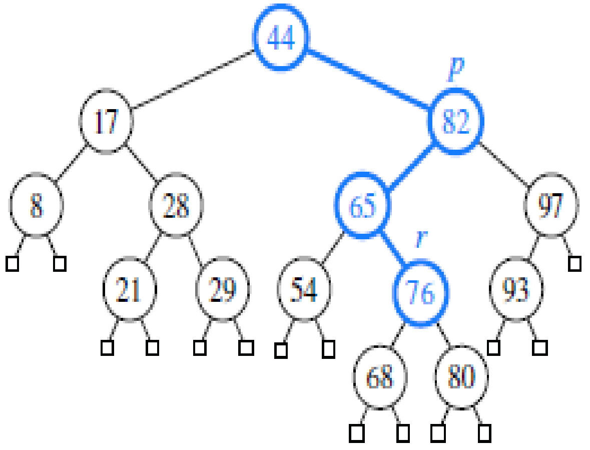
DELETION CONT.

- Deletion position p has two internal children
  - Locate position *r* containing the entry having the greatest key that is strictly less than that of position *p* (the rightmost internal position of the left subtree of position *p*)
  - Use *r*'s entry as a replacement for the one being deleted at position *p*.
  - Delete the node at position *r* from the tree.

executes in time $O(h)$

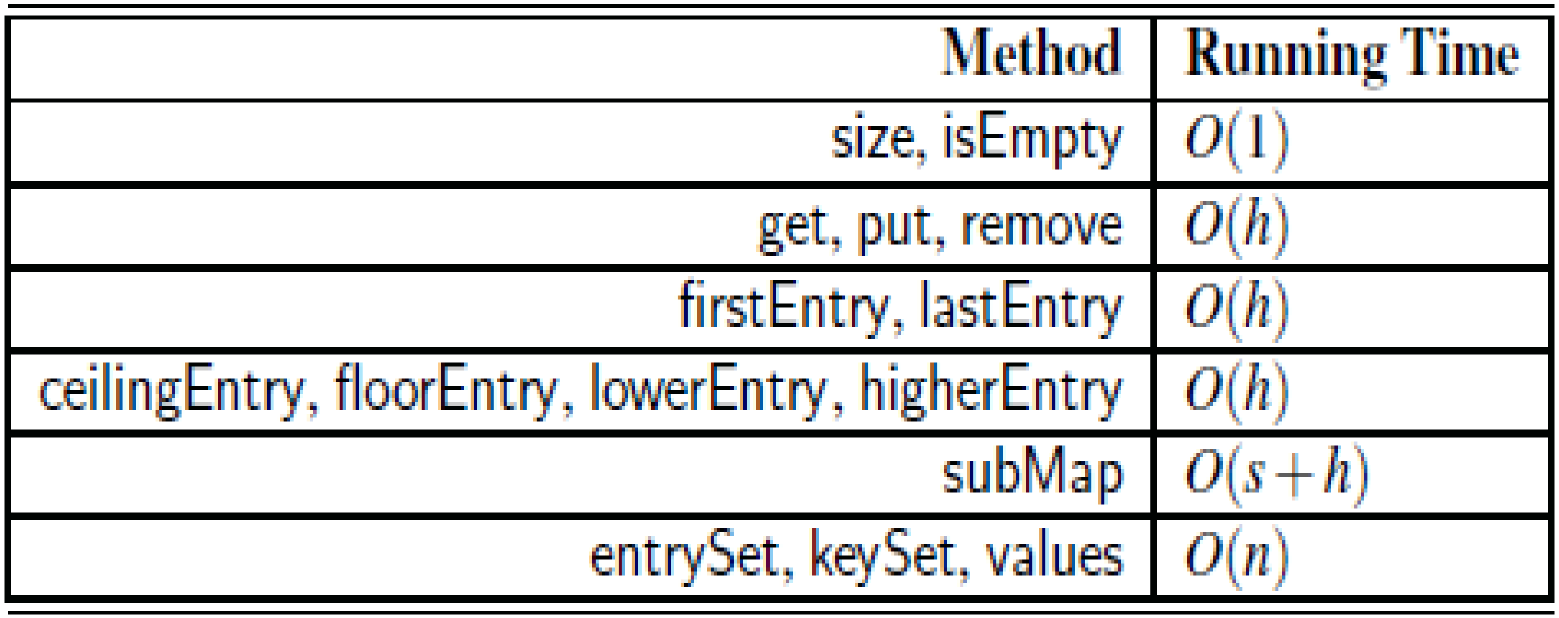


Before deleting 88



After deleting 88

PERFORMANCE OF A BINARY SEARCH TREE



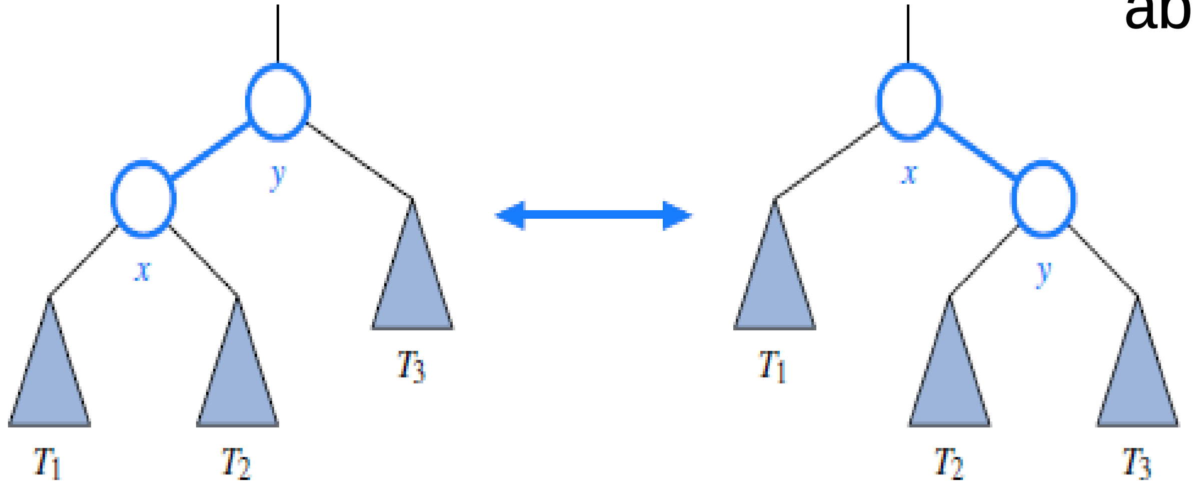
Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(h)$
firstEntry, lastEntry	$O(h)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(h)$
subMap	$O(s+h)$
entrySet, keySet, values	$O(n)$

* subMap implementation can be shown to run in $O(s+h)$ worst-case bound for a call that reports s results

BALANCED SEARCH TREES

- Augmenting a standard binary search tree with occasional operations to reshape the tree and reduce its height
  - Examples> AVL trees, splay trees, and red-black trees
- The primary operation to rebalance a binary search tree is known as a *rotation*
  - allows the shape of a tree to be modified while maintaining the search-tree property.

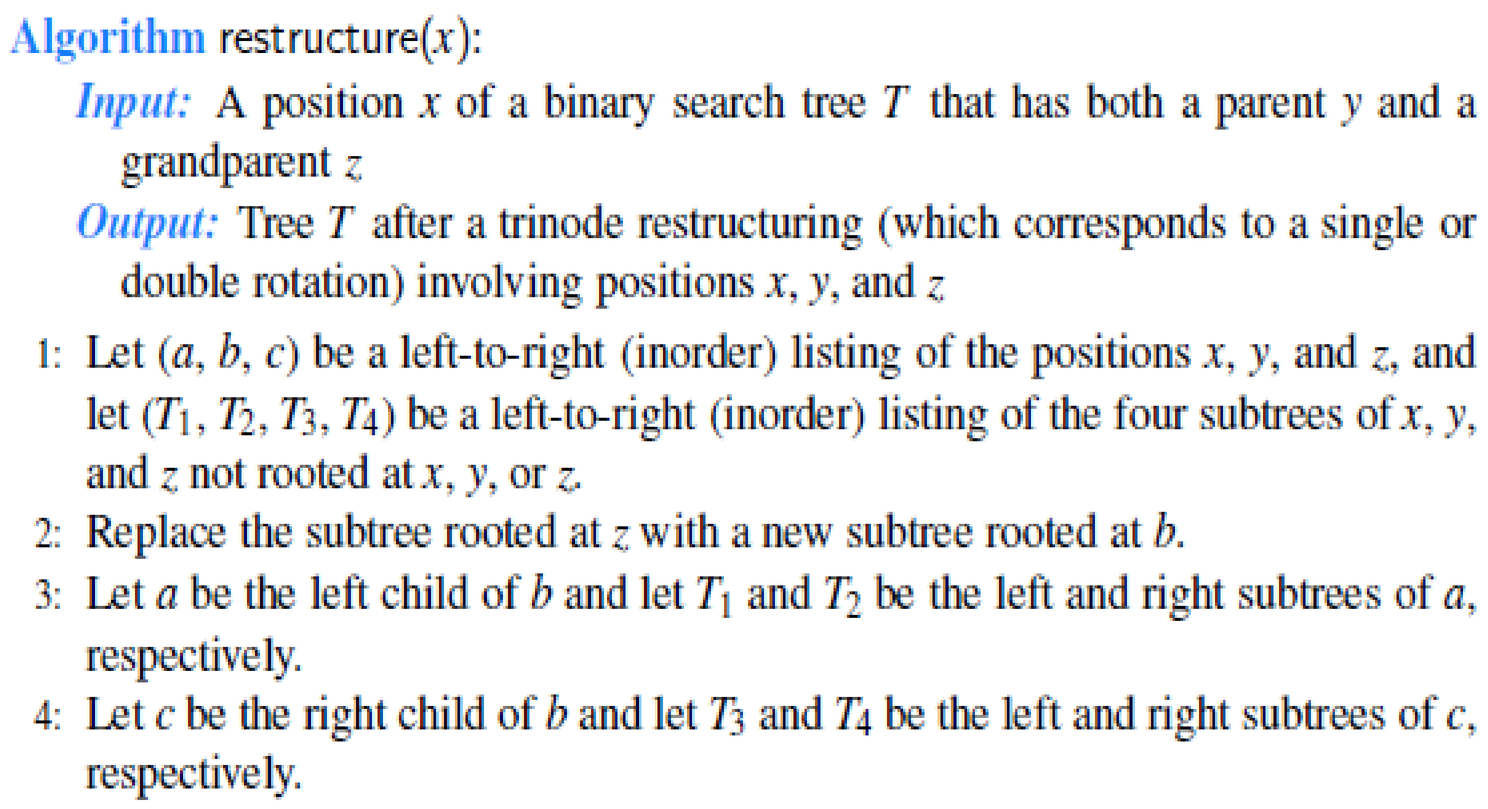
"rotate" a child to be above its parent



$O(1)$ time with a linked binary tree representation

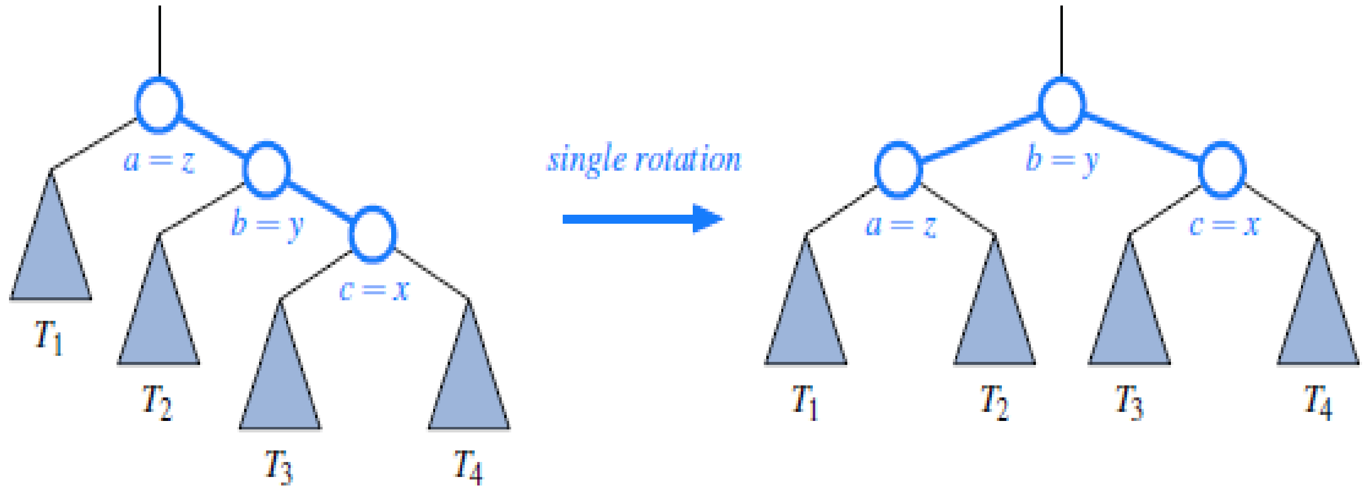
TRINODE RESTRUCTURING.

- *Trinode restructuring* is a compound rotation operations with the goal to restructure the subtree rooted at *the grandparent z* in order to reduce the overall path length to current node *x* and its subtrees.

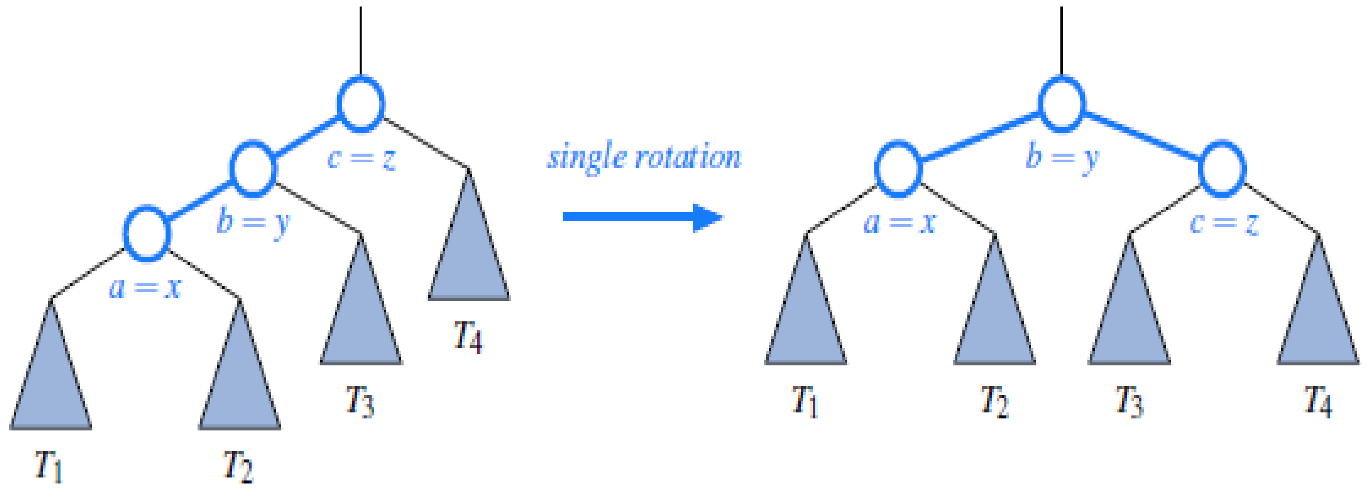


```
Algorithm restructure(x):
    Input: A position $x$ of a binary search tree $T$ that has both a parent $y$ and a
      grandparent $z$
    Output: Tree $T$ after a trinode restructuring (which corresponds to a single or
      double rotation) involving positions $x$, $y$, and $z$
  1: Let $(a, b, c)$ be a left-to-right (inorder) listing of the positions $x$, $y$, and $z$, and
     let $(T_1, T_2, T_3, T_4)$ be a left-to-right (inorder) listing of the four subtrees of $x$, $y$,
     and $z$ not rooted at $x$, $y$, or $z$.
  2: Replace the subtree rooted at $z$ with a new subtree rooted at $b$.
  3: Let $a$ be the left child of $b$ and let $T_1$ and $T_2$ be the left and right subtrees of $a$,
     respectively.
  4: Let $c$ be the right child of $b$ and let $T_3$ and $T_4$ be the left and right subtrees of $c$,
     respectively.
```

EXAMPLE OF A TRINODE RESTRUCTURING OPERATION 1

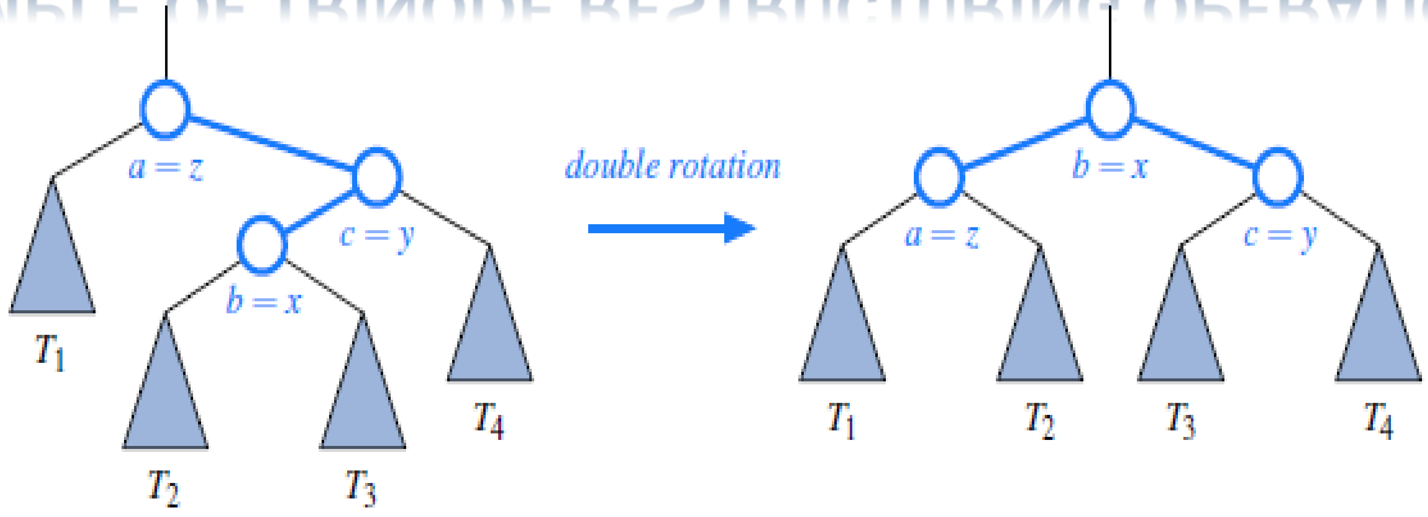


(a)

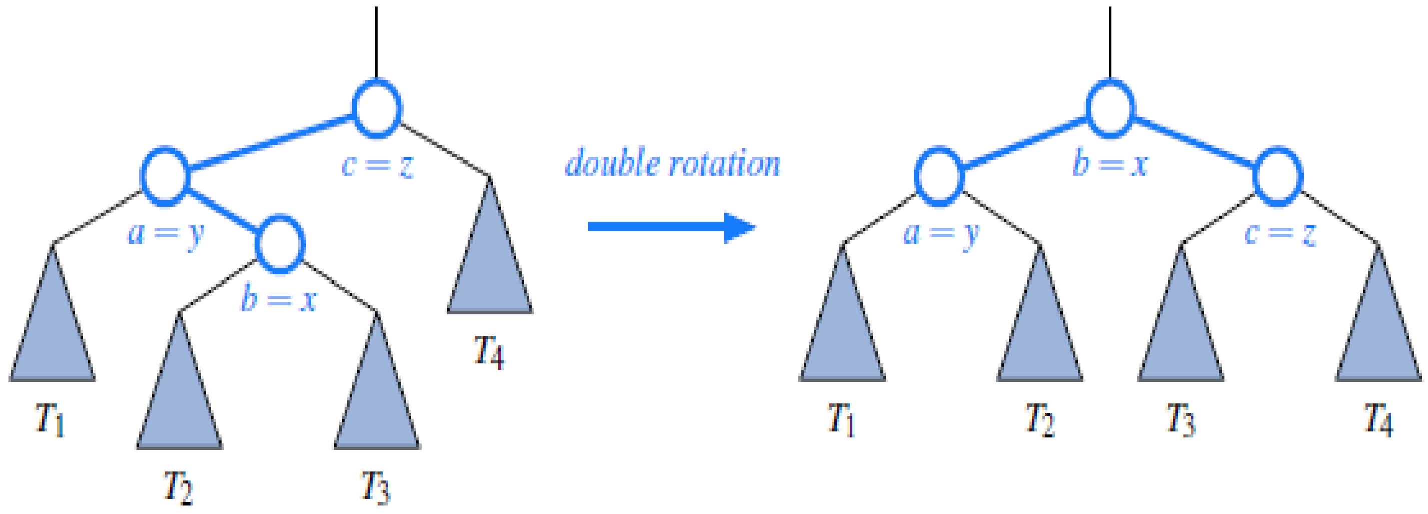


(b)

EXAMPLE OF TRINODE RESTRUCTURING OPERATION 2



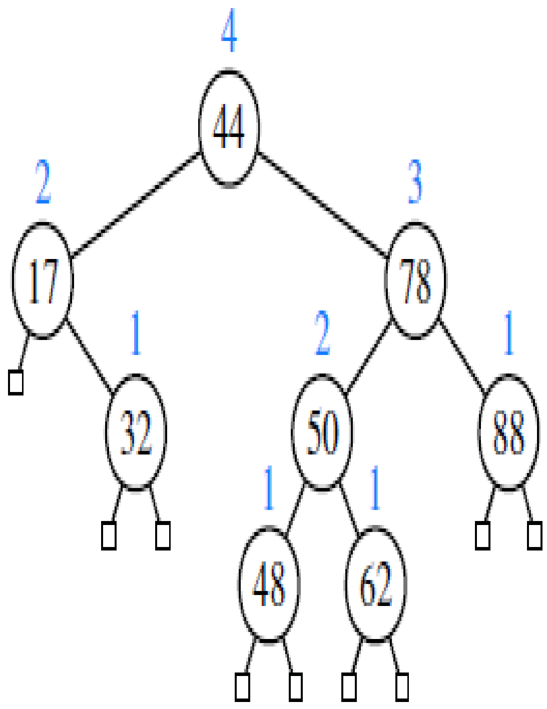
(c)



(d)

DEFINITION OF AN AVL TREE

- Any binary search tree *T* that satisfies the height-balance property is said to be an ***AVL tree***, named after the initials of its inventors: Adel'son-Vel'skii and Landis.
- ***Height-Balance Property***: For every internal position *p* of *T*, the heights of the children of *p* differ by at most 1.



PROPERTIES OF AVL TREE

- height-balance property allows
  - subtree of an AVL tree is itself an AVL tree.
  - The height of an AVL tree storing *n* entries is *O*(log*n*).

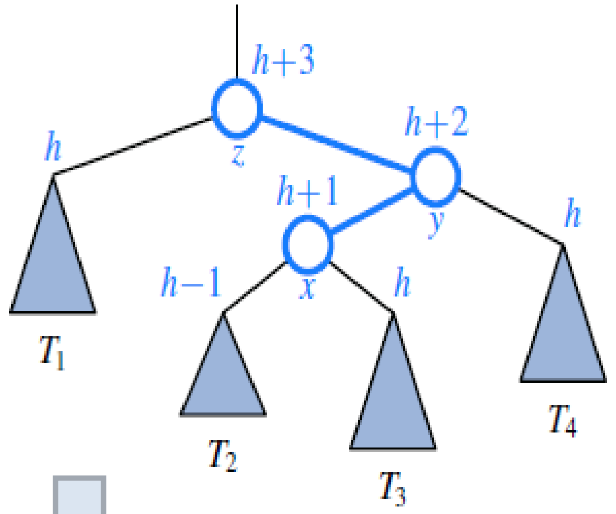
    (view 11.3 for the proof)
- height-balance property characterizing AVL trees is

  equivalent to saying that every position is balanced.
- Given a binary search tree *T*, we say that a position is ***balanced*** if the absolute value of the difference between the heights of its children is at most 1,
- ***AVL tree*** guarantees worst-case logarithmic running time for all the fundamental map operations

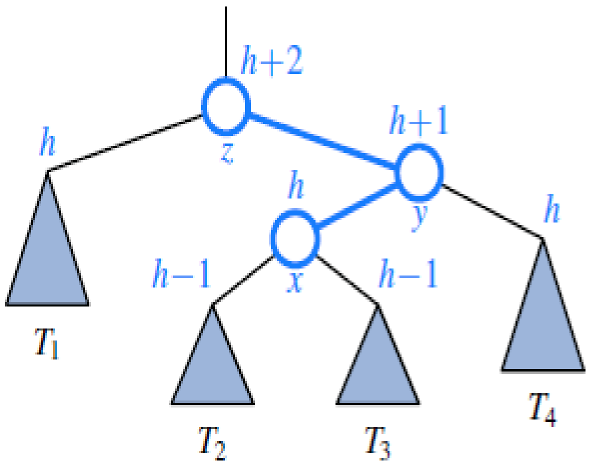
UPDATE OPERATIONS: INSERTION

- The insertion and deletion operations starts off with corresponding operations of (standard) binary search trees, but with post-processing for each operation to restore the balance
  - After insertion, the height-balance property may violated
  - Restructure *T* to fix any unbalance with a “search-and-repair” strategy.
- Any ancestor of *z* that became temporarily unbalanced becomes balanced again, and this one restructuring restores the height-balance property ***globally***.

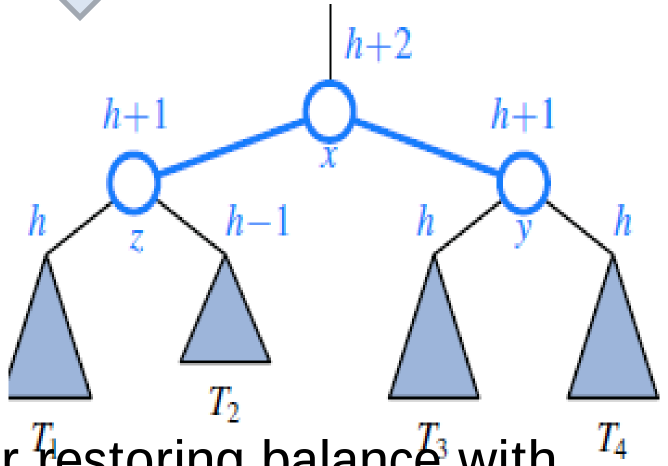
- Let *z* be the first position we encounter in going up from *p* toward the root of *T* such that *z* is unbalanced
- let *y* denote the child of *z* with greater height
- let *x* be the child of *y* with greater height (there cannot be a tie)
- Perform restructure(*x*)



after an insertion in subtree *T*3 causes imbalance at *z*



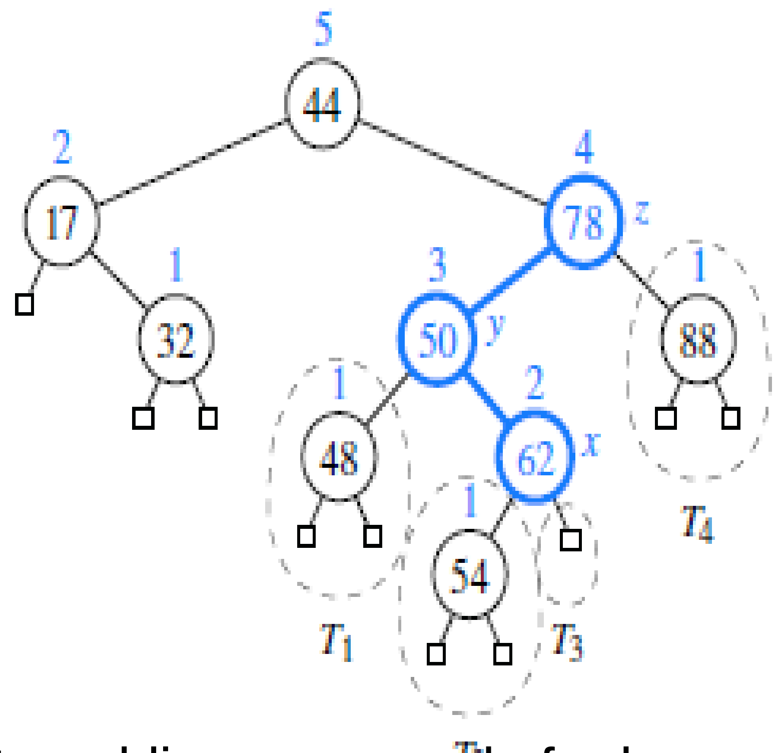
before the insertion



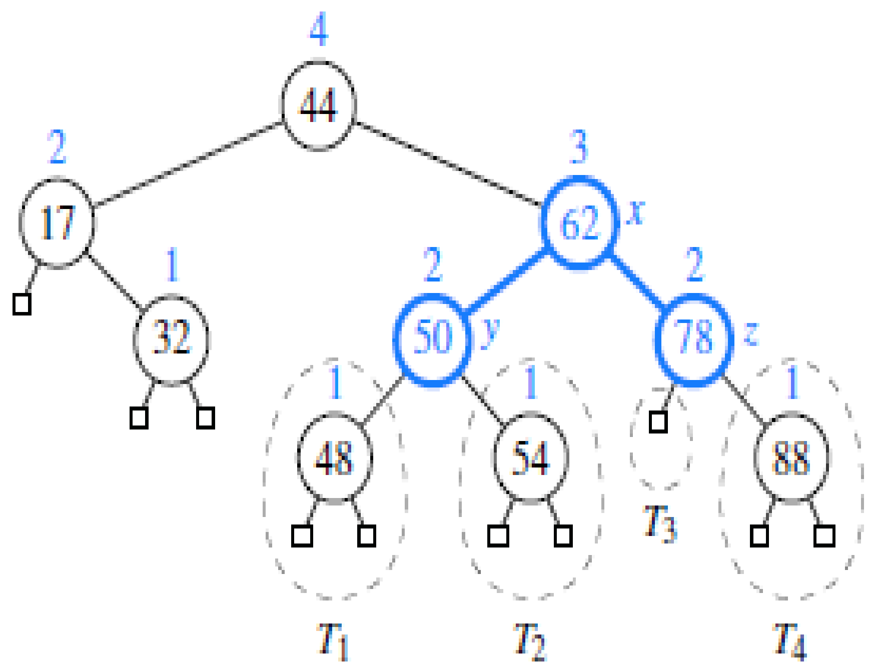
after restoring balance with trinode restructuring

EXAMPLE OF INSERT

insertion of an entry with key 54 in the AVL tree



after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced;



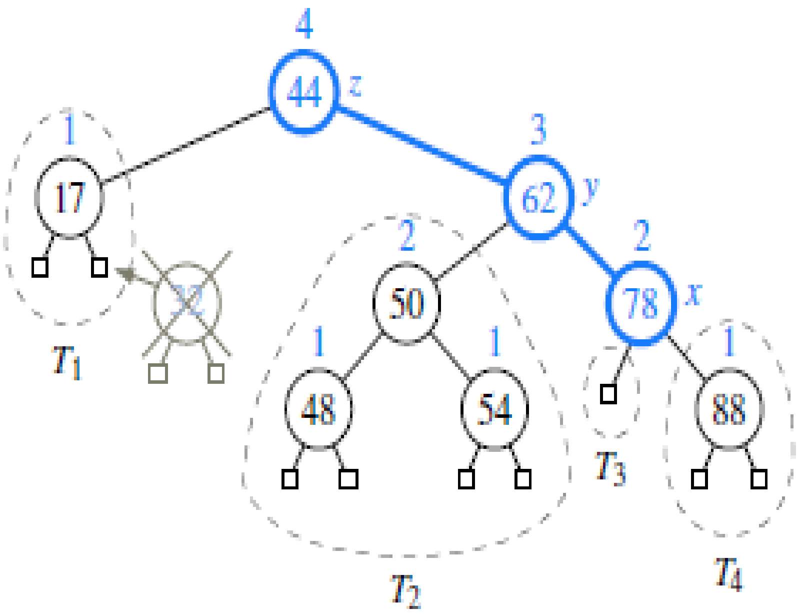
a trinode restructuring restores the height-balance property

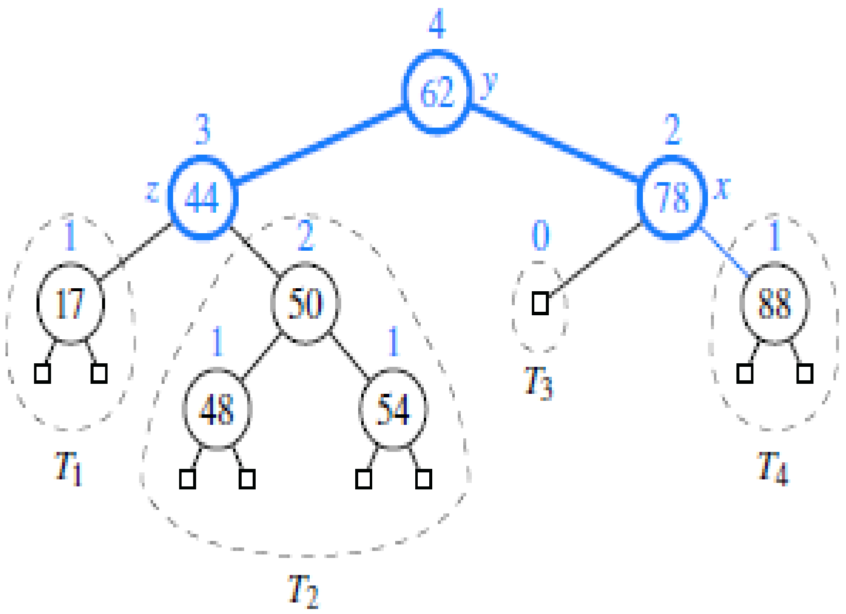
UPDATE OPERATIONS: DELETION

- As with insertion, we use trinode restructuring to restore balance in the tree *T* after deletion.
- let *z* be the first unbalanced position encountered going up from *p* toward the root of *T*,
- let *y* be that child of *z* with greater height
- let *x* be the child of *y* defined as follows:
  - if one of the children of *y* is taller than the other, let *x* be the taller child of *y*;
  - else (both children of *y* have the same height), let *x* be the child of *y* on the same side as *y*
- Run restructure(*x*) operation.
- After rebalancing *z*, we continue walking up *T* looking for unbalanced positions
  - The height-balance property is guaranteed to be ***locally*** restored within the subtree of *b* but not globally.

EXAMPLE

Deletion of the entry with key 32 from the AVL tree



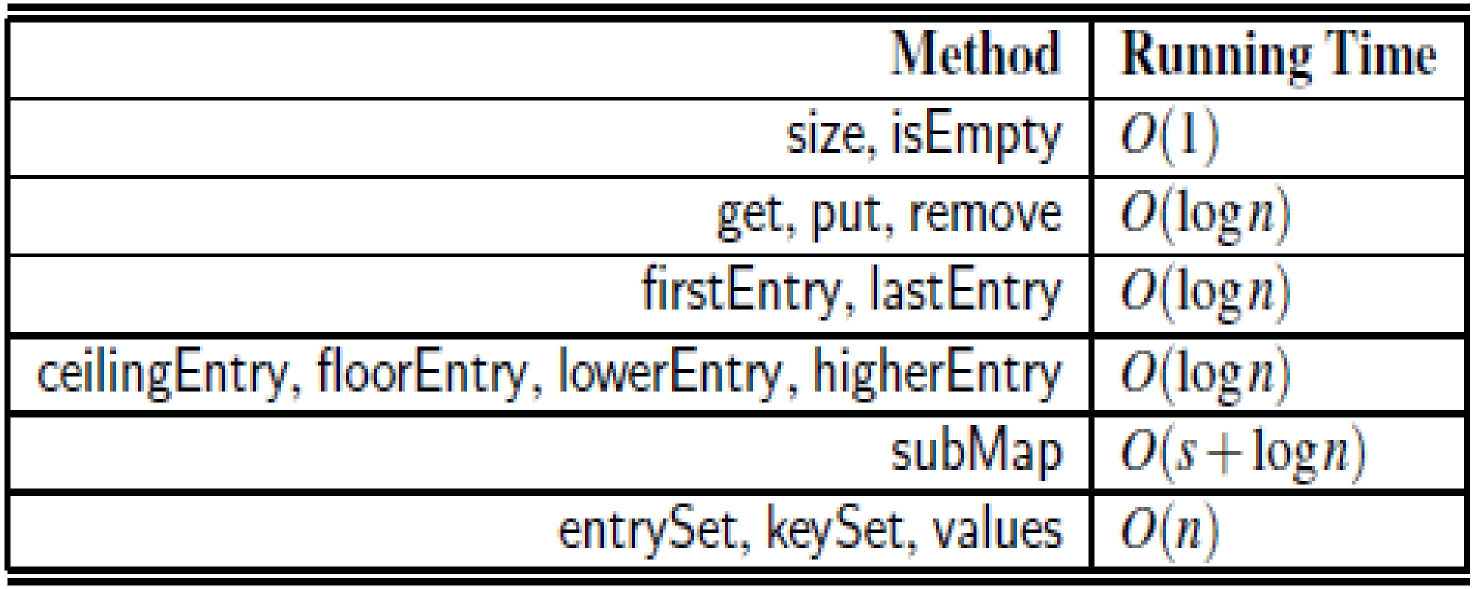


after removing the node storing key 32, the root becomes unbalanced

A trinode restructuring of *x*, *y*, and *z* restores the height-balance property.

PERFORMANCE OF AVL TREES

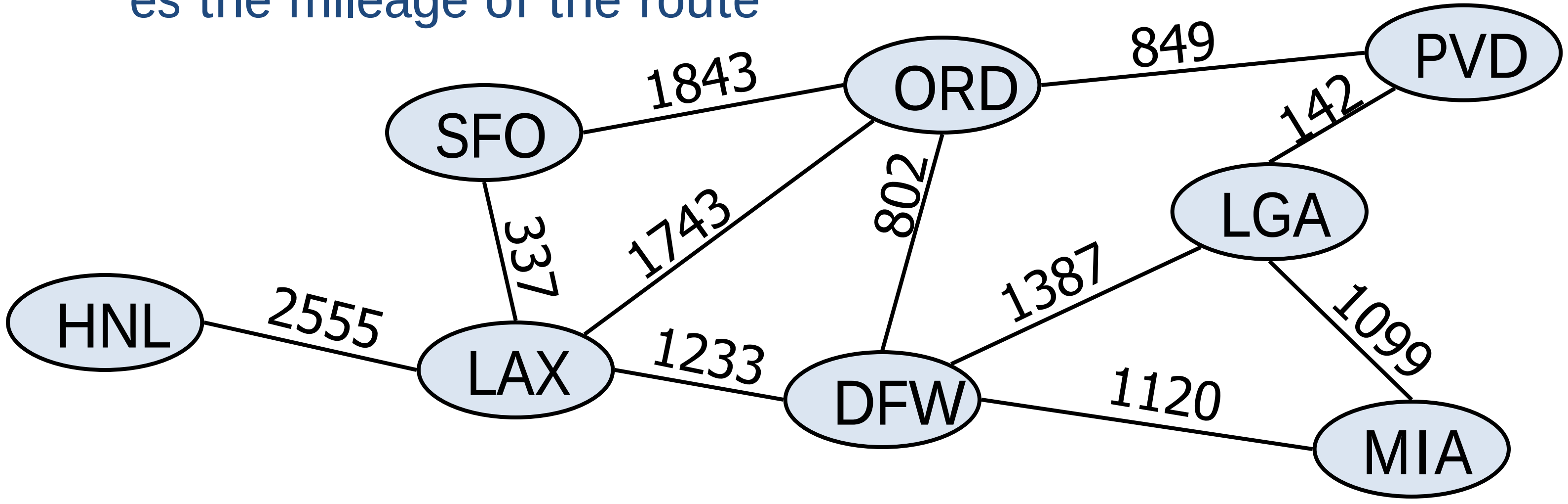
- the height of an AVL tree with *n* entries is guaranteed to be *O*(log*n*).



Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(\log n)$
firstEntry, lastEntry	$O(\log n)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s+\log n)$
entrySet, keySet, values	$O(n)$

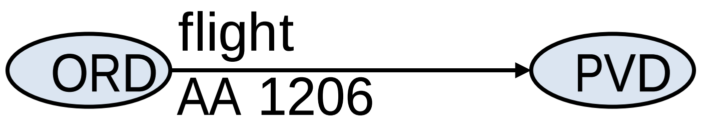
GRAPHS

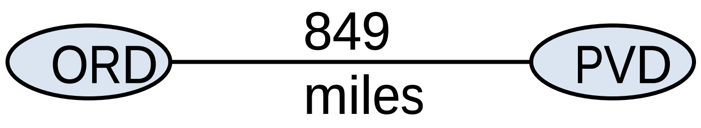
- A **graph** is a pair (V, E), where
  - V is a set of nodes, called **vertices** (aka nodes)
  - E is a collection of pairs of vertices, called **edges** (aka arcs)
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stor es the mileage of the route



EDGE TYPES

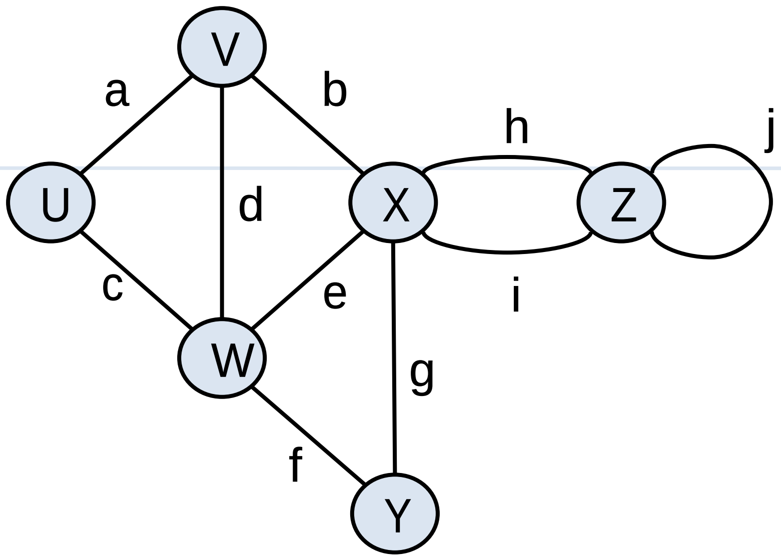
- **Directed edge**
  - ordered pair of vertices (*u*,*v*)
  - first vertex *u* is the origin
  - second vertex *v* is the destination
  - e.g., a flight
- **Undirected edge**
  - unordered pair of vertices (*u*,*v*)
  - e.g., a flight route
- **Directed graph**
  - all the edges are directed
  - e.g., route network
- **Undirected graph**
  - all the edges are undirected
  - e.g., flight network





- **Mixed graph :** graph that has both directed and undirected edges

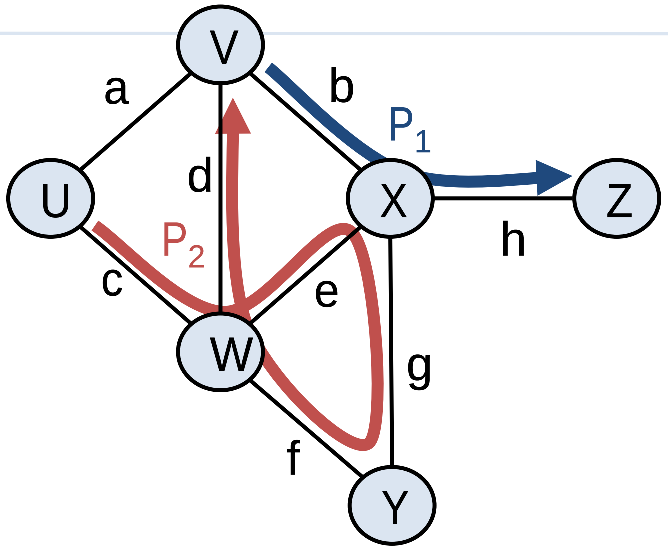
TERMINOLOGY



- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- **Degree** of a vertex
  - deg(X)= 5; X has degree 5
- Parallel edges (multiple edges)
  - h and i are parallel edges
  - Edges are collections (not sets)
- Self-loop
  - j is a self-loop

- ***outgoing edges*** of a vertex:
  - directed edges whose origin is that vertex.
- ***incoming edges*** of a vertex:
  - directed edges whose destination is that vertex.
- ***in-degree*** & ***out-degree*** of a vertex *v*
  - the number of the incoming and outgoing edges of *v*,
  - Denoted indeg(*v*) and outdeg(*v*)

TERMINOLOGY (CONT.)

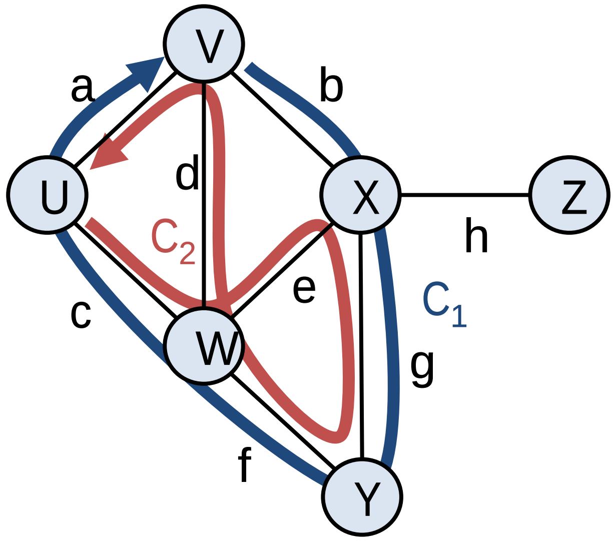


- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is **not** simple

- Graphs are said to be ***simple*** if they do not have parallel edges or self-loops
- Most graphs are simple; we will assume that a graph is simple unless otherwise specified

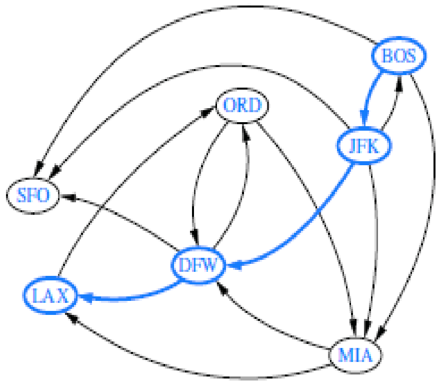
TERMINOLOGY (CONT.)

- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct, except for the first and the last
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is **not** simple

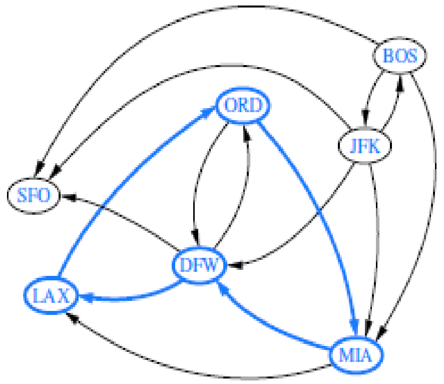


TERMINOLOGY (CONT.)

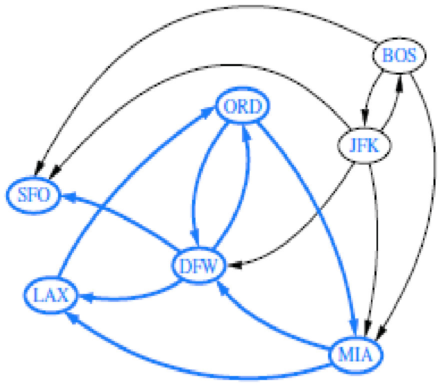
- Given vertices *u* and *v* of a (directed) graph *G*,
- *u* ***reaches*** *v*, and that *v* is ***reachable*** from *u*, if *G* has a (directed) path from *u* to *v*.
- ***reachability :***
  - undirected graph ***reachability*** is **symmetric**, that is to say, *u* reaches *v* if an only if *v* reaches *u*.
  - directed graph ***reachability*** is **asymmetric**, it is possible that *u* reaches *v* but *v* does not reach *u*,



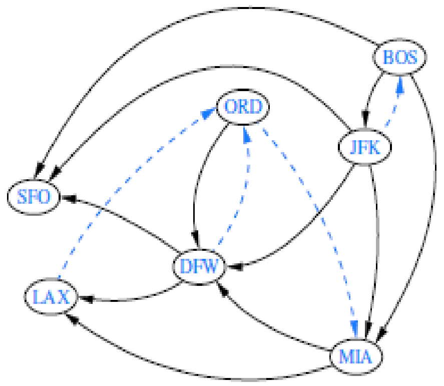
a directed path



strongly connected subgraph



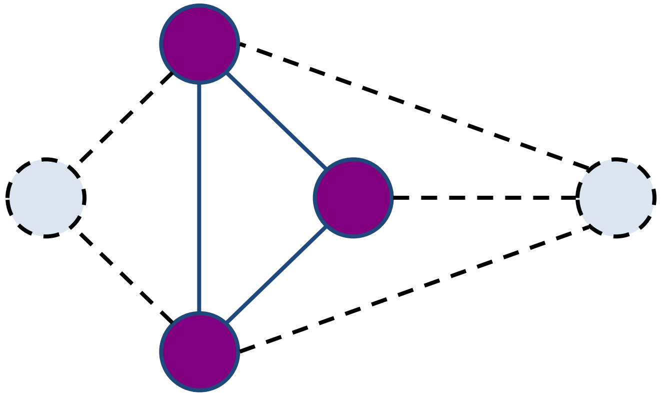
subgraph of the vertices and edges reachable from ORD

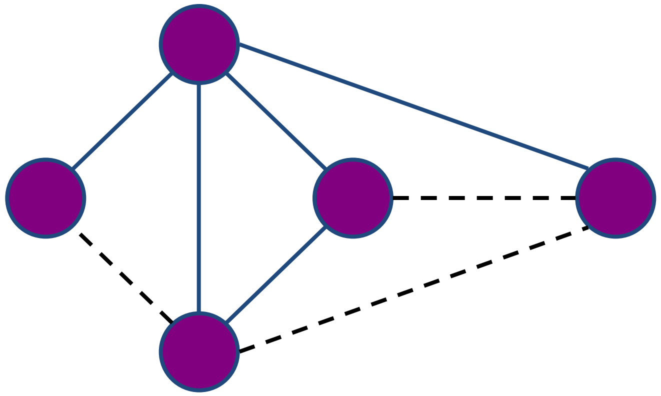


removal of the dashed edges results in a directed acyclic graph

SUBGRAPHS

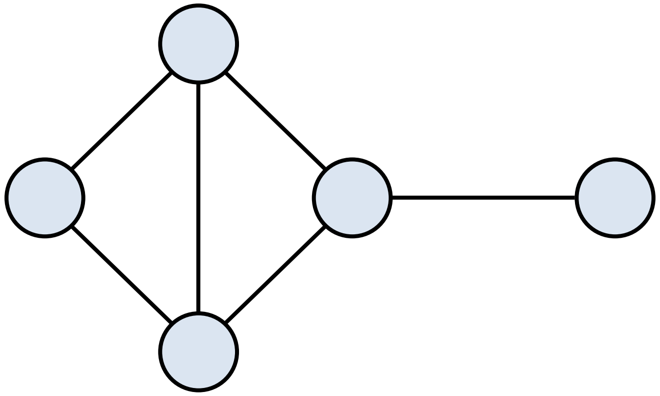
- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G

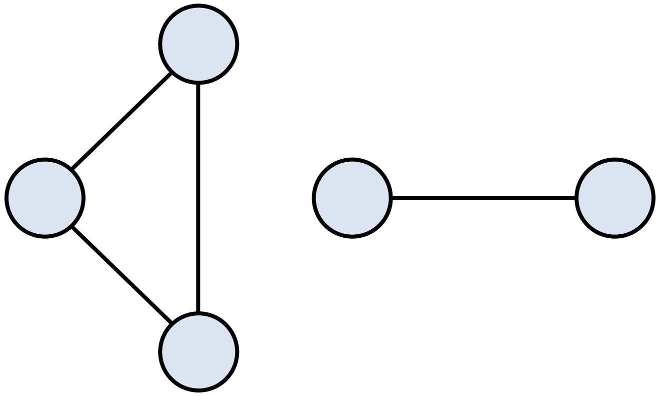

Subgraph


Spanning subgraph

CONNECTIVITY

- A graph is ***connected*** if, for any two vertices, there is a path between them.
- A directed graph *G* is ***strongly connected*** if for any two vertices *u* and *v* of *G*, *u* reaches *v* and *v* reaches *u*.
- A connected component of a graph G is a maximal connected subgraph of G


Connected graph

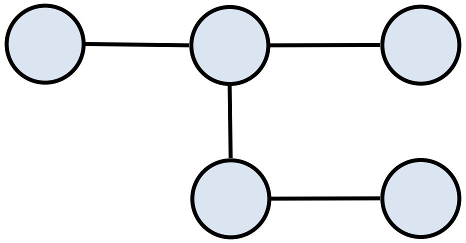

Non connected graph with two connected components

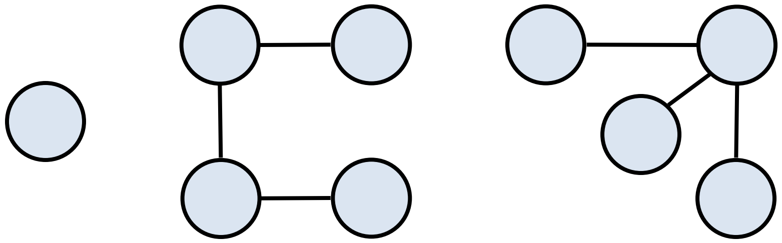
TREES AND FORESTS

- A (free) **tree** is an undirected graph T such that
  - T is connected
  - T has no cycles

  <u>This definition of tree is different from the one of a rooted tree</u>

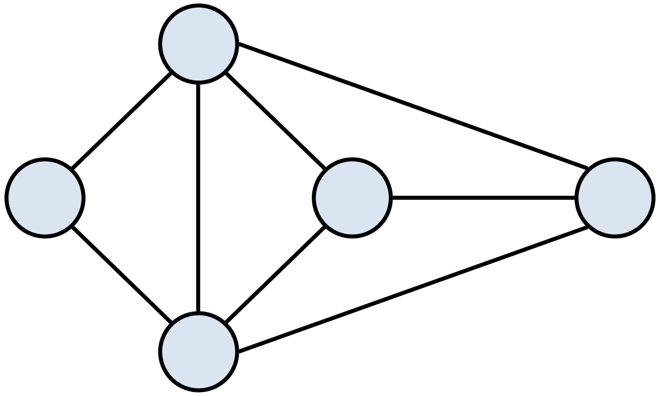
- A **forest** is an undirected graph without cycles
- The connected components of a forest are trees

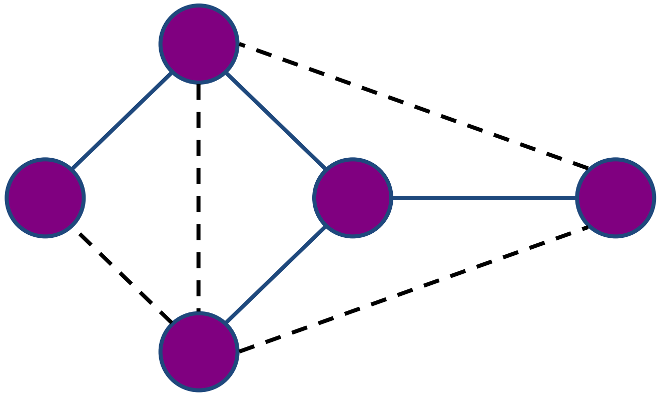

Tree


Forest

SPANNING TREES AND FORESTS

- A **spanning tree** of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- A spanning forest of a graph is a spanning subgraph that is a forest


Graph


Spanning tree

PROPERTIES

Property 1: If *G* is a graph with *m* edges and vertex set *V*, then

$$\sum_{v \text{ in } V} \mathbf{deg}(v) = 2m$$

Proof: each edge is counted twice

Property 2: If *G* is a directed graph with *m* edges and vertex set *V*, then

$$\sum_{v \text{ in } V} \mathbf{indeg}(v) = \sum_{v \text{ in } V} \mathbf{outdeg}(v) = m$$

Property 3: Let *G* be a simple graph with *n* vertices and *m* edges. If *G* is undirected, then

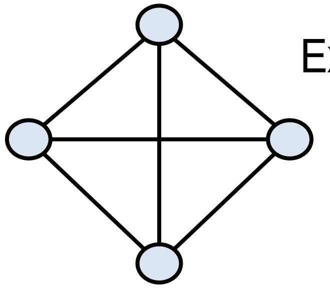
$$m \leq n(n-1)/2$$

Proof: each vertex has degree at most $(n - 1)$

=> A simple graph with *n* vertices has $O(n^2)$ edges.

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

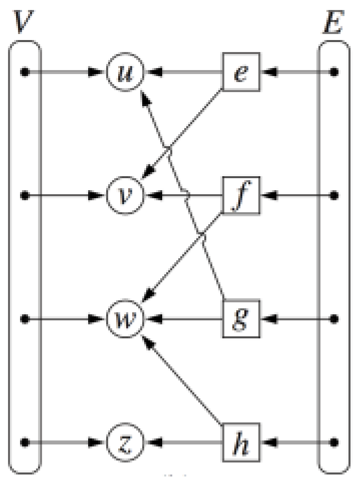
Let *G* be an undirected graph

- If *G* is connected, then $m \geq n-1$.
- If *G* is a tree, then $m = n-1$.
- If *G* is a forest, then $m \leq n-1$.

DATA STRUCTURES FOR GRAPHS

- In an ***edge list***, we maintain an unordered list of all edges.
  - This minimally suffices, but there is no efficient way to locate a particular edge (u,v), or the set of all edges incident to a vertex v.
- In an ***adjacency list***, we additionally maintain, for each vertex, a separate list containing those edges that are incident to the vertex.
  - This organization allows us to more efficiently find all edges incident to a given vertex.
- An ***adjacency map*** is similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key.
  - This allows more efficient access to a specific edge (u,v), for example, in $O(1)$ expected time with hashing.
- An ***adjacency matrix*** provides worst-case $O(1)$ access to a specific edge (u,v) by maintaining an $n \times n$ matrix, for a graph with n vertices.
  - Each slot is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the slot will store null.

PERFORMANCE OF THE EDGE LIST STRUCTURE



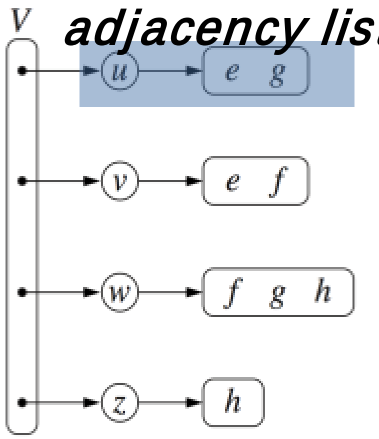
space usage is $O(n+m)$

Method	Running Time
numVertices( ), numEdges( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
getEdge(u, v), outDegree(v), outgoingEdges(v)	$O(m)$
insertVertex(x), insertEdge(u, v, x), removeEdge(e)	$O(1)$
removeVertex(v)	$O(m)$

Exhaustive inspection of all edges needed.

when a vertex v is removed from the graph, all edges incident to v must also be removed

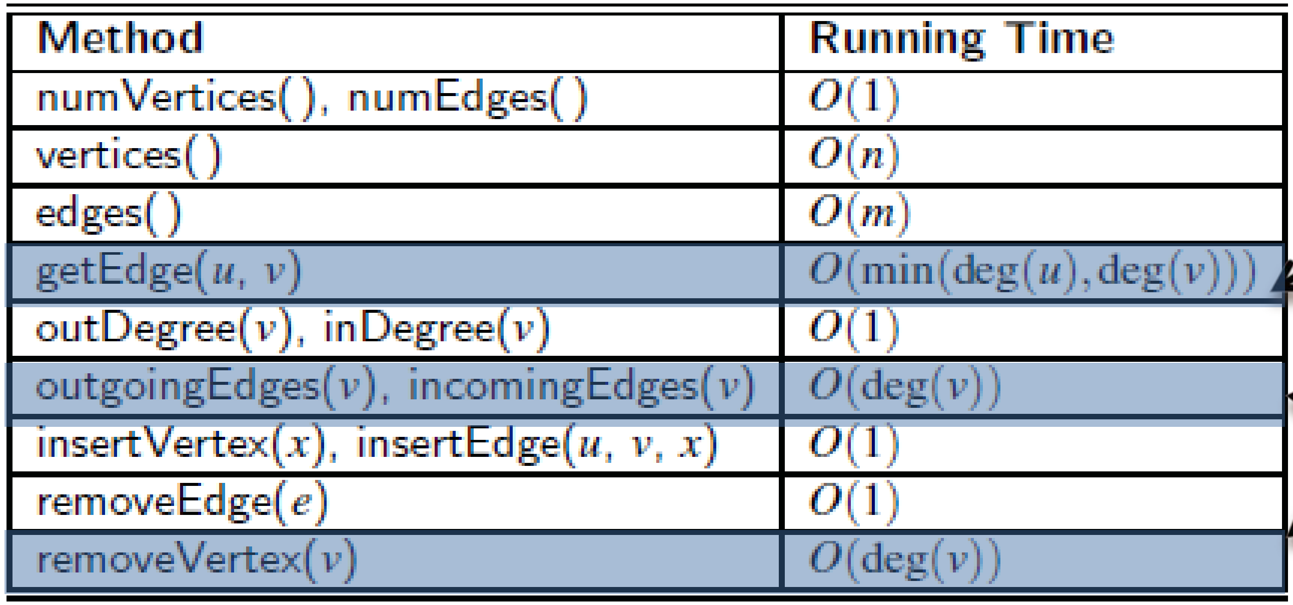
PERFORMANCE OF THE ADJACENCY LIST STRUCTURE



adjacency list $l_{\text{out}}(v)$

assuming that the primary collection V and E, and all secondary collections $I(v)$ are implemented with doubly linked lists.

using $O(n+m)$ space



Method	Running Time
numVertices( ), numEdges( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
getEdge(u, v)	$O(\min(\deg(u), \deg(v)))$
outDegree(v), inDegree(v)	$O(1)$
outgoingEdges(v), incomingEdges(v)	$O(\deg(v))$
insertVertex(x), insertEdge(u, v, x)	$O(1)$
removeEdge(e)	$O(1)$
removeVertex(v)	$O(\deg(v))$

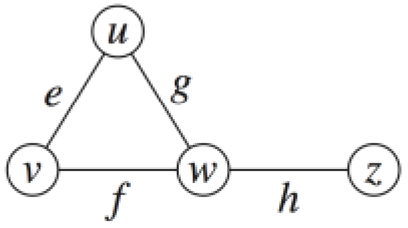
search through either $I(u)$ or $I(v)$

based on use of $I(v)$.

DATA STRUCTURES FOR GRAPHS: ADJACENCY MATRIX

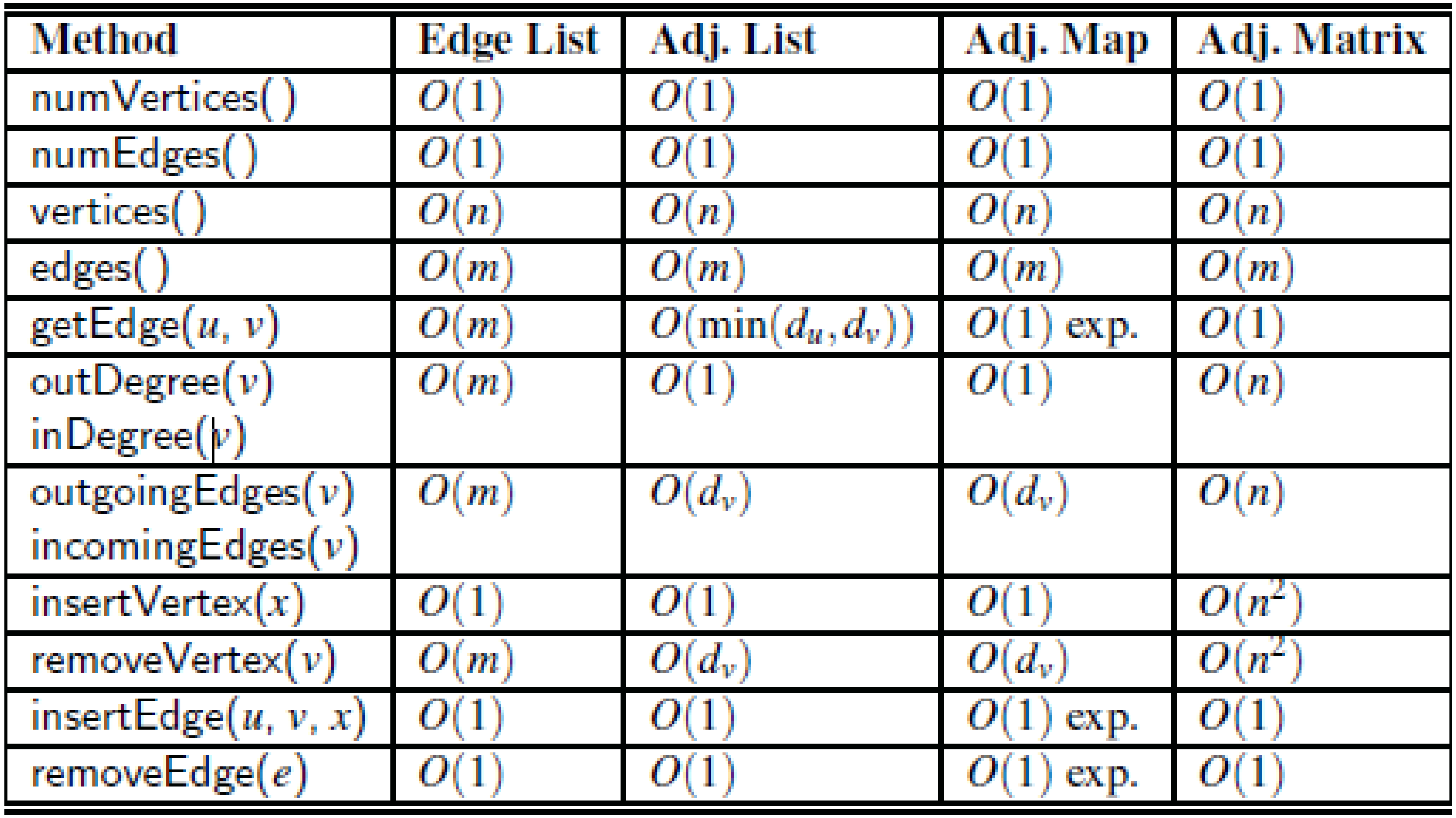
- *adjacency matrix A* allows us to locate an edge between a given pair of vertices in ***worst-case* O(1) time**.
- cell *A*[*i*][*j*] holds a reference to the edge (*u*,*v*), if it exists, where *u* is the vertex with index *i* and *v* is the vertex with index *j*
- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent verti ces
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no e dge and 1 for edge

$O(n^2)$ space usage



		0	1	2	3
u →	0		*e*	*g*	
v →	1	*e*		*f*	
w →	2	*g*	*f*		*h*
z →	3			*h*	

PERFORMANCE: SIMPLE GRAPH



Method	Edge List	Adj. List	Adj. Map	Adj. Matrix
numVertices( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
numEdges( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$

outDegree(v)
inDegree(v)
	$O(m)$	$O(1)$	$O(1)$	$O(n)$

outgoingEdges(v)
incomingEdges(v)
	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$

insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

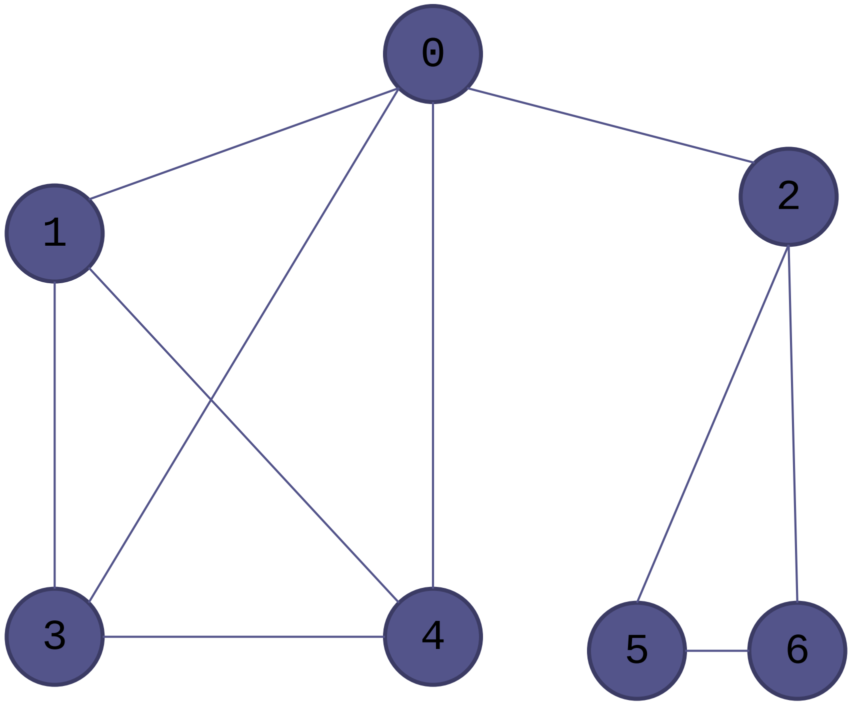
adjacency matrix uses $O(n^2)$ space, while all other structures use $O(n+m)$ space

GRAPH TRAVERSAL

- A *traversal* is a systematic procedure for exploring a graph by examining all of its vertices and edges.
- A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.
- We will look at two efficient graph traversal algorithms
  - *depth-first search (DFS)*
  - *breadth-first search (BFS)*

Example of a Depth-First Search (cont.)

Mark 0 as visited



Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6, 0

Finish order:
4, 3, 1, 6, 5, 2, 0

0
unvisited
0
visited
0
being visited

Breadth-First Search

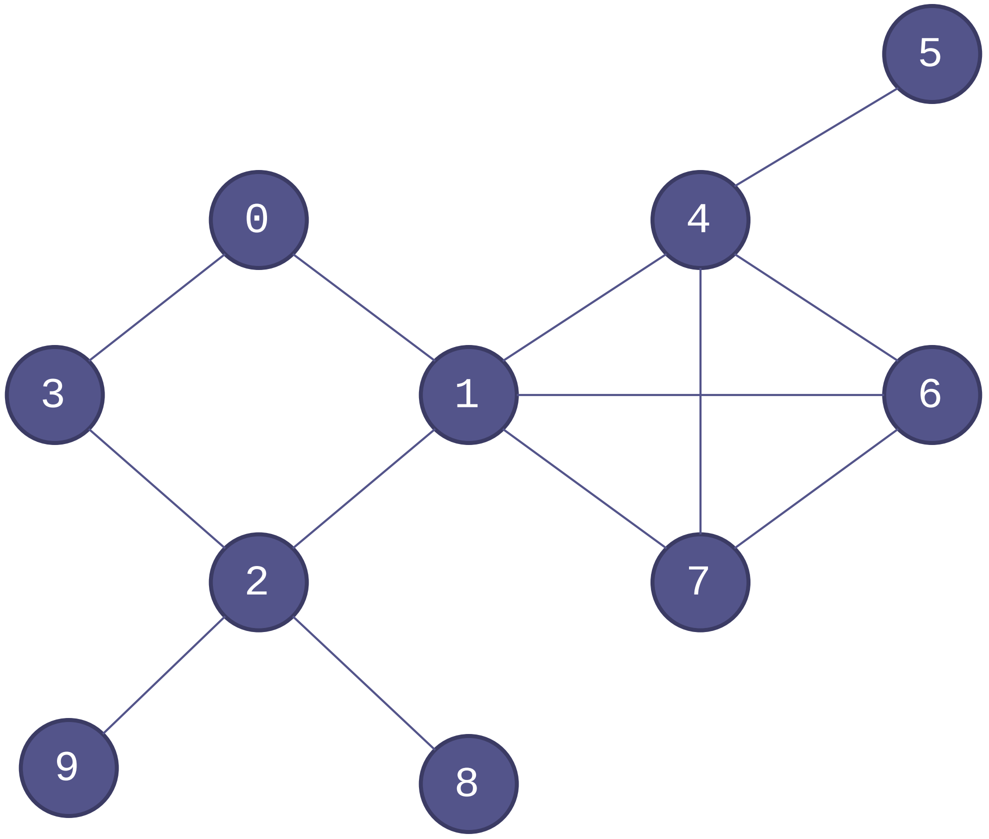
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

Example of a Breadth-First Search (cont.)

The queue is empty; all vertices have been visited

Queue:

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9, 5



0
unvisited
0
visited
0
identified

Breadth-First Search Properties

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

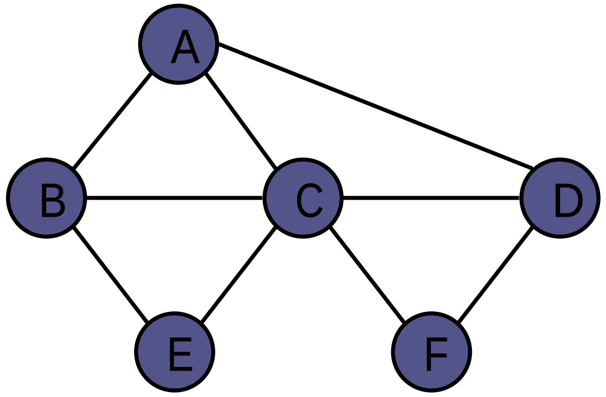
Property 2

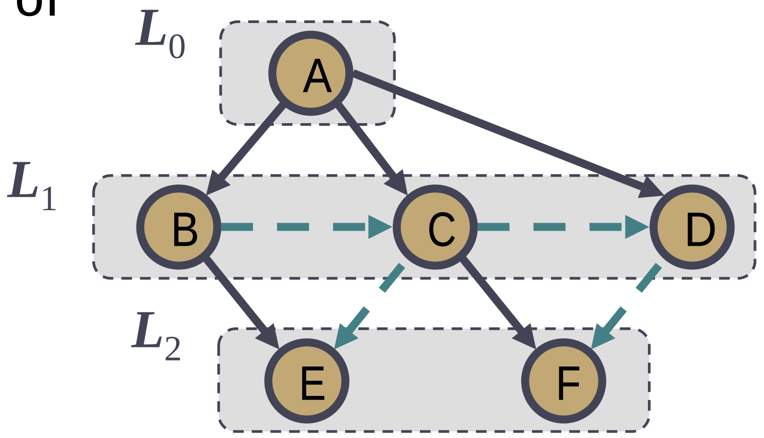
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges

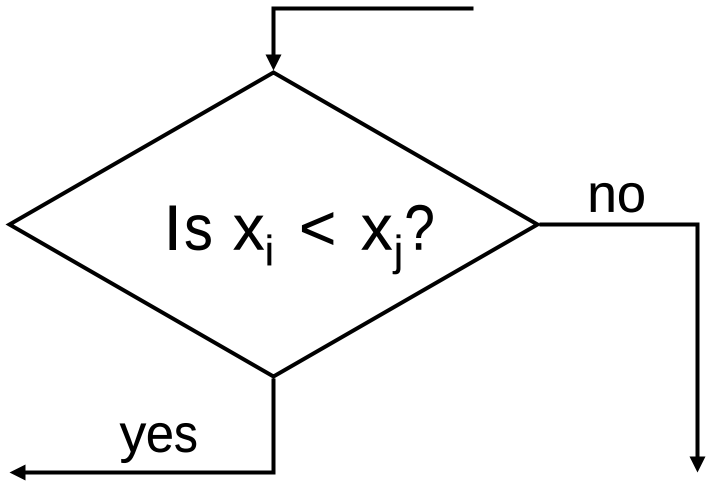




SORTING ALGORITHMS

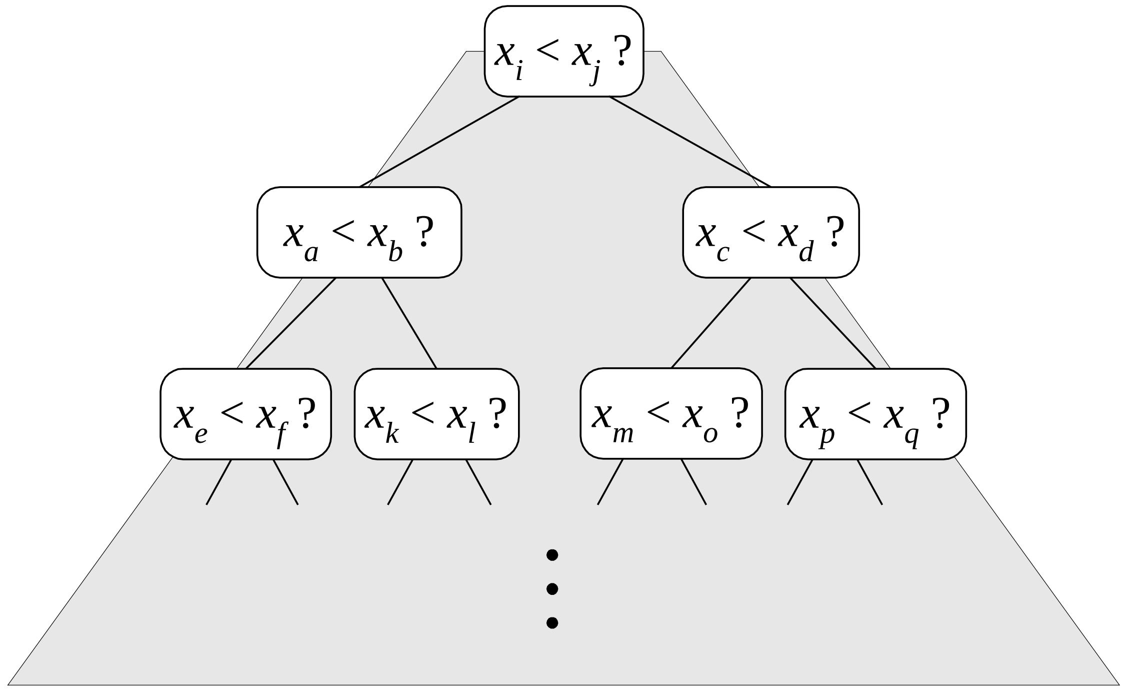
COMPARISON-BASED SORTING

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.



COUNTING COMPARISONS

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree



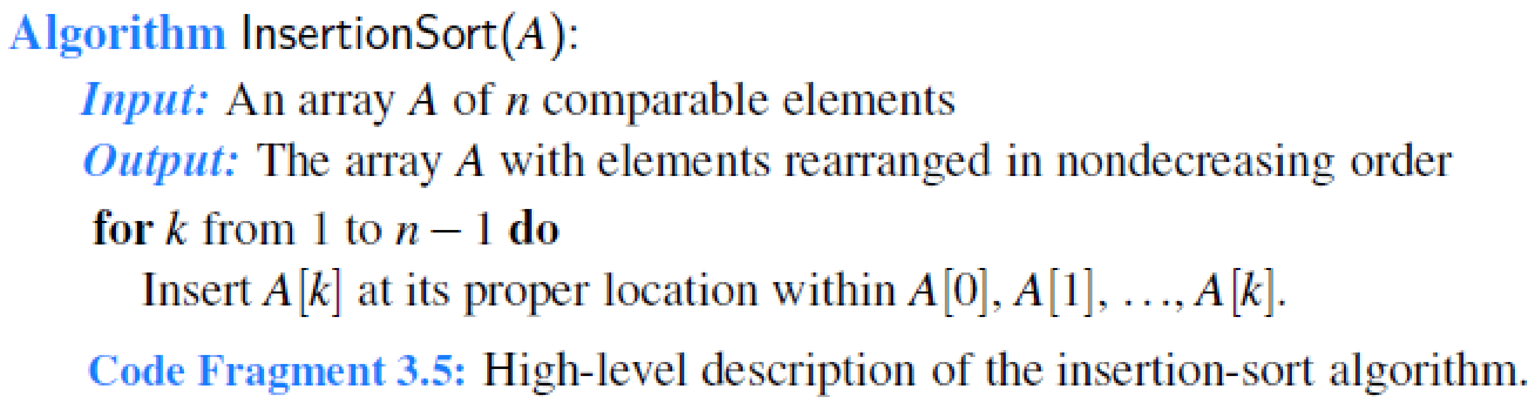
THE LOWER BOUND

- Any comparison-based sorting algorithms takes at least log (n!) time
- Therefore, any such algorithm takes time at least

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2).$$

- That is, any comparison-based sorting algorithm must run in Ω(n log n) lower bound on its running time.

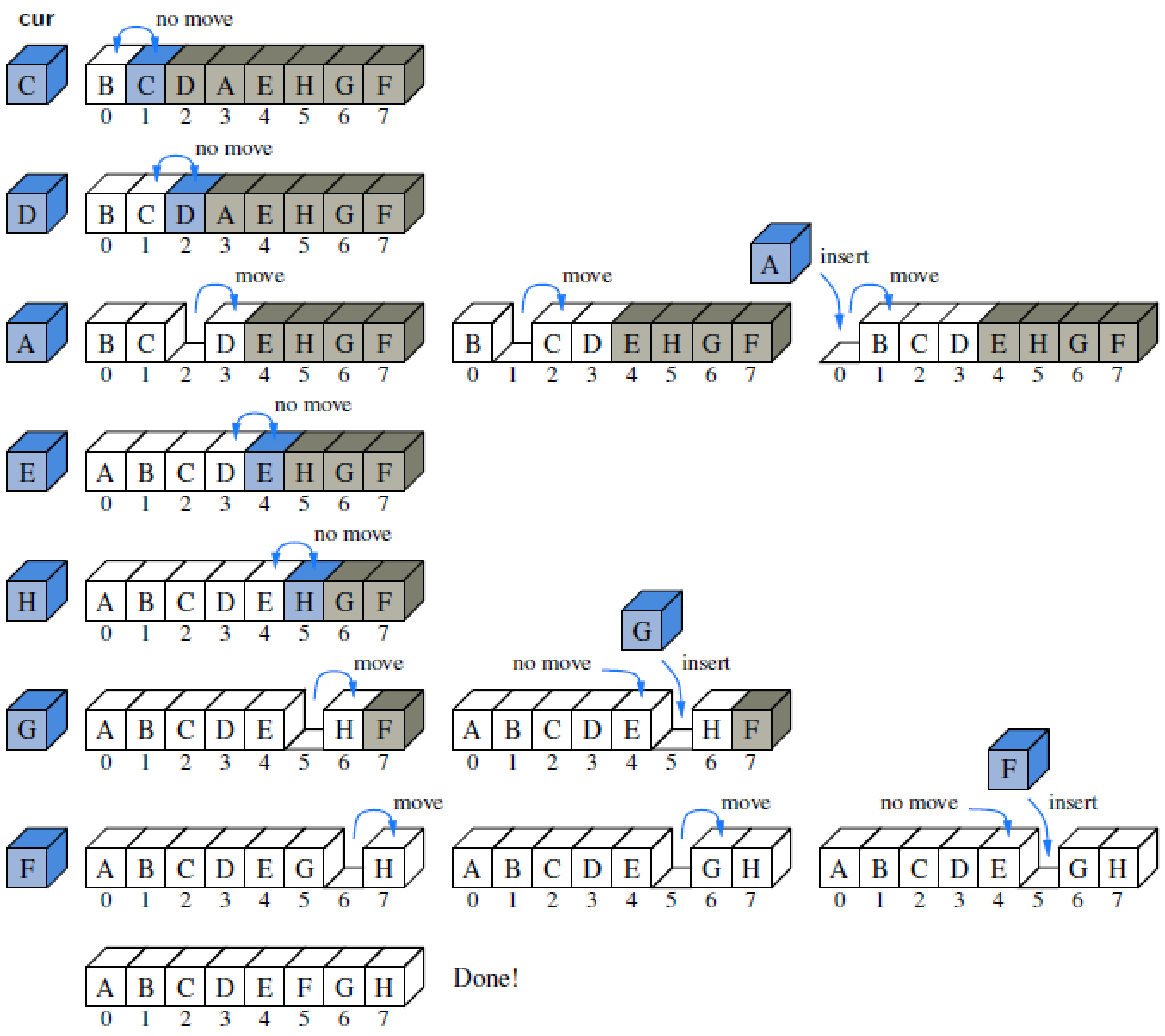
INSERTION-SORT ALGORITHM (IN-PLACE INSERTION-SORT)



```
Algorithm InsertionSort($A$):
    Input: An array $A$ of $n$ comparable elements
    Output: The array $A$ with elements rearranged in nondecreasing order
    for $k$ from 1 to $n-1$ do
        Insert $A[k]$ at its proper location within $A[0], A[1], \ldots, A[k]$.
```

Code Fragment 3.5: High-level description of the insertion-sort algorithm.

The algorithm proceeds by considering one element at a time, placing the element in the correct order relative to those before it.


cur
no move
move
insert
Done!
C
D
A
E
H
G
F
B C D A E H G F
A B C D E F G H
0 1 2 3 4 5 6 7

INSERTION-SORT

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to

$$O(1+2+\ldots+(n-1)+n) = O\left(\sum_{i=1}^{n} i\right)$$

  2. Removing th m the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

SELECTION-SORT

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$O(n + (n-1) + \cdots + 2 + 1) = O\left(\sum_{i=1}^{n} i\right)$$

- Selection-sort runs in $O(n^2)$ time

HEAP SORT

- Consider the pqSort scheme, this time using a heap-based implementation of the priority queue
- Phase 1: insert all data into heap:
  - takes $O(n\log n)$ time. (Could be improved to $O(n)$ with bottom-up construction)
- Phase 2: removeMin all data in the heap
  - j th removeMin operation runs in $O(\log(n-j+1))$, since the heap has $n-j+1$ entries at the time the operation
  - Summing over all j, this phase takes $O(n\log n)$ time
- Overall: The heap-sort algorithm sorts a sequence S of n elements in $O(n\log n)$ time, assuming two elements of S can be compared in $O(1)$ time.

MERGE-SORT

- Merge-sort on an input sequence S with n elements consists of three steps:
  - *Divide*: If S has zero or one element, return S. *Otherwise* partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - *Conquer*: recursively sort S_1 and S_2
  - *Combine*: merge sorted S_1 and sorted S_2 into a unique sorted sequence

```
Algorithm mergeSort(S)
    Input sequence S with n
        elements
    Output sequence S sorted
        according to C
    if S.size() > 1
        (S_1, S_2) ← partition(S, n/2)
        mergeSort(S_1)
        mergeSort(S_2)
        S ← merge(S_1, S_2)
```

MERGING TWO SORTED SEQUENCES

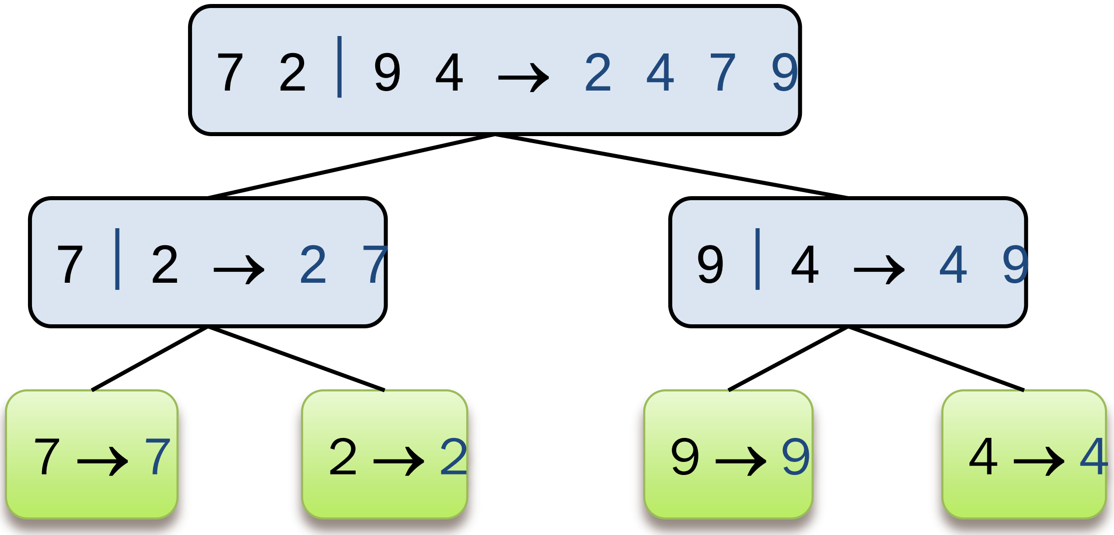
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

```
Algorithm merge(A, B)
    Input sequences A and B with
          n/2 elements each
    Output sorted sequence of A ∪ B

    S ← empty sequence
    while ¬A.isEmpty()  ∧ ¬B.isEmpty()
        if A.first().element() < B.first().element()
            S.addLast(A.remove(A.first()))
        else
            S.addLast(B.remove(B.first()))
    while ¬A.isEmpty()
        S.addLast(A.remove(A.first()))
    while ¬B.isEmpty()
        S.addLast(B.remove(B.first()))
    return S
```

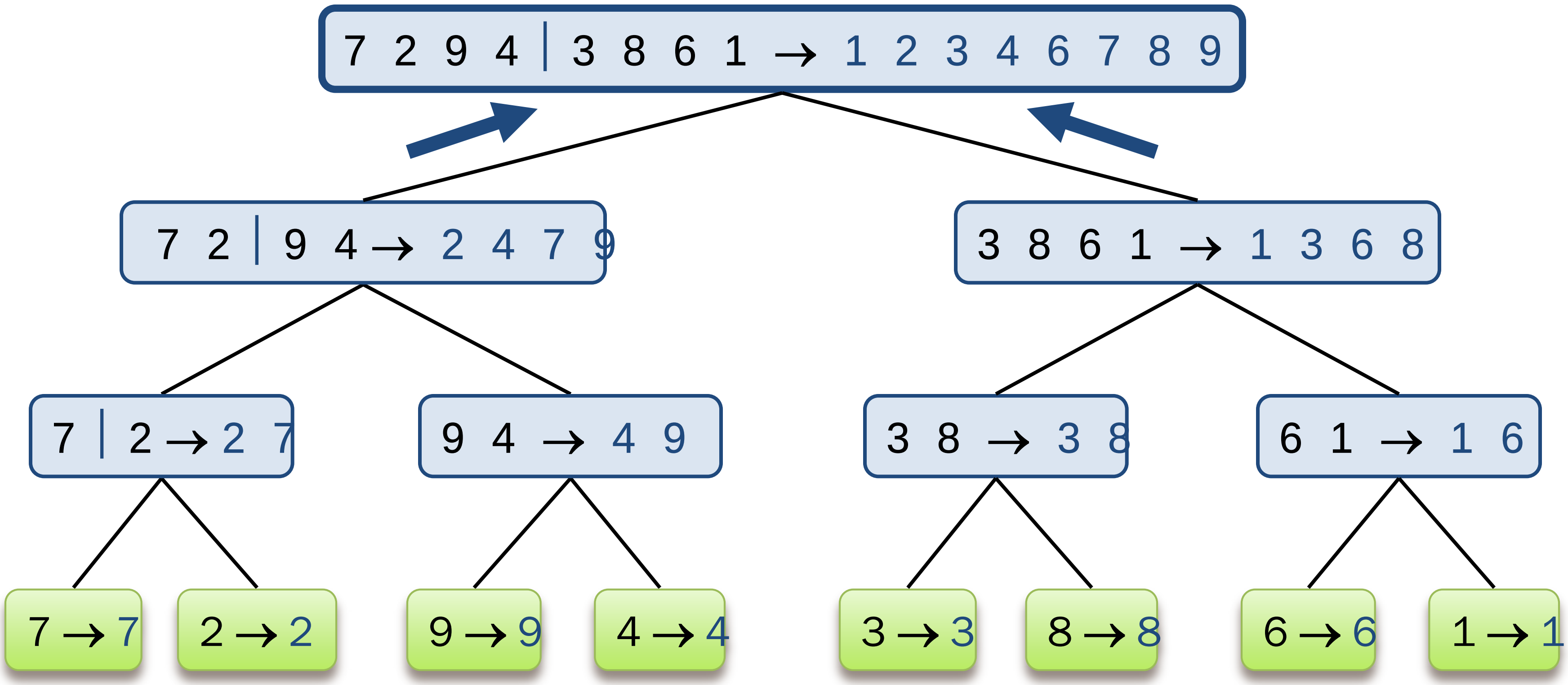
MERGE-SORT TREE

- An execution of merge-sort is depicted by a binary tree T, called the *merge-sort tree*
  - Each **node** represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the **root** is the initial call
  - the **leaves** are calls on subsequences of size 0 or 1



EXECUTION EXAMPLE (CONT.)

- Merge



ARRAY-BASED IMPLEMENTATION OF MERGE-SORT 1

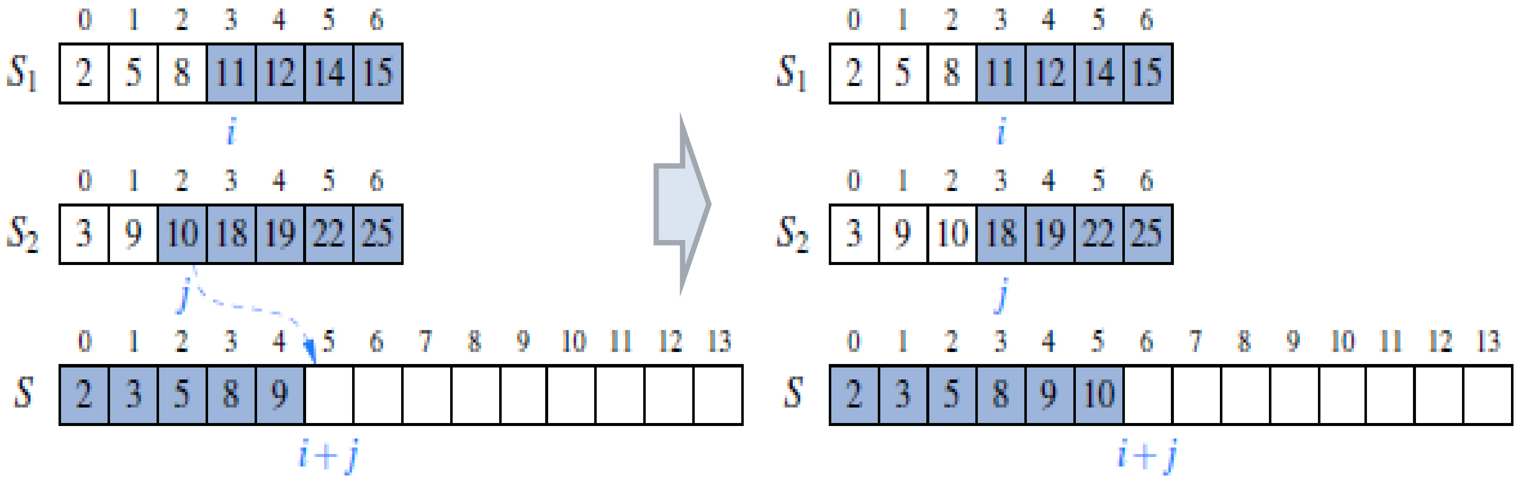
```
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
  int n = S.length;
  if (n < 2) return;                                // array is trivially sorted
  // divide
  int mid = n/2;
  K[ ] S1 = Arrays.copyOfRange(S, 0, mid);          // copy of first half
  K[ ] S2 = Arrays.copyOfRange(S, mid, n);          // copy of second half
  // conquer (with recursion)
  mergeSort(S1, comp);                              // sort copy of first half
  mergeSort(S2, comp);                              // sort copy of second half
  // merge results
  merge(S1, S2, S, comp);                 // merge sorted halves back into original
}
```

ARRAY-BASED IMPLEMENTATION OF MERGE-SORT 2

```
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {
  int i = 0, j = 0;
  while (i + j < S.length) {
    if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
      S[i+j] = S1[i++];                  // copy ith element of S1 and increment i
    else
      S[i+j] = S2[j++];                  // copy jth element of S2 and increment j
  }
}
```

indices *i* &*j* represents the number of elements of *S*1 & S2 that have been copied to *S*

A step in the merge of two sorted arrays for which *S*2[*j*] < *S*1[*i*].



	0	1	2	3	4	5	6
S_1	2	5	8	11	12	14	15
S_2	3	9	10	18	19	22	25

Before: $i = 3$, $j = 2$

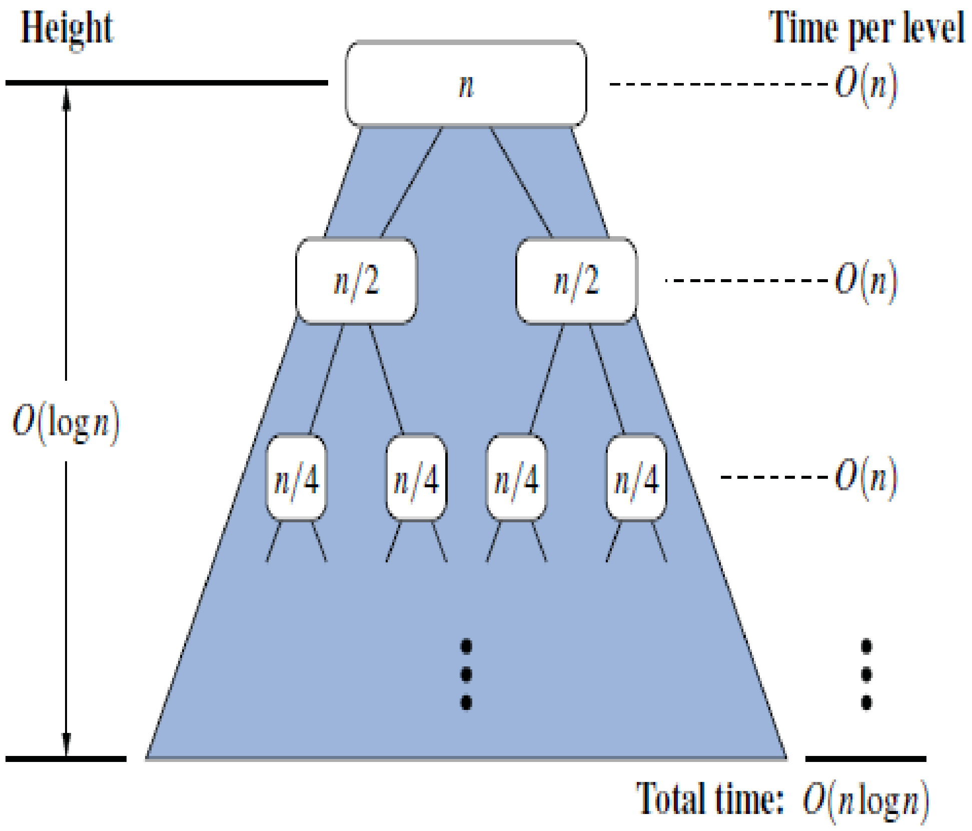
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	2	3	5	8	9									

After: $i = 3$, $j = 3$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S	2	3	5	8	9	10								

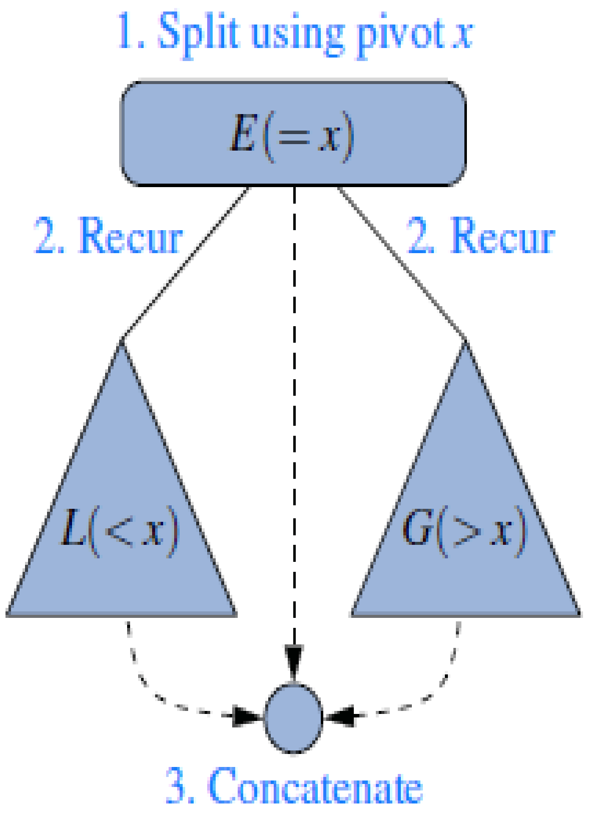
ANALYSIS OF MERGE-SORT

- The height ***h*** of the merge sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall work done at the nodes of depth ***i*** is $O(n$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n\log n)$



QUICK-SORT

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - ***Divide***: pick a random element ***x*** (called pivot) and partition ***S*** into
    - ***L*** elements less than ***x***
    - ***E*** elements equal ***x***
    - ***G*** elements greater than ***x***
  - ***Conquer:*** Recursively sort ***L*** and ***G***
  - ***Combine:*** join ***L***, ***E*** and ***G***



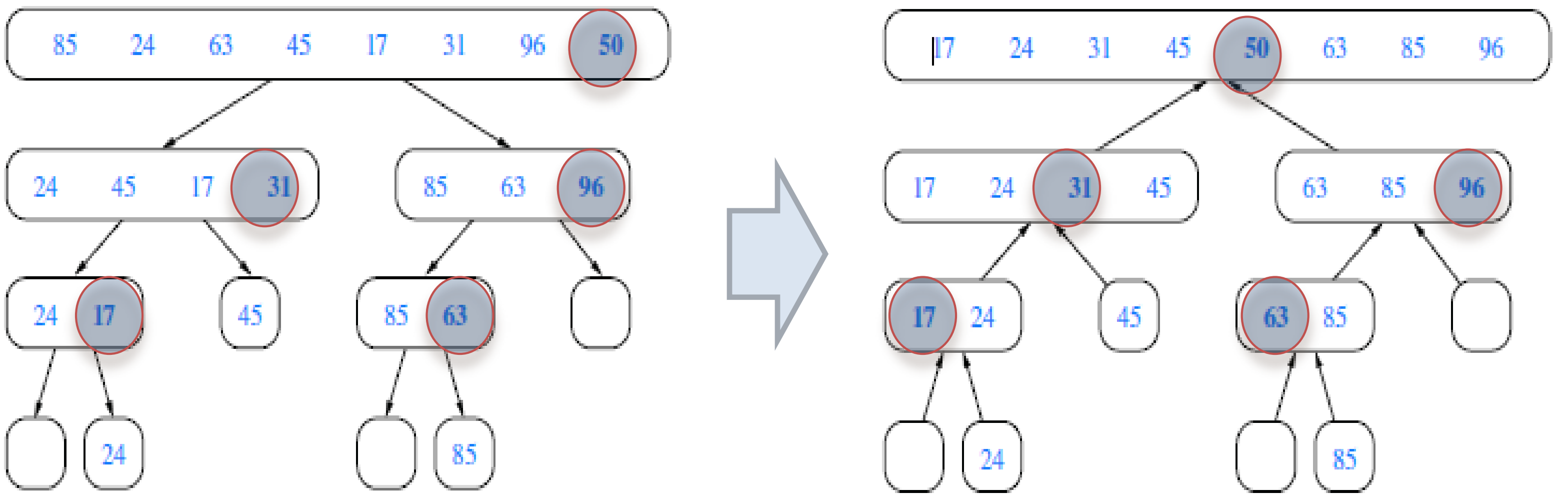
PARTITION

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else { y > x }
            G.addLast(y)
    return L, E, G
```

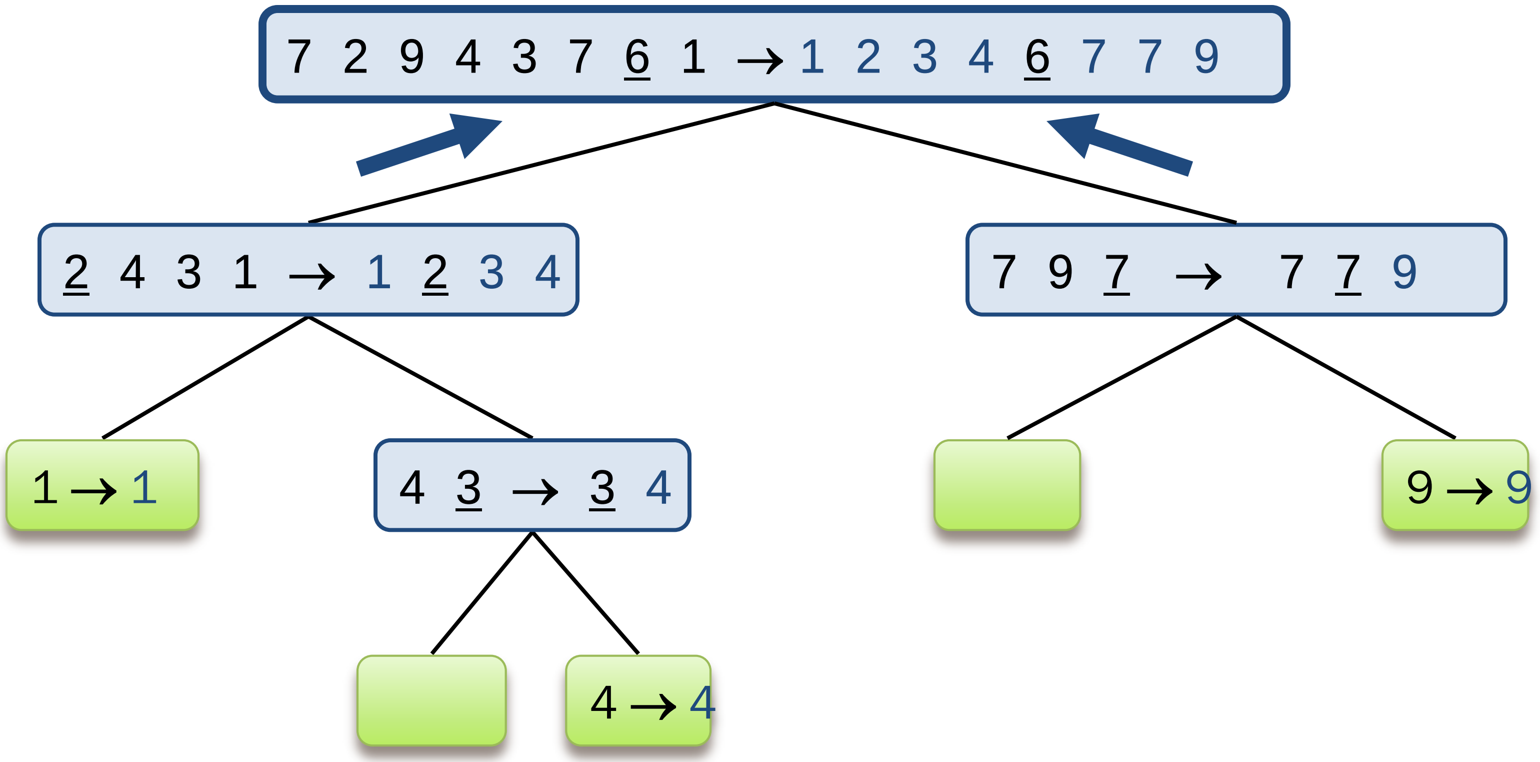
QUICK-SORT TREE

- An execution of quick-sort is depicted by a binary tree called *quick-sort tree*.
  - Each **node** represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The **root** is the initial call
  - The **leaves** are calls on subsequences of size 0 or 1



EXECUTION EXAMPLE (CONT.)

- Join, join



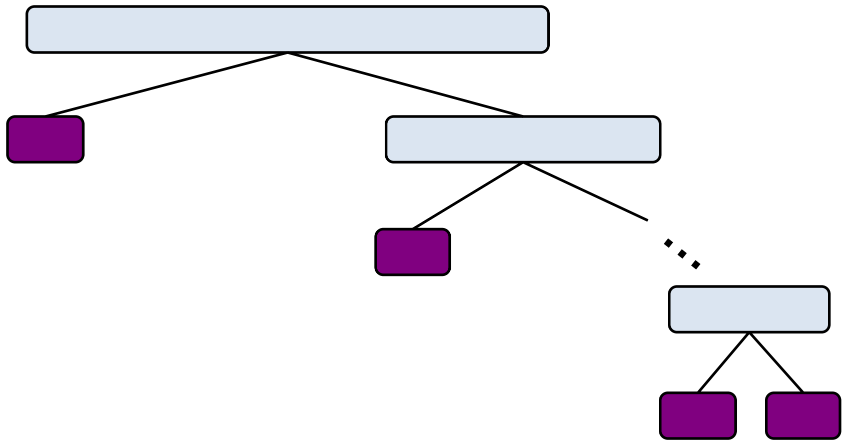
WORST-CASE RUNNING TIME

- The worst case for quick-sort occurs when the pivot is the unique minim
  um or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

- Thus, the <u>worst-case running time of quick-sort is $O(n^2)$</u>

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



LINEAR TIME SORTING

- We showed that the lower bound of sorting with comparison is $\Omega(n\log n)$ time.
- Can we do better? Yes, with special assumptions about the input sequence to be sorted.
- We will consider the problem of sorting a sequence of entries, each a key-value pair, where the keys have a restricted type
  - Bucket-Sort
  - Radix-Sort

BUCKET-SORT

- Let be S be a sequence of n (key, element) entries with integer keys in the range [0, N-1], for some integer N≥2,
- **Bucket-sort** uses the keys as indices into an auxiliary array B of size N (buckets)

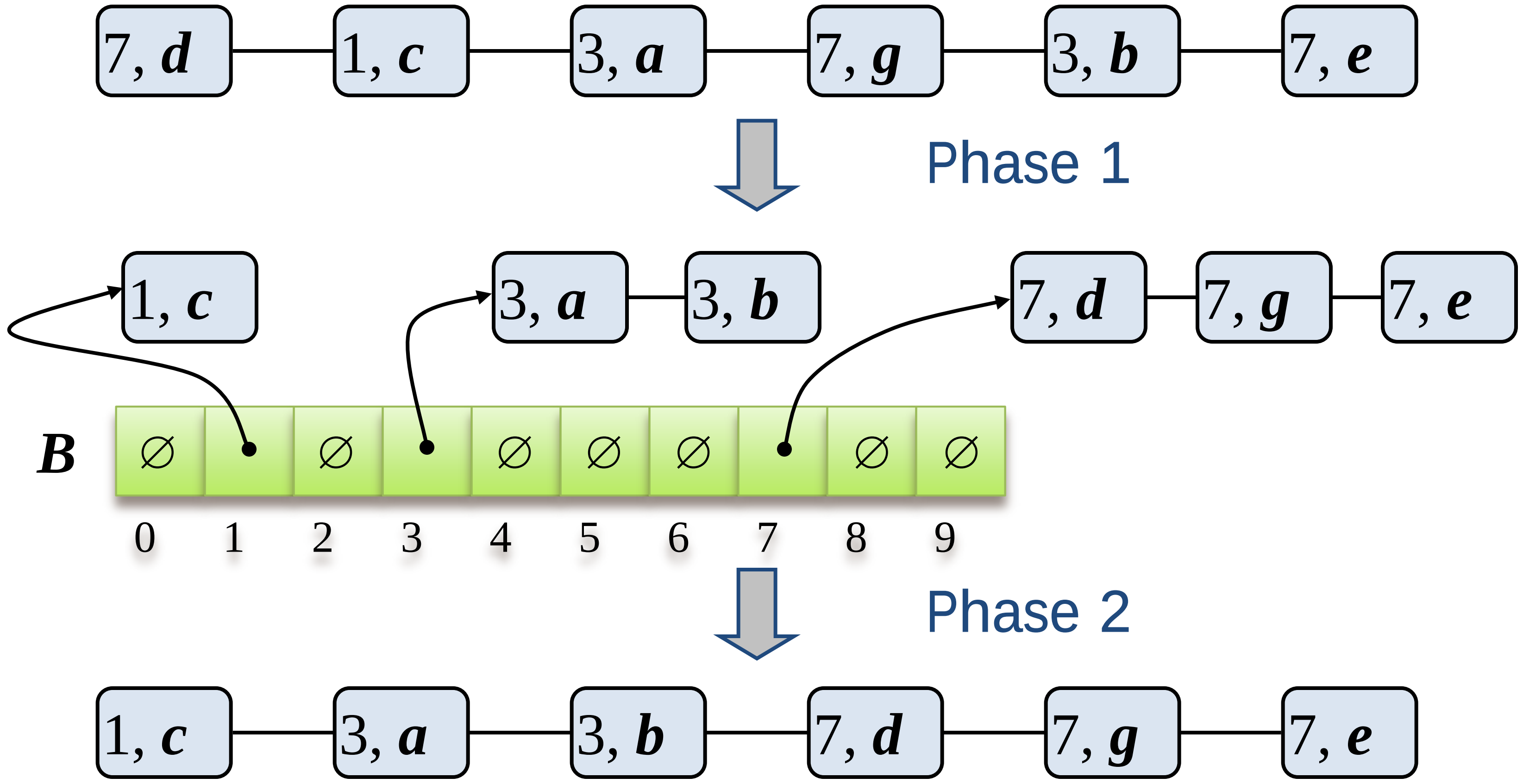
  Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$

  Phase 2: For $i = 0, \ldots, N - 1$, move the entries of bucket $B[i]$ to the end of sequence S
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

  Bucket-sort takes $O(n + N)$ time

EXAMPLE

- Key range [0, 9]



PROPERTIES AND EXTENSIONS

- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator
- Stable Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm
- Extensions
  - Integer keys in the range $[a, b]$
    - Put entry (k, o) into bucket $B[k - a]$
  - String keys from a set D of possible strings, where D has constant size (e.g., names of the 50 U.S. states)
    - Sort D and compute the rank $r(k)$ of each string k of D in the sorted sequence
    - Put entry (k, o) into bucket $B[r(k)]$

STABLE SORTING

- When sorting key-value pairs, an important issue is how equal keys are handled. Let $S = ((k_0,v_0), \ldots , (k_{n-1},v_{n-1}))$ be a sequence of such entries.
- We say that a sorting algorithm is ***stable*** if, for any two entries (k_i,v_i) and (k_j ,v_j) of S such that $k_i = k_j$ and (k_i,v_i) precedes (k_j ,v_j) in S before sorting (that is, $i < j$), entry (k_i,v_i) also precedes entry (k_j ,v_j) after sorting.
- Stability is important for a sorting algorithm because applications may want to preserve the initial order of elements with the same key.
- Bucket-sort guarantees stability as long as we ensure that all sequences act as **queues**

RADIX-SORT

- **Radix-sort** is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, N - 1]$
- Radix-sort runs in time $O(d(n+N))$ where the d is the dimension of keys, n is the number of data, and keys range is [0...N-1]

```
Algorithm radixSort(S, N)
    Input sequence S of d-tuples such
        that $(0, \dots, 0) \le (x_1, \dots, x_d)$ and
        $(x_1, \dots, x_d) \le (N - 1, \dots, N - 1)$
        for each tuple $(x_1, \dots, x_d)$ in S
    Output sequence S sorted in
        lexicographic order
    for $i \leftarrow d$ downto 1
        bucketSort(S, N)
```

SUMMARY OF SORTING ALGORITHMS

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ **expected**	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)
bucket-sort	$O(n+N)$	▪ integer keys of range [0 … N]
radix-sort	$O(d(n+N))$	▪ d integer keys of range [0 … N]

What would work best when the set is already sorted or almost sorted?