4CCS1DST Data Structures

1ST YEAR SECOND SEMESTER

Franch Tapia, Alex
KING'S COLLEGE LONDON | JAN-MAY

Table of Contents

4CCS1DST Data Structures

The Recursion Pattern

Recursion: when a method calls itself.

Example 1: the **factorial** function

$n! = 1 * 2 * \ldots * (n-1) * n$ and let zero factorial be 1

- Recursive definition:

$$f(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot f(n-1), & \text{if } n \geq 1 \end{cases}$$

- How to use it:

$$f(0) = 1; \quad f(1) = 1 \cdot f(0) = 1 \cdot 1 = 1;$$

$$f(4) = 4 \cdot f(3) = 4 \cdot (3 \cdot f(2)) = 4 \cdot (3 \cdot (2 \cdot f(1)))$$
$$= 4 \cdot (3 \cdot (2 \cdot (1 \cdot f(0)))) = 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) = 24$$

Another recursive phrase could be *My ancestors are my father and all his ancestors* – example.

I use the idea of ancestors twice.

Key features of a recursive method

- There should be a base case (or more).
- Every possible chain of recursive calls must reach a base case.
- Each recursive call should make progression towards a base case.

Visualizing Recursion

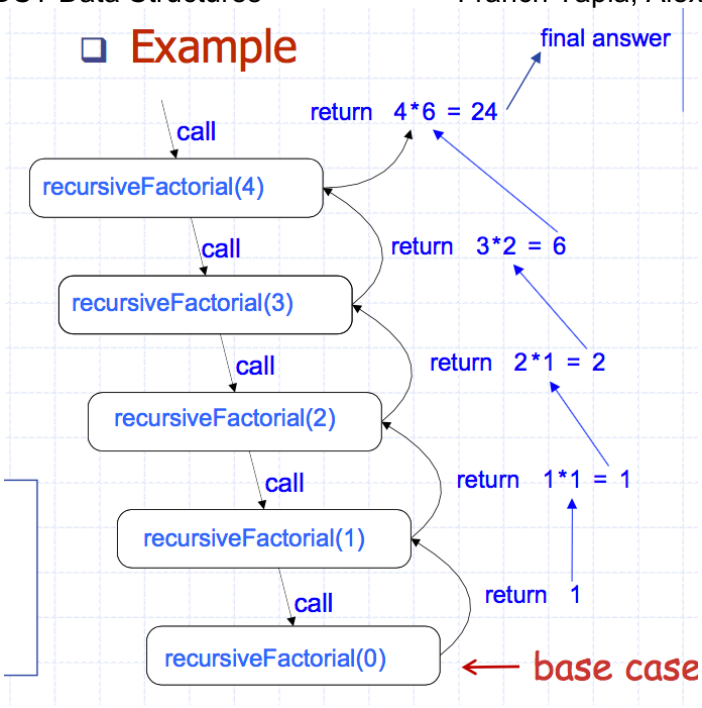
To visualize recursion, we create a **recursion trace**

This contains:

- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

Given a certain code:

```
public static int  recursiveFactorial(int n)
{
   if (n <= 0) return 1;
   else return n * recursiveFactorial(n - 1);
}
```

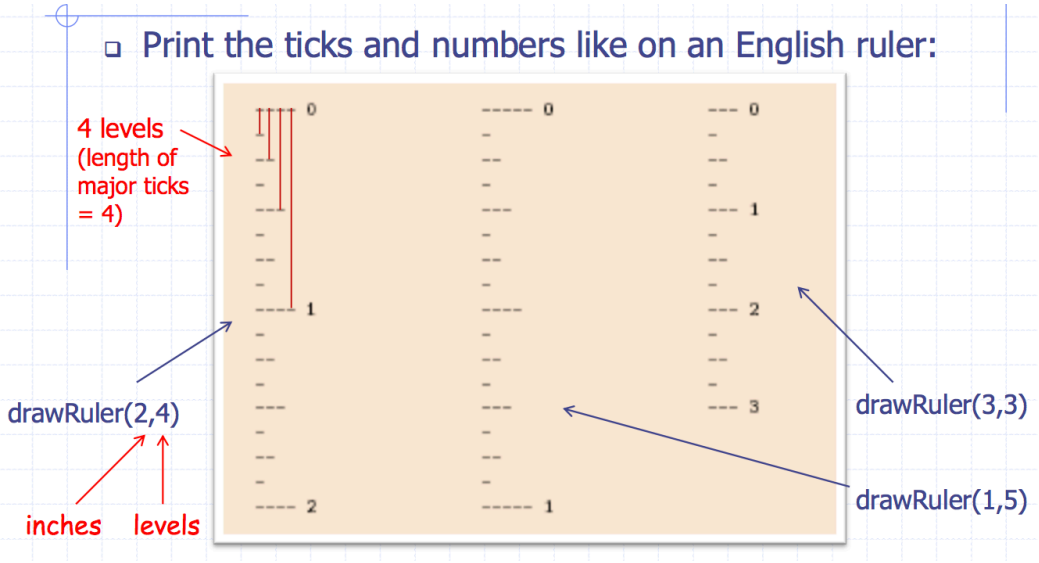


Example 2: English Ruler

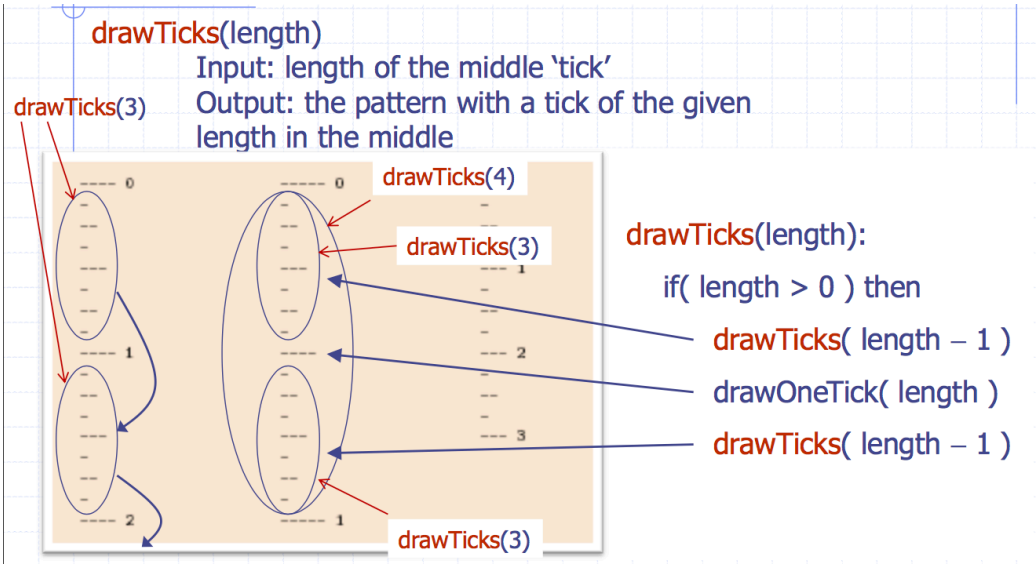
An English ruler has inches. We would like to print the ticks in the ruler given two arguments, the length of the ruler and the numbers on the ruler.

drawRuler(int lengthOfRuler, int levels)

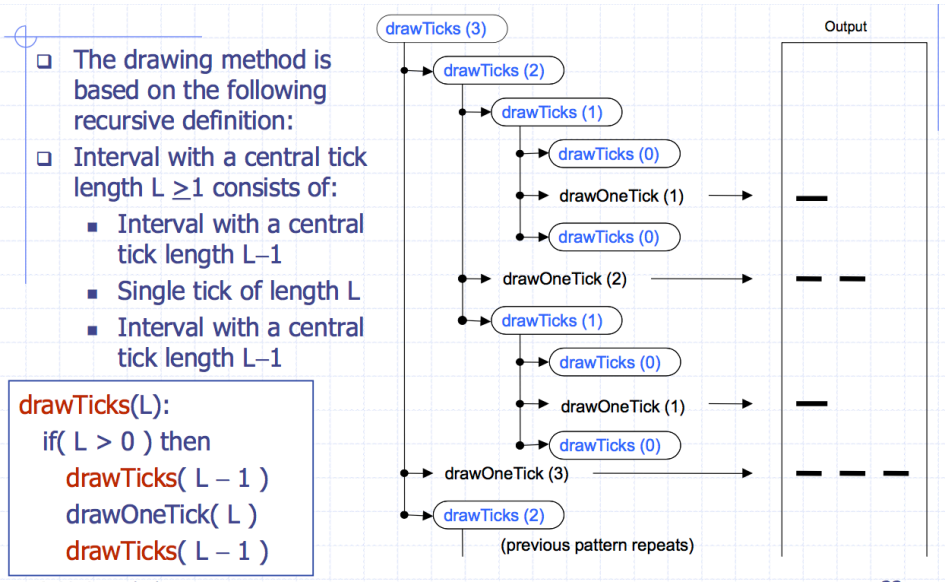
We need a recursive base, hence the pattern for one inch. At the same time we want to print the ticks and numbers on the ruler.



Using recursion:



We can visualize this:



In code, this can be implemented (in a more complex way):

```
package lecture1;
public class Ruler {

   // draw ruler  (one loop; each iteration draws one inch)
   public static void drawRuler(int nInches, int majorLength) { … }

   // draw inner ticks for one unit at level tickLength+1 (recursive method)
   public static void drawTicks(int tickLength) { … }

   // draw one tick
   public static void drawOneTick(int tickLength) { … }

   // draw one tick with a label
   public static void drawOneTick(int tickLength, int tickLabel) { … }

   // test
   public static void main(String args[ ]) { Ruler.drawRuler(2, 4); }
}
```

```
// draw ruler  (one loop; each iteration draws one inch)
public static void drawRuler(int nIns, int majorL) {
   drawOneTick(majorL, 0);          // draw tick 0 and its label
   for (int i = 1; i <= nIns; i++) {
      drawTicks(majorL - 1);        // draw ticks for this inch
      drawOneTick(majorL, i);       // draw tick i and its label
   }
}
// draw ticks of given length for one inch (recursive method)
public static void drawTicks(int tickLength) {
   if (tickLength > 0) {            // stop when length drops to 0
      drawTicks(tickLength - 1);    // recursively draw left ticks
      drawOneTick(tickLength);      // draw center tick
      drawTicks(tickLength - 1);    // recursively draw right ticks
   }
}
```

```
drawRuler(3,4):

---- 0
...
---- 1
...
---- 2
...
---- 3
```

```
// draw one tick with a label
public static void drawOneTick(int tickLength, int tickLabel) {
  for (int i = 0; i < tickLength; i++)
    System.out.print("-");
  if (tickLabel >= 0) System.out.print(" " + tickLabel);
  System.out.print("\n");
}

// draw a tick with no label
public static void drawOneTick(int tickLength) {
  drawOneTick(tickLength, - 1);
}
```

Linear Recursion

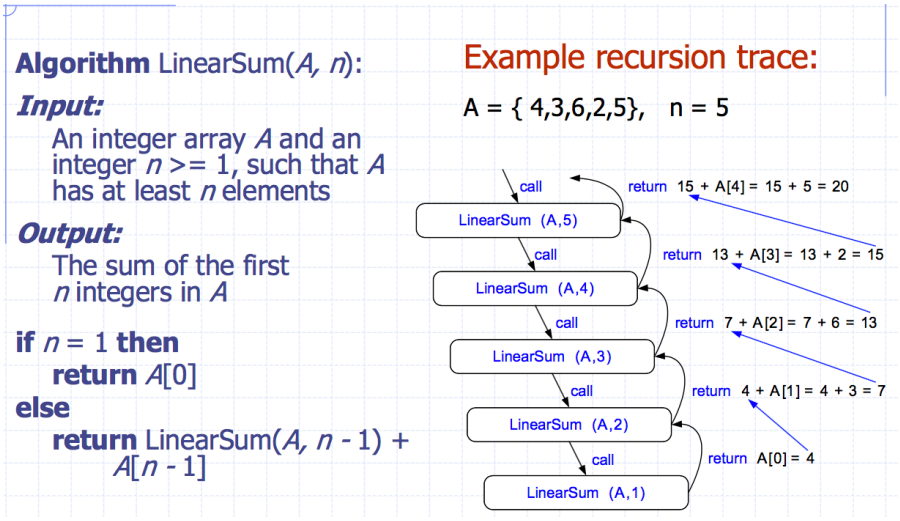
Linear recursion: When recursion only occurs once.

Test for base case:

- Test a set of base cases
- **Every** possible chain of recursive calls **must** eventually reach a base case (if not an infinite loop will occur), the base case should not use recursion.

Recur once:

- Check that if there is a decision to make (regarding what recursive call to make) it always chooses the correct one.
- Make sure all recursive calls progress towards a base case.



Computing Powers

The power function: $p(x, n) = x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,n-1) & \text{else} \end{cases}$$

Example: $p(5,3) = 5 \bullet (p,2) = 5 \bullet \big(5 \bullet p(5,1)\big) = 5 \bullet \Big(5 \bullet \big(5 \bullet p(5,0)\big)\Big) = 5 \bullet 5 \bullet 5 \bullet 1$

This power function that runs in time linear in n (we have n, recursive calls; n multiplications) but this can be faster by recursive squaring.

Recursive Squaring

$2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16$ (3 multiplications)
$2^4 = (2^2)^2 = 4^2 = 16$ (2 squarings)

As we can see by doing this we are decreasing the number of operations by a large amount, this difference becomes larger the larger the squaring.

$2^{32} = 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 = 4{,}294{,}967{,}296$ (31 multiplications)
$2^{32} = (2^{16})^2 = ((((2^2)^2)^2)^2)^2 = \ldots$ (5 squarings)

But we have to be careful because sometimes we can come across a non, even number so we have to take a 2 out to make the index even.

$2^{14} = 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 = 16{,}384$ (13 multiplications)
$2^{14} = (2^7)^2 = (2 \cdot 2^6)^2 = (2 \cdot (2^3)^2)^2 = (2 \cdot (2 \cdot 2^2)^2)^2 = \ldots$
(5 multiplications/squarings)

We can derive a more efficient linearly recursive algorithm using repeated squaring:

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

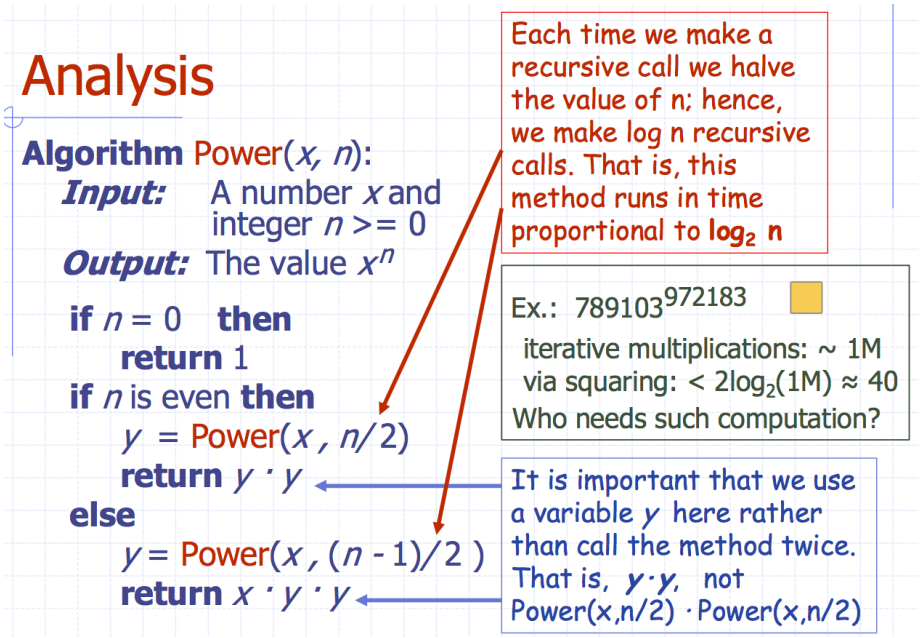
$$p(x,n) = \begin{cases} 1, & \text{if } n = 0 \\ (p(x,n/2))^2, & \text{if } n > 0 \text{ is even} \\ x \cdot (p(x,(n-1)/2))^2, & \text{if } n > 0 \text{ is odd} \end{cases}$$

- Example:

$x^{14} = p(x, 14) = [\, p(x, 7)\,]^2 =$
$= [x \cdot [p(x, 3)]^2\,]^2 = [x \cdot [x \cdot [p(x, 1)]^2\,]^2\,]^2$
$= [x \cdot [x \cdot [x \cdot [p(x, 0)]^2\,]^2\,]^2\,]^2$
$= [x \cdot [x \cdot [x \cdot 1]^2\,]^2\,]^2$

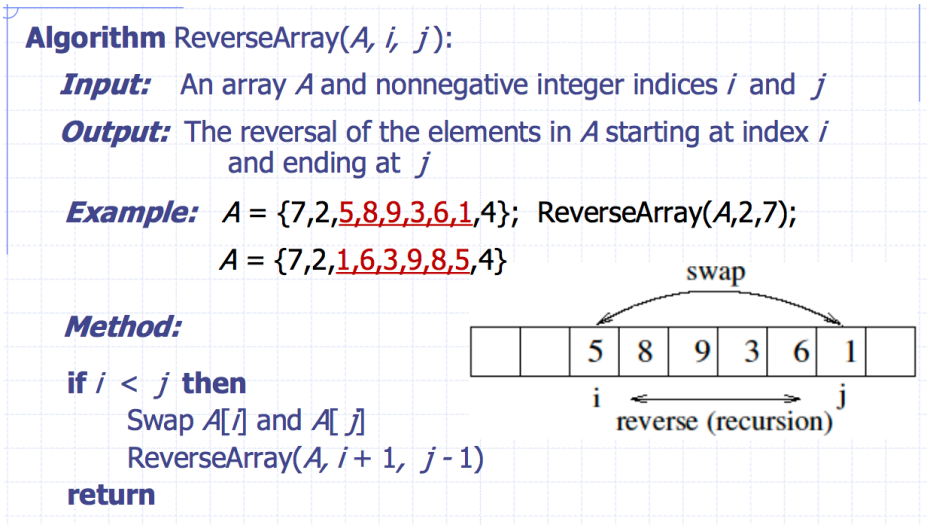
Algorithm Power(*x*, *n*):
Input: A number *x* and integer *n* >= 0
Output: The value x^n

```
if n <= 0 then
    return 1
if n is even then
    y = Power(x, n/2)
    return y · y
else
    y = Power(x, (n-1) /2)
    return x · y · y
```



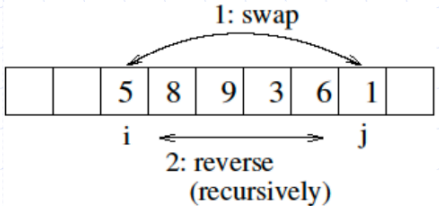
Tail Recursion

Tail recursion (is a special type of linear recursion) occurs when a linearly recursive method makes its recursive call as its last step (for example array-reversal). These methods can easily be converted to non-recursive methods (to iterative ones which are typically changed by the compiler itself).



The iterative solution:

```
Algorithm IterativeReverseArray(A, i, j ):
    Input:   An array A and nonnegative integer indices i and j
    Output:  The reversal of the elements in A starting at
             index i and ending at j

  while i < j do
    Swap A[i ] and A[ j ]
    i = i + 1
    j = j - 1
  return
```



Binary Recursion

Occurs whenever there are **two recursive calls** for each non-base case.

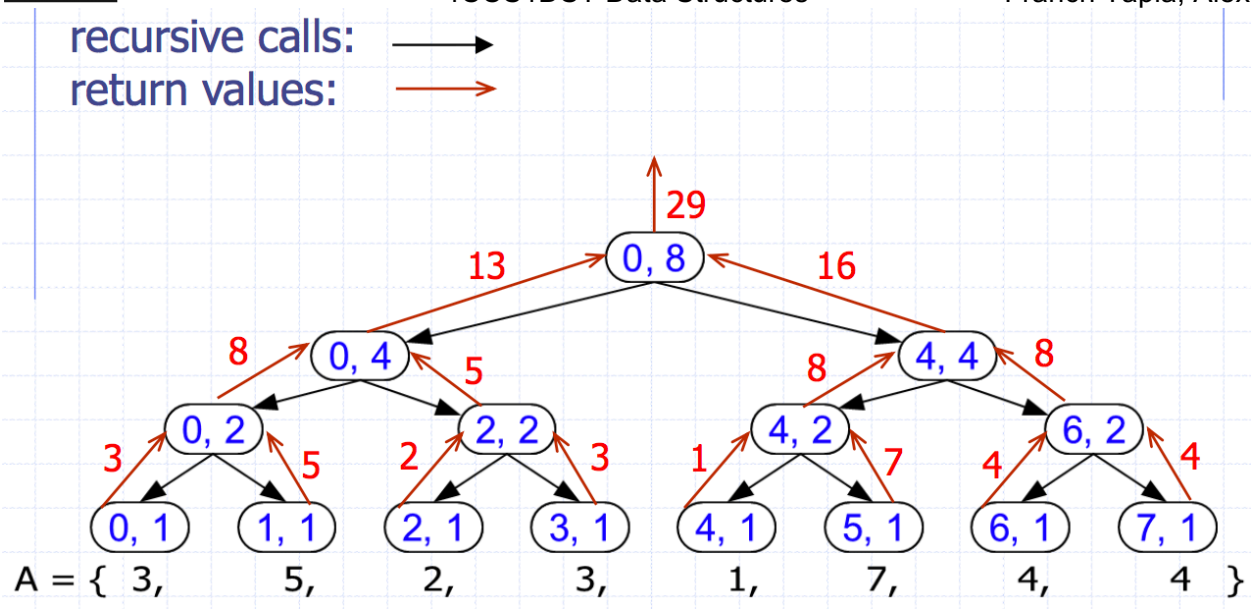
Example 1: when drawing, tick sign the English ruler.

```
public static void drawTicks(int tickLength) { // draw ticks of given length
  if (tickLength > 0) {              // stop when length drops to 0
    drawTicks(tickLength - 1);       // recursively draw left ticks
    drawOneTick(tickLength);         // draw center tick
    drawTicks(tickLength - 1);       // recursively draw right ticks
  }
}
```

two recursive calls

Example 2: add all numbers in an integer array A

```
Algorithm BinarySum(A, i, n):
    Input:   An array A and integers i >= 0 and n >= 1
    Output:  The sum of the n integers in A starting at index i

  if n = 1 then
    return A[i ]
  return BinarySum(A, i, ⌈n/2⌉) + BinarySum(A, i + ⌈n/2⌉, ⌊n/2⌋)
```



Example 3: Computing Fibonacci Numbers

Given the first two Fibonacci Numbers (0,1) we can define the next ones by adding the last two.

Recursively:

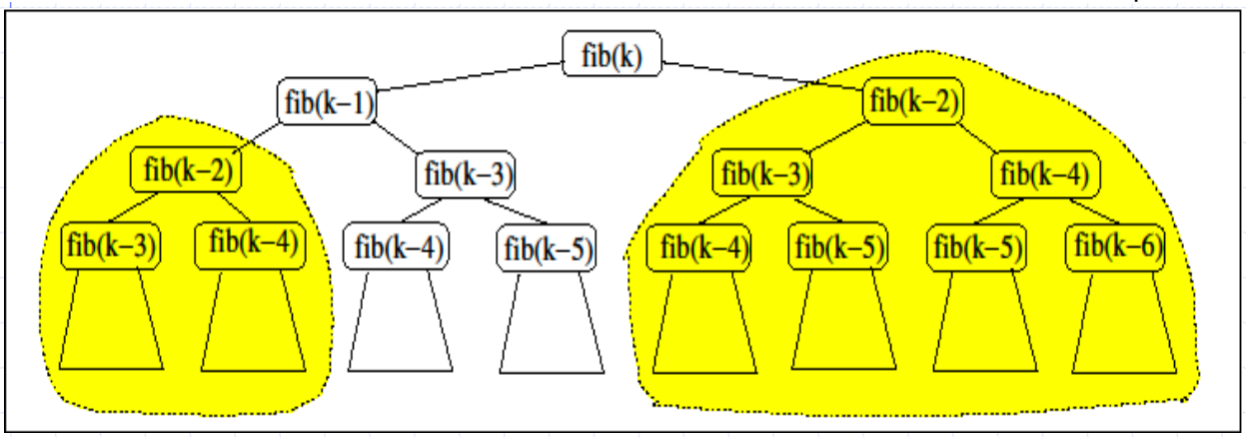
Recursive algorithm (first attempt):

Algorithm BinaryFib(k):
Input: Nonnegative integer k
Output: The k-th Fibonacci number F_k

```
if k <= 1 then
    return k
else
    return BinaryFib(k - 1) + BinaryFib(k - 2)
```

F_4
F_5
F_6
F_7

This is a possibility, nevertheless it is a very inefficient one. This is because by using this method we are repeating computations of certain elements of the tree various times, which means that we are wasting time and power. Look at the recursion tree for BinaryFib(k).



As you can see we are doing fib(k-2) twice.

This shows that sometimes although we can build a solution with a kind of recursion we have to make sure it's the most efficient. We can create another linear algorithm that will also solve the problem. We can find two Fibonacci numbers and add them to find the next.

- Use linear recursion instead

Algorithm LinearFib(k):
Input: An integer $k \geq 1$
Output: Pair of Fibonacci numbers (F_k, F_{k-1})

```
if k <= 1 then
    return ( k, 0 )
else
    ( i, j ) = LinearFib(k – 1)
    return ( i + j, i )
```

F_1, F_0 (if k = 1), F_{k-1}, F_{k-2}, (F_k, F_{k-1})

- LinearFib(k) makes k–1 recursive calls

(BinaryFib(k) makes at least $2^{k/2}$ recursive calls)

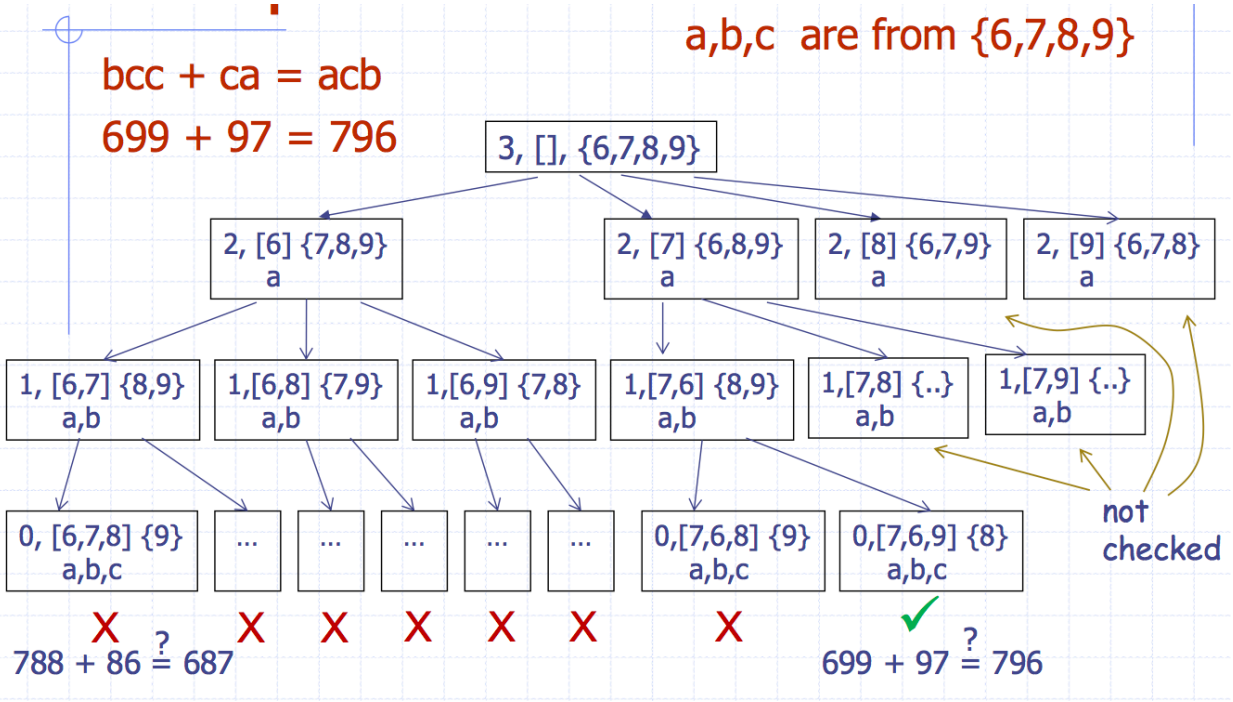
We can compare the time for an algorithm to execute by using System.currentTimeMillis() as we have seen before.

```
> java  FibonacciTest  42
fib(42) = 267914296 [computed iteratively in 0 ms]
fib(42) = 267914296 [computed by linear recursion in 0 ms]
fib(42) = 267914296 [computed by binary recursion in 2785 ms]
>
```

Note that if they were all to be 0, we could run the algorithm many times. Also, it gives better perspective of which algorithm is more efficient.

```
Run iterFib, linearFib 1,000,000 times and divide time by 1,000,000:

fib(42) = 267914296 [computed iteratively in 1.14E-4 ms]
fib(42) = 267914296 [computed by linear recursion in 2.96E-4 ms]
fib(42) = 267914296 [computed by binary recursion in 2818 ms]
```

We can also have a multiple recursion, in which recursion occurs many times, for instance when solving a puzzle with letters and numbers.



Inheritance

Inheritance: a way in which we can reuse code from another class.

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Abstract Classes

Abstract method: a method which has no definition (without its body)

Abstract class: a class which may have abstract methods.

- Abstract classes are used only as super classes in inheritance hierarchies.
- These classes cannot be to instantiate objects as they are not complete.
- Subclasses must override the methods which are empty.

Interfaces

An interface is an ultimate abstract class:

- It was only have public abstract methods.
- [Java 8] can have static final fields (constants), static methods and default methods.
- A class implements the interface if it provides definitions for all the abteact methods in the interface.

Data Structures

Data structure: a systematics way of organizing, accessing and updating data (maintained in the computer memory during the execution of a broader computing task).

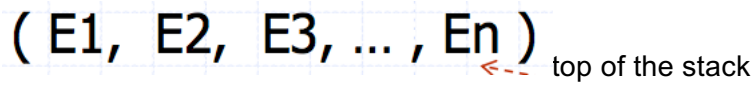
Algorithm: a step-by-step procedure for performing some tasks.

Abstract data type (ADT): a model of data structure that specifies the type data stored, the operations supported on them and the parameters of the operations.

- In Java, it can be expressed by an interface.
- It can also be a concrete class.

Stack Data Structure

Stack: a sequence of elements with one end designated as the top of the stack:

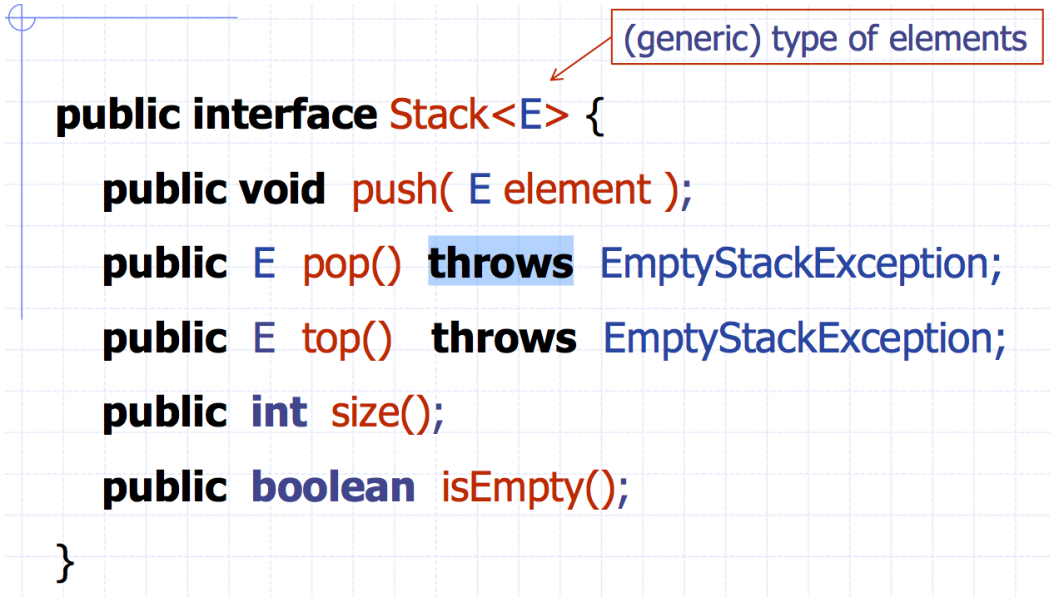


Main stack operations:

- `push(e)` - inserts element e at the top of the stack.
- `pop()` – removes and returns the element at the top of the stack.

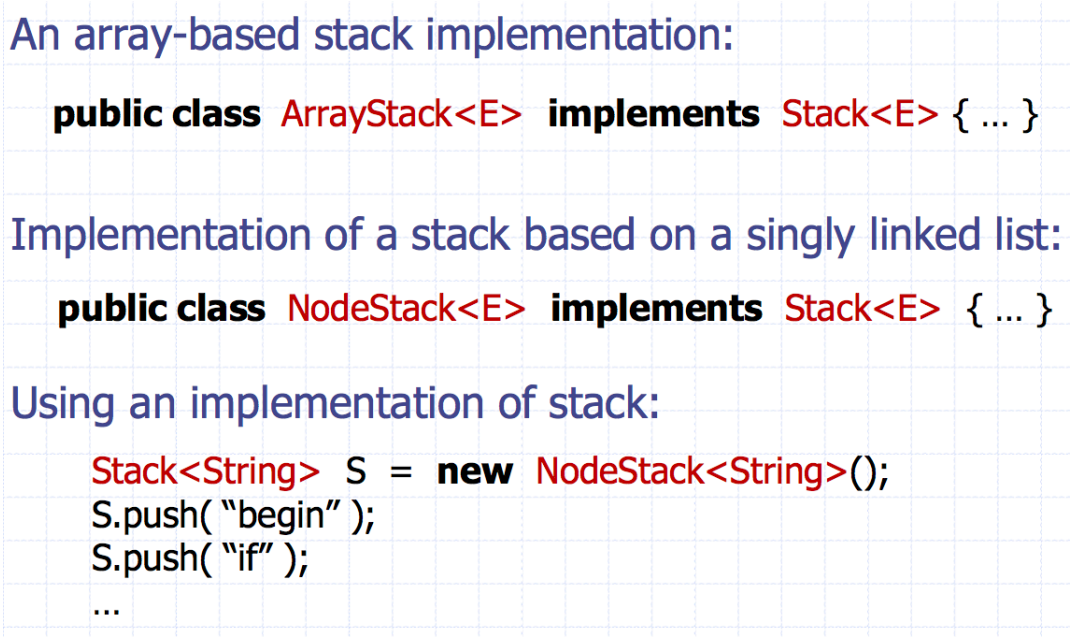
Additional:

- `top()` – returns top element of the stack.
- `size()` – returns the number of elements stored.
- `isEmpty()` – indicates if stack is empty.



Throws refers to exceptions, it returns something else depending on what the exception is.

Implementations of the stack interface



Linked lists

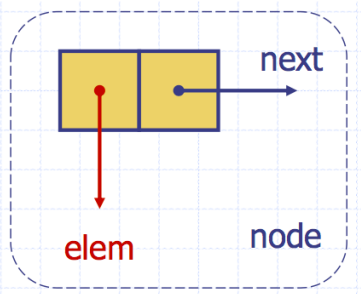
Linked list: an alternative to array for storing a sequence of objects.

Singly Linked List

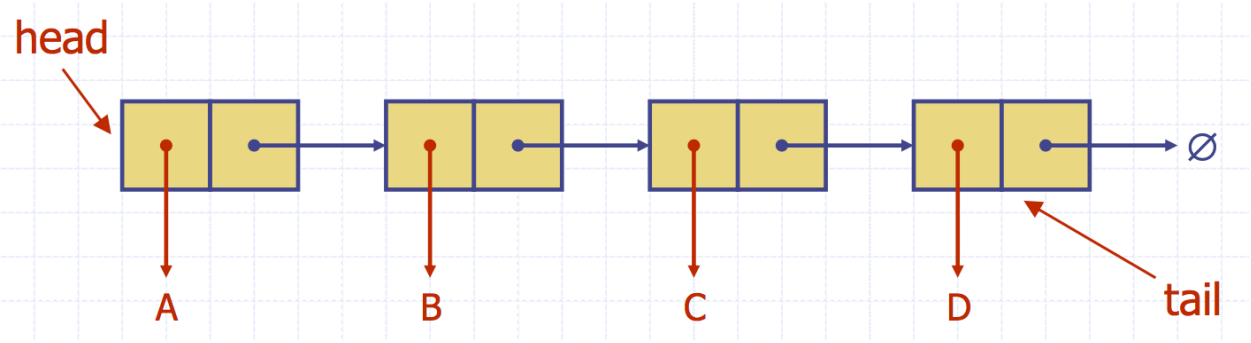
A singly list is a sequence of **nodes**.

A **node** stores an:

- Element
- Link to the next node (doubly linked lists will have info about where the element before is).



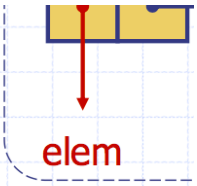
The nodes are spread out in memory, not one to another.



We are storing elements A,B,C,D.

We have to know where the first element, then we can always go to the next one.

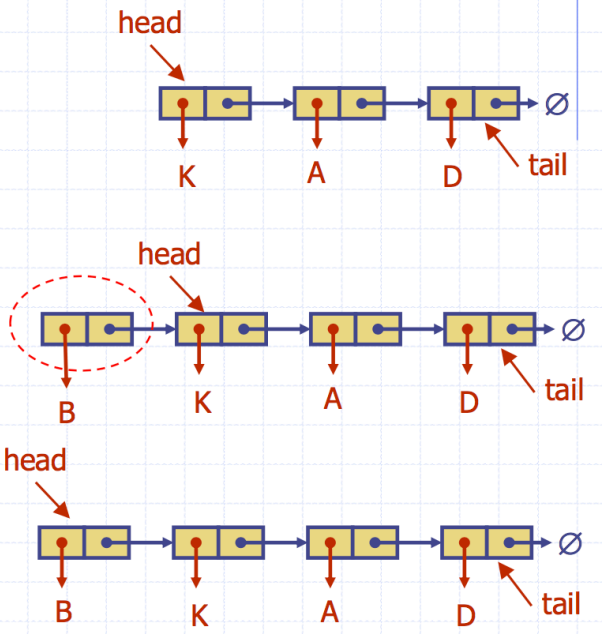
Linked lists can be good because we can **easily implement a new node at the beginning**, simply create a new node and make it point towards your current first one.




```
public class  Node<E> {
   // instance variables:
   private  E  element;
   private  Node<E>  next;

   // creates a node with the given element and next node
   public  Node( E  e,  Node<E>  n )  {  element  =  e;   next  =  n; }

   // creates a node with null references to its element and next node
   public  Node()   {  this(null,  null); }

   // accessor methods:
   public  E  getElement()  {   return  element;  }
   public  Node<E>  getNext()  {  return  next;  }

   // modifier methods:
   public void  setElement( E  newElem )   {  element  =  newElem; }
   public void  setNext( Node<E>  newNext )  {  next  =  newNext;  }
}
```

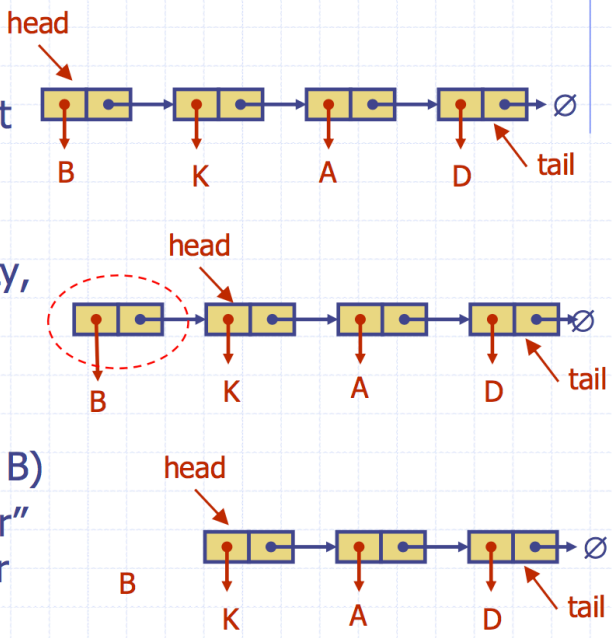
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node
5. Update "size", if maintained
6. If inserting to empty list, update tail, if maintained



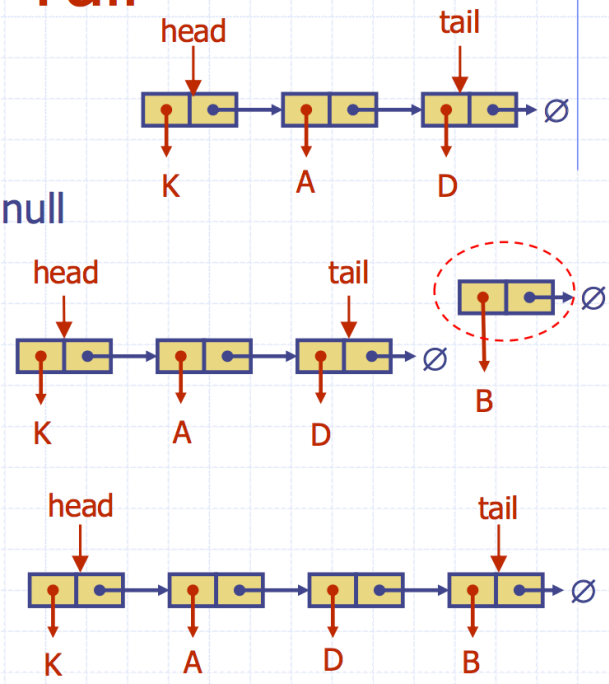
Removing at the Head

1. Update head to point to next node in the list
2. Update "size", if maintained
3. If the list is now empty, update tail, if maintained
4. Return the removed element (here, object B)
5. The "garbage collector" will reclaim the former first node



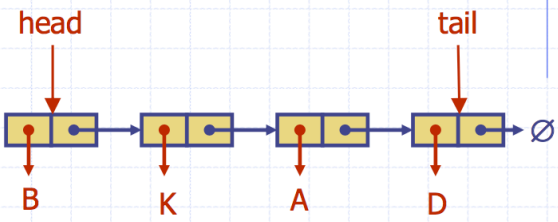
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node (if the list wasn't empty)
5. Update tail to point to new node
6. If inserting to empty list, update head
7. Update "size", if maintained



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



Analysis of Algorithms

Running Time

The running time of an algorithm tends to increase as the input increases. Analysis of algorithms try to understand this dependence and how it affects computation.

We focus on the **worst-case scenario running time.**

Experimental Studies

By running an algorithm (quite extensively) we can use a variety of tools (System.currentTimeMillis()), then we plot our results and compare it to other algorithms.

Theoretical Analysis

- Uses high level description of the algorithm instead of an implementation.
- Takes into account all possible inputs.
- Allows us to evaluate the speed of an algorithm independent from the hardware or software.
- Characterizes running time of an algorithm as a function of the input size, n.

> For a given algorithm, determine a function ***f(n)*** that characterizes the (worst-case) running time of this algorithm as a function of the input size ***n***.

Pseudocode Details

- **Control flow**
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - **Indentation replaces braces**
- **Method declaration**

  Algorithm *methodName* (*arg* [, *arg*…])

  Input …(explain the arguments)

  Output …(explained the return values)
- **Method call**

  methodN (*arg* [, *arg*…])

  var.methodN (*arg* [, *arg*…])
- **Return value**

  return *expression*
- **Expressions**

  ← Assignment (like = in Java)

  = Equality testing (like == in Java)

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

- **CPU**
- **Memory:** a potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character. (Memory cells are numbered)
- **Unit time**
  - Each CPU operation takes unit time
  - Accessing any cell in memory takes unit time

Primitive operations executed from our algorithm

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model (constant number of time units)

- Examples:
  - Assigning a value to a variable
  - Comparing two numbers
  - Evaluating an expression
  - Indexing into an array
  - Following an object reference
  - Calling a method
  - Returning from a method

Counting Primitive Operations

We have to try to find the formula f(n).

```
Algorithm arrayMax(A, n)                    # operations
    currentMax ← A[0]                           2
    for i ← 1 to n − 1 do                       2n + 1
        if A[i] > currentMax then               2(n − 1)
            currentMax ← A[i]                   2(n − 1)
        { increment counter i }                 2(n − 1)
    return currentMax                           1
                                        Total   8n − 2
```

array indexing + assignment

initialize i plus n x (subtract +compare)

Estimating Running Time

Given our formula.

We define:

a = time taken by the fastest primitive operation.

b = time taken for the slowest primitive operation.

At least this, at most this.

- Let $T(n)$ be worst-case time of *arrayMax*. Then

$$(8a)\,n - 2a = a(8n-2) \le T(n) \le b(8n-2) = (8b)\,n - 2b$$

Growth Rate

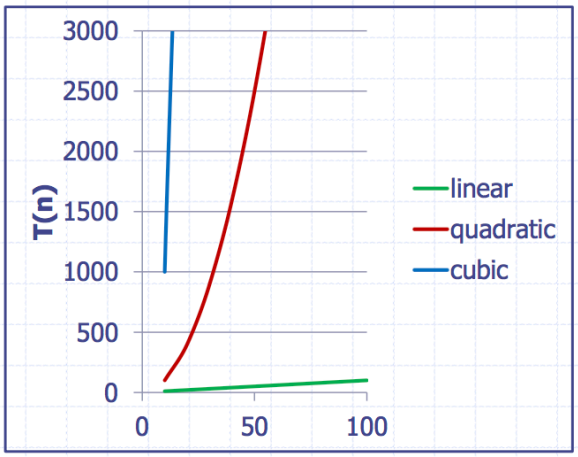
- Is independent of hardware or implementation as it's a property of the algorithm.

if runtime (time for n) is...	time for n + 1	time for 2 n	time for 4 n
$c \log_2 n$	$c \log_2 (n + 1)$	$(c \log_2 n) + c$	$(c \log_2 n) + 2c$
$c\,n$	$c(n+1) = c\,n + c$	**$2 * c\,n$**	$4\,c\,n$
$c\,n \log n$	$\sim c\,n \log n + c \log n$	$2c\,n \log n + 2cn$	$4c\,n \log n + 8cn$
$c\,n^2$	$\sim c\,n^2 + 2c\,n$	**$4 * c\,n^2$**	$16\,c\,n^2$
$c\,n^3$	$\sim c\,n^3 + 3c\,n^2$	$8\,c\,n^3$	$64\,c\,n^3$
$c\,2^n$	$c\,2^{n+1} = 2(c\,2^n)$	$c2^{2n} = 2^n(c2^n)$	$c2^{4n} = 2^{3n}(c2^n)$

Linear runtime: doubles when problem size doubles

Quadratic runtime: quadruples when problem size doubles

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



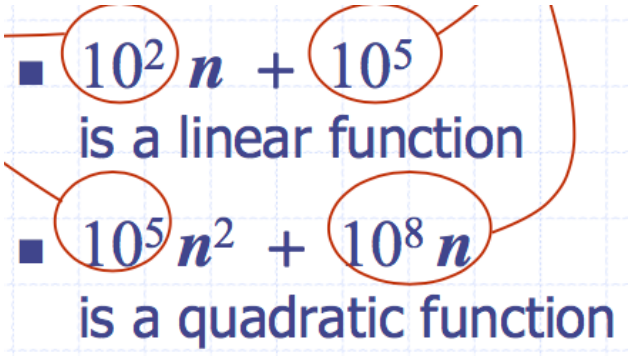
n	linear	quadratic	cubic
1,000	1,000 µs's	1,000,000 ms's	1,000,000,000 sec's

Additionally, constant computation which are those that are not affected by the size of input.

Note that constant factors do not affect the running times. For example:

In the example 10^5 is just a constant number.

Additionally, lower-order terms are not that important since when inputting very large inputs, this becomes redundant (much smaller).



Big-Oh Notation

A comparison of the growth of two functions.

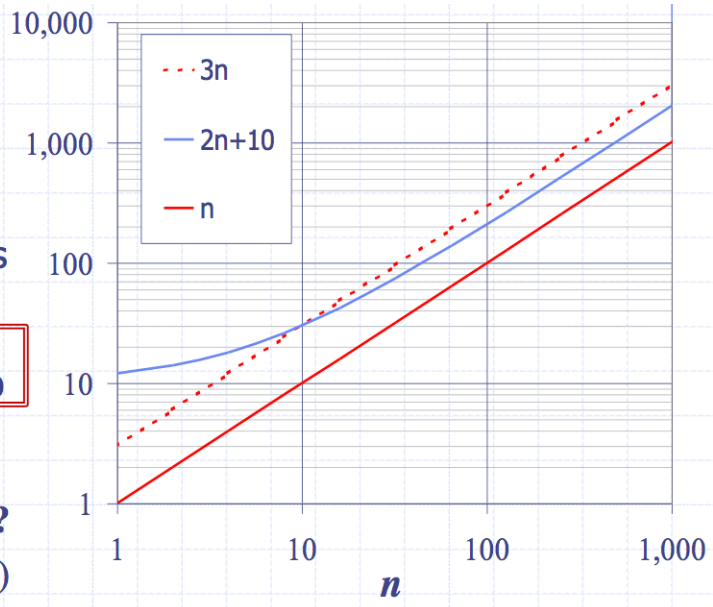
- Given functions $f(n)$ and $g(n)$, we say that

  $f(n)$ is $O(g(n))$

  if there are positive constants c and n_0 such that

  $f(n) \leq cg(n)$ for $n \geq n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$, for all $n \geq n_0$?
  - $(c - 2)\, n \geq 10$ (need $c > 2$)
  - $n \geq 10/(c - 2)$ for all $n \geq n_0$?
  - Yes, pick $c = 3$ and $n_0 = 10$:



$2n + 10 \leq 3n$, for all $n \geq 10$

The rate of function of f(n) is not greater than the rate of growth of function of g(n).

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Rules

- If f(n) is a polynomial of order d, then f(n) = $O(n^d)$
- Drop the lower-order terms.
- We always try to give the smallest possible class of function.

- Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- We use the simplest expression of the class.

- Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

- We can also write equality sign but it is read as is.

Big-Oh Examples

- $7n - 2$
  - $7n-2$ is $O(n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq cn$, for $n \geq n_0$
  - this is true for $c = 7$ and $n_0 = 1$
- $3n^3 + 20n^2 + 5$
  - $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$, for $n \geq n_0$
  - this is true for $c = 4$ and $n_0 = 21$
- $3 \log n + 5$
  - $3 \log n + 5$ is $O(\log n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$, for $n \geq n_0$
  - this is true for $c = 8$ and $n_0 = 2$

In all examples, other values of c and n_0 could also work!

Asymptotic Algorithm Analysis

This determines the running time in big-Oh notation.

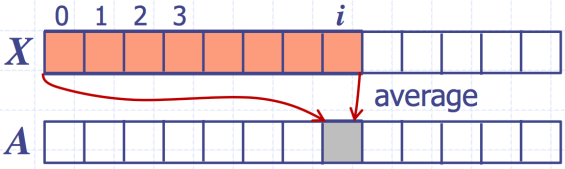
To do Asymptotic Algorithm Analysis:

- We find the worst-case number of primitive operations executed as a function input size.
- We don't need this function exactly, we only want to express it using big-Oh notation.

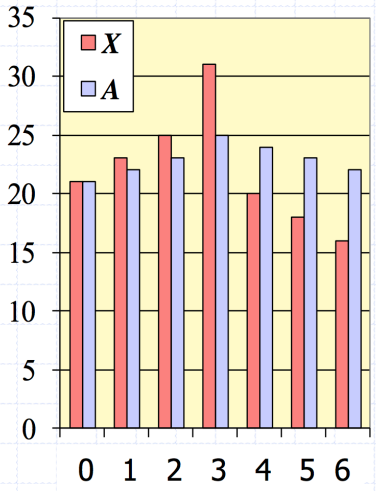
Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages.
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$



- Computing the array A of prefix averages of another array X has applications in financial analysis.



- The following algorithm computes prefix averages in quadratic time by directly applying the definition

Algorithm *prefixAverages1*(X, n)

	#operations
Input array X of n integers	
Output array A of prefix averages of X	
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	$n + 1$
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (i+1) + \ldots + n$
$s \leftarrow s + X[j]$	$0 + 1 + \ldots + i + \ldots + (n-1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

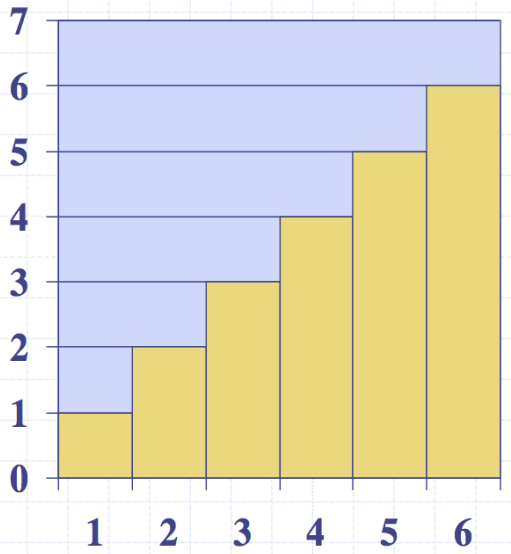
- The running time of ***prefixAverages1*** is

$$2\cdot(1+2+\ldots+n)+3n+2$$

$$= O(1+2+\ldots+n)$$

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

$O(n(n+1)/2) = O(n^2)$ time



Big-Oh and Relatives

Big-Oh

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

Big-Omega

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

Big-Theta

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

f(n) is O(g(n))
and
f(n) is Ω(g(n))

- **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

  $f(n) \geq c \cdot g(n)$ 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

  $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$ $\Leftrightarrow$ $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

- **$5n^2 + 10$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

  Let $c = 1$ and $n_0 = 1$

- **$5n^2 + 10$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

  Let $c = 1$ and $n_0 = 1$

- **$5n^2 + 10$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$, if it is $\Omega(g(n))$ and $O(g(n))$.

  We already know that $5n^2 + 10$ is $\Omega(n^2)$. Show that $5n^2 + 10$ is $O(n^2)$.

  $f(n)$ is $O(g(n))$, if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

  Let $c = 6$ and $n_0 = 4$

Big Oh – upper bound running time

Big Omega – lower bound running time

Big Theta – best describes the rate of growth of the function. Tight bound.

Stacks and Queues

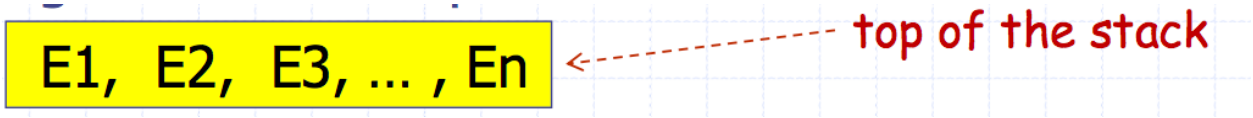
Abstract Data Types (ADTs)

And ATD specifies:

- Type of data stored
- Operations on the data
- Error conditions associated with operations *(although it does not say how operations are implemented)*

The Stack ADT

An instance of the stack data structure is a sequence of elements with one end designated as the top of the stack:



Main Operations:

- `push(e)` insert element e at the top of the stack
- `pop()` remove and return the element at the top of the stack
- `isEmpty()` check if stack is empty

Additional stack operations:

- `top()` return the top element in the stack (without removing)
- `size()` return the number of elements stored

Note that an error will occur for both `pop()` and `top()` if stack is empty.

```
public interface Stack<E> {
  public  void  push(E element);
  public  E  pop()
       throws  EmptyStackException;
  public E top()
       throws  EmptyStackException;
  public  int  size();
  public boolean  isEmpty();
}
```

Note that this different from the built-in Java class java.util.Stack<E>

Exceptions

Attempting the execution of an operation of an ADT may sometimes cause an error condition. In java we use the keyword throw which returns something which is not what we normally do.

```
Stack<Integer> s  =  new  ArrayStack<Integer>();
...
```

```
System.out.println( "top of stack: " + s.top() );   // not safe!
```

```
if ( !s.isEmpty() ) {  System.out.println("top of stack: " + s.top() ); }
else {  System.out.println("stack is empty"); }
```

```
try {  System.out.println("top of stack: " + s.top() ); }
catch (EmptyStackException ex) {System.out.println("empty stack"); }
```

With the keyword catch we trap the thrown exception we make our program to something rational.

Applications

Direct applications:

- Page-visited history in a Web Browser (want to go to previous)
- Undo sequence in a text editor
- Chain of method calls in JVM (go from a method to another and then go back)

Indirect applications:

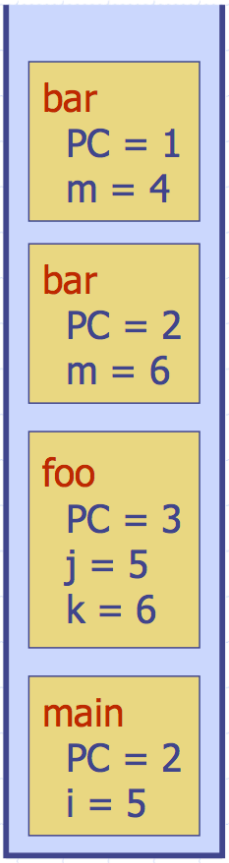
- Auxiliary data structures for algorithms
- Components of other data structures

Method Stack in JVM

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM **push**es on the stack a "frame" containing
  - local variables
  - program counter (PC), keeping track of the current instruction
- When a method ends, its frame is **pop**ped from the stack and control is passed to the method now on top of the stack
- Method stack supports recursion

```
   main() {
1:    int i = 5;
2:    foo(i);
   }

   foo(int j) {
1:    int k;
2:    k = j+1;
3:    bar(k);
4:    …
   }

   bar(int m) {
1:    if  (m > 5) {
2:       bar(m-2);
      }
3:    …
   }
```



 8

Array-based Stack implementation

- We add elements from left to right
- A variable keeps track of the index of the top element.

```
Algorithm size()
  return t + 1

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

The array storing the stack elements may become full, a push operation will then throw `FullStackException`

```
Algorithm push(o)
  if  t = S.length – 1  then
      throw FullStackException
  else
      t ← t + 1
      S[t] ← o
```

Performance and Limitations
Performance

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

Limitations:

- The maximum size of the stack must be defined priori and cannot be changed
- Trying to push a new element into a full stack causes and implementation-specific exception.

```
public class ArrayStack<E>
        implements Stack<E> {

  // S[ ] holds stack elements
  protected E S[ ];

  // index to top element
  protected int top = –1;

  // constructor
  public ArrayStack(int cap) {
    S = (E[ ]) new Object[cap]);
  }
```

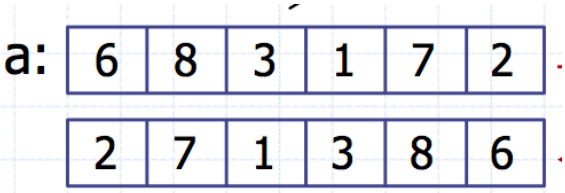
"new E[cap]" – not allowed because E is not a concrete type


```
public E pop()
    throws EmptyStackException {

  if isEmpty()
    throw new
        EmptyStackException
        ("Empty stack: cannot pop");

  E element = S[top];
  top – –;
  S[top] = null;
            // for garbage collection
  return element;
}

… (other Stack methods)
```

Note should be S[top+1] double check.

Example use in Java

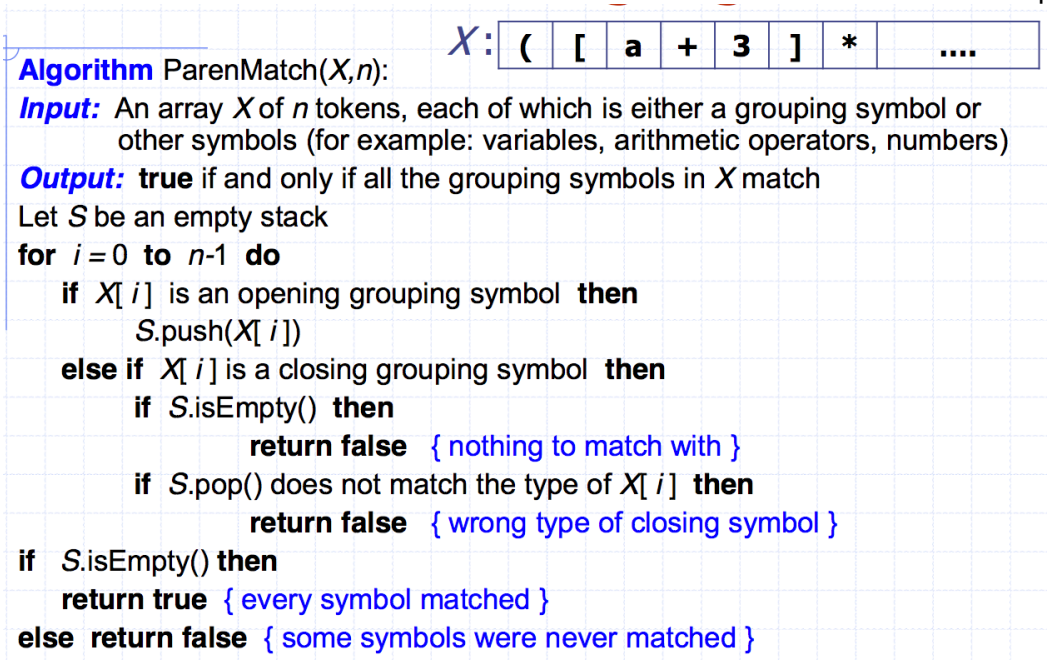
Reverse elements in array

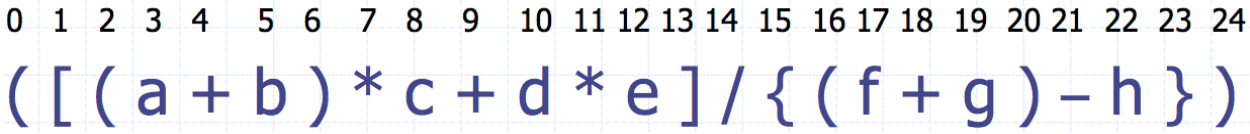



```
public class StackTester {
    public static <E> void reverse(E[ ] a) {
        Stack<E> S = new ArrayStack<E>(a.length);
        for (int i=0; i < a.length; i++) { S.push(a[i]); }
        for (int i=0; i < a.length; i++) { a[i] = S.pop(); }
    }
    public static void main(String[ ] args) {
        String[ ] s = { "Jack", "Kate", "Hurley", "Jin", "Boone" };
        System.out.println( "s = " + Arrays.toString(s) );
        reverse(s);
        System.out.println("s = " + Arrays.toString(s));
    }
}
```

Parenthesis Matching

Each (, { or [must be paired with), } or [



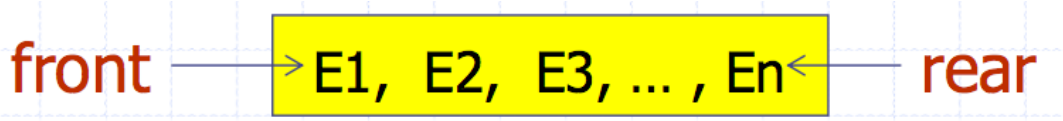


In this case, the stack will be empty at the end, hence our computation has worked correctly. If for some reason the matching wasn't successful ***REMEMBER that our stack popped the top of the stack which means that our final stack will have one element less.***

Queues

The Queue ADT

- Stores a collection of arbitrary elements.
- Insertions and deletions follow **FIFO First-In First-Out** scheme.
- There is a "queue" insertions are at the rear and removals are at the front.



Main queue operations:

- `enqueue(e)` – inserts elements e at the end of the queue.
- `dequeue()` – removes and returns the element at the front of the queue.
- `front()` – returns the element at the front without removing it.
- `size()` – returns the number of elements stored.
- `isEmpty()` – returns a boolean value indicating whether no elements are stored.

Exceptions:

- Attempting `dequeue` or `front` on an empty queue throws `EmptyQueueException`

Operation	*Output*	*Queue*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()

Applications of Queues

Direct

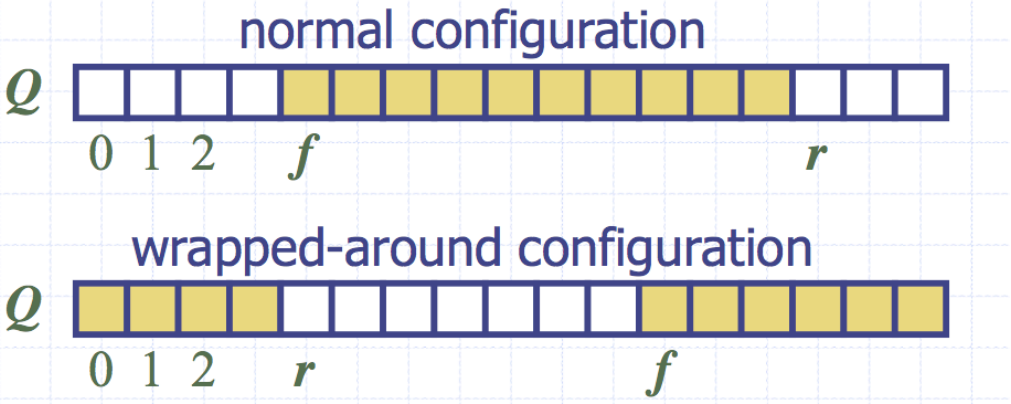
- Access to shared resources (e.g. printer)
- Multiprogramming

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Array-based Queue (circular Queue)

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

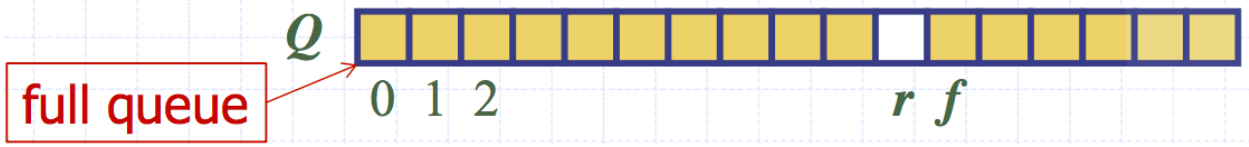


Operations

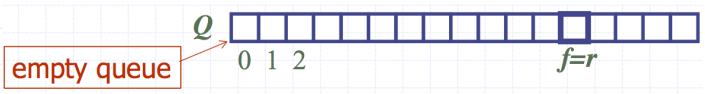
Algorithm *size*()
  return $(r - f + N) \bmod N$

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

Algorithm *enqueue*(*e*)
  if *size*() $= N - 1$ **then**
    throw *FullQueueException*
  else
    $Q[r] \leftarrow e$
    $r \leftarrow (r + 1) \bmod N$



```
Algorithm dequeue()
  if isEmpty() then
     throw EmptyQueueException
  else
     e ← Q[f]
     f ← (f + 1) mod N
     return e
```

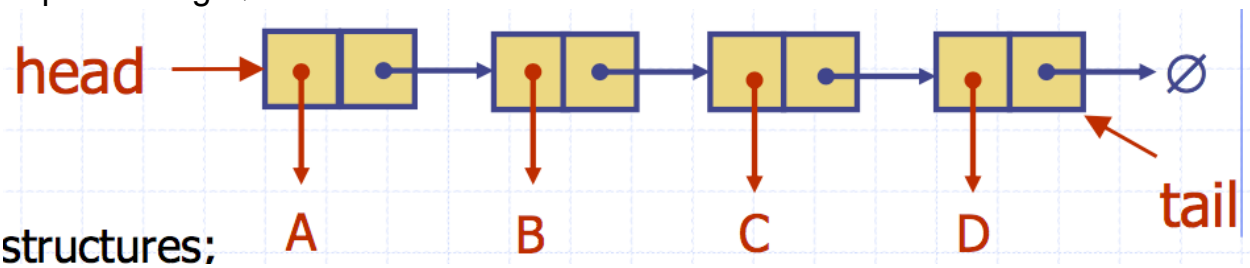


Queue Interface in Java

- Requires a definition of the class `EmptyQueueException`
- Built-in interface: `public interface Queue<E> extends Collection<E>`

```
public interface Queue<E> {
  public int size();
  public boolean isEmpty();
  public E front()
      throws EmptyQueueException;
  public void enqueue(E element);
  public E dequeue()
      throws EmptyQueueException;
}
```

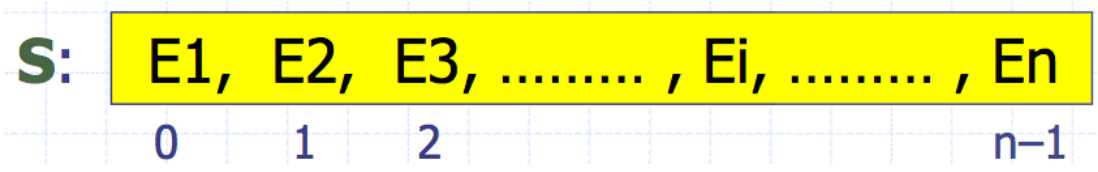
Implementing Queue with a Linked List



```
public class NodeQueue<E> implements Queue<E> {
    protected Node<E> head, tail;          // the head and tail nodes
    protected int size;                    // number of elements in queue

    /** Creates an empty queue. */
    public NodeQueue() { head = null; tail = null; size = 0; }

    public int size() { return size; }     // return the current queue size

    public boolean isEmpty() {             // returns true iff queue is empty
        if ( (head==null) && (tail==null) ) return true;
        return false;
    }

public void enqueue(E elem) {      // same as insertAtTail in SLinkedList
    Node<E> node = new Node<E>(elem, null);      // new tail node
    if ( size == 0 ) head = node;      // special case: empty queue
    else tail.setNext(node);           // add node at the tail of the list
    tail = node;   size++;             // update tail and size
    }
public E dequeue() throws EmptyQueueException {
                                // similar to removeAtHead in SLinkedList
    if ( size == 0 ) throw new EmptyQueueException("Queue is empty.");
    E tmp = head.getElement();
    head = head.getNext();
    size--;
    if (size == 0 ) tail = null;       // the queue is now empty
    return tmp;
 }
public E front() { ... }     public static void main(String[] args) { ... } }
```

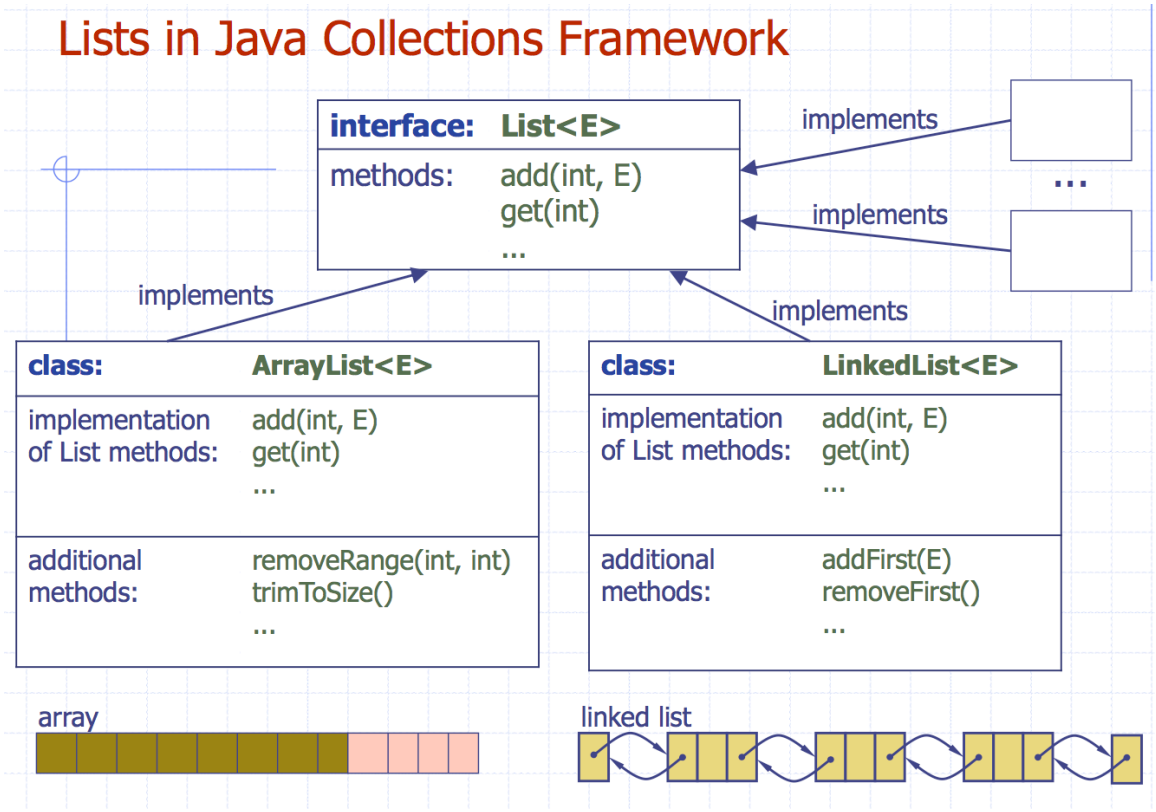
Lists

List: a collection S of elements stored in a certain linear order.

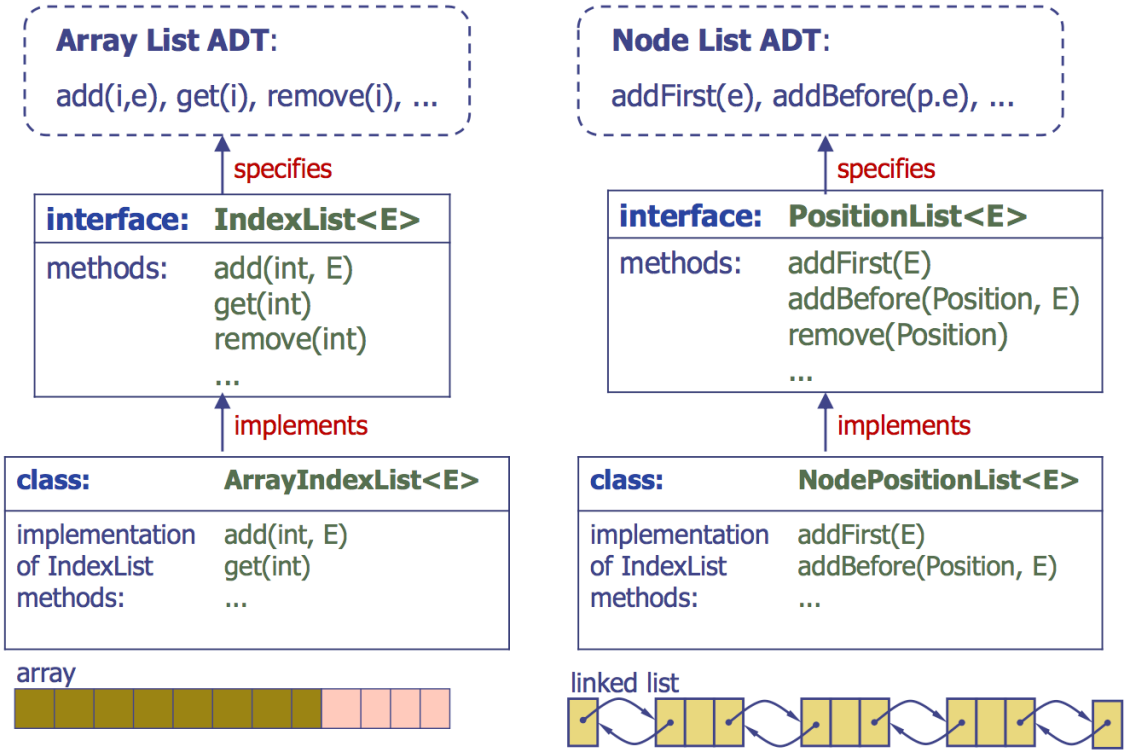


We can refer to these elements as first, second etc.

Index: the index of the element e in S is the number of elements that are before e in S. This index is used to specify where to insert a new element into the list, or where to remove an old element from.



Our Lists: ADTs, interfaces, classes



Array List ADT

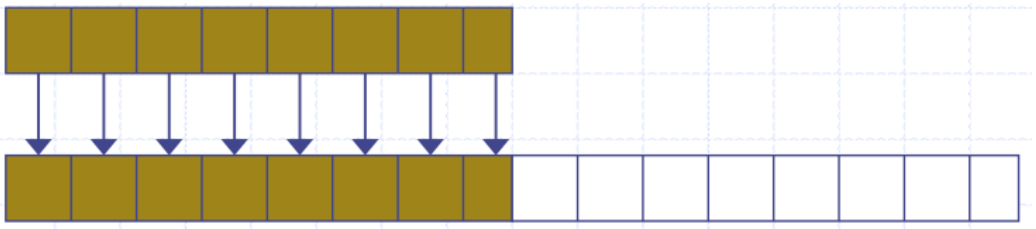
- `size()` – returns the number of elements.
- `isEmpty()` – returns true if its empty.
- `get(i)` – returns the element of S with index i, error occurs if **i > 0** or **i > size() – 1.**
- `set(i,e)` – replace the element at index I and return the old element at this index, an error occurs if **i > 0** or **i > size() – 1.**
- `add(i,e)` – insert a new element e into S to gave index i, error occurs if **i > 0** or **i > size() – 1.**
    - Note that if there is already an element in that array, all the elements are shifted to the right and the new element is added (always checks for error first).
- `remove(i)` – removes from S, and returns the element at index i, error occurs if **i > 0** or **i > size() – 1.**
    - Note that when an element is removed elements are shifted left (always checks for error first).

Operation	*Output*	*S*
add(0,5)	–	(5)
add(0,3)	–	(3, 5)

get(1)	*5*	(3, 5)
add(2,7)	–	(3, 5, 7)
get(3)	*"error"*	(3, 5, 7)
remove(1)	*5*	(3,7)
add(1,6)	–	(3, 6, 7)
add(1,7)	–	(3, 7, 6, 7)

Making full arrays larger

Say we have filled up our array and we want to make it bigger.



To do so we will first create a new, larger array and then copy the elements from the current array into it. It can be difficult to specify the size of the new array since we don't know how many more elements the user wants to store and re-creating another array can be costly for our program.

Start increasing by 1

Say we decide to increase the size of our array by 1 every time it's filled.

Length of full current Array	Length of new Array	Cost	Explanation
1	2	1	Transfer one element over.
2	3	2	Transfer two elements over.
N	N+1	N	Transfer n elements over

This creates a list like so:

1 + 2 + 3 + 4 + 5 … + N, this is a sequence and we can calculate the sum of the list by using the equation:

$$S_n = \frac{n(n+1)}{2}$$

This means that the sum is:

$$S_n = \frac{n^2+n}{2} \approx \frac{n^2}{2} \approx n^2$$

Therefore we can say:

$$\Theta(1+2+3+\cdots+n) = \Theta(n^2)$$

Say instead of adding 1 each time we added another number c; then:

still $\Theta(n^2)$: $c+2c+3c+ \ldots +n = c(1+2+\ldots+n/c) \approx n^2/(2c) = \Theta(n^2)$.

If we double the capacity

- If we double the capacity the running time is Theta(n), for adding elements when there is room in the array.
- Plus the time spent on increasing the array (let's assume the initial capacity of the array is 2)
  Then the cost is the following:
  First will be 4 since we have to transfer 4 elements, second 8 because we have to transfer 8 elements…

$$\Theta(4+8+16+\cdots+n) = \Theta(n^2)$$

Double the capacity, then the running time is

- Θ(n), for adding the elements when there is room in the array,
- plus the time spent on increasing the array,

which is (assuming that the initial capacity of the array is 2):

$\Theta(4 + 8 + 16 + 32 + \ldots + N)$

$= \Theta(2N - 4) = \Theta(N) = \Theta(n)$,

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

where $N = 2^k$ is the final capacity of the array; $n \le N < 2n$.

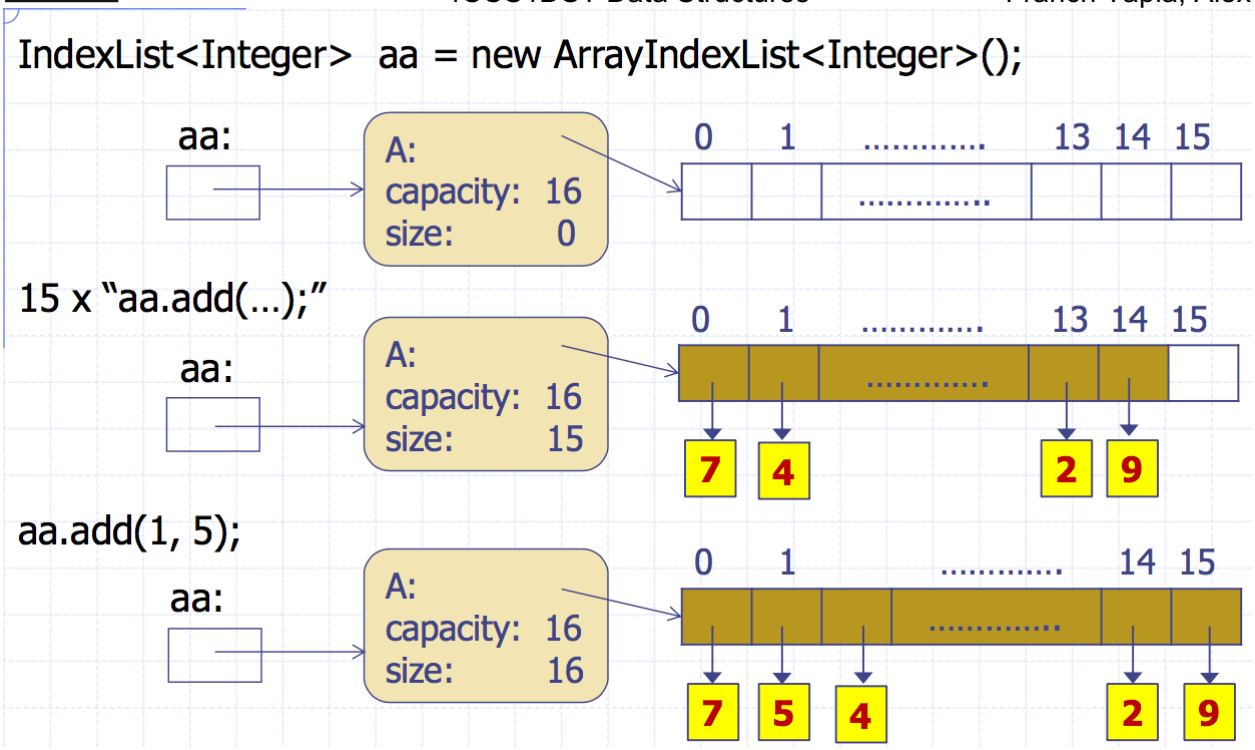
Therefore, when the array is full and the operation "add" is performed, we double the capacity of the array.

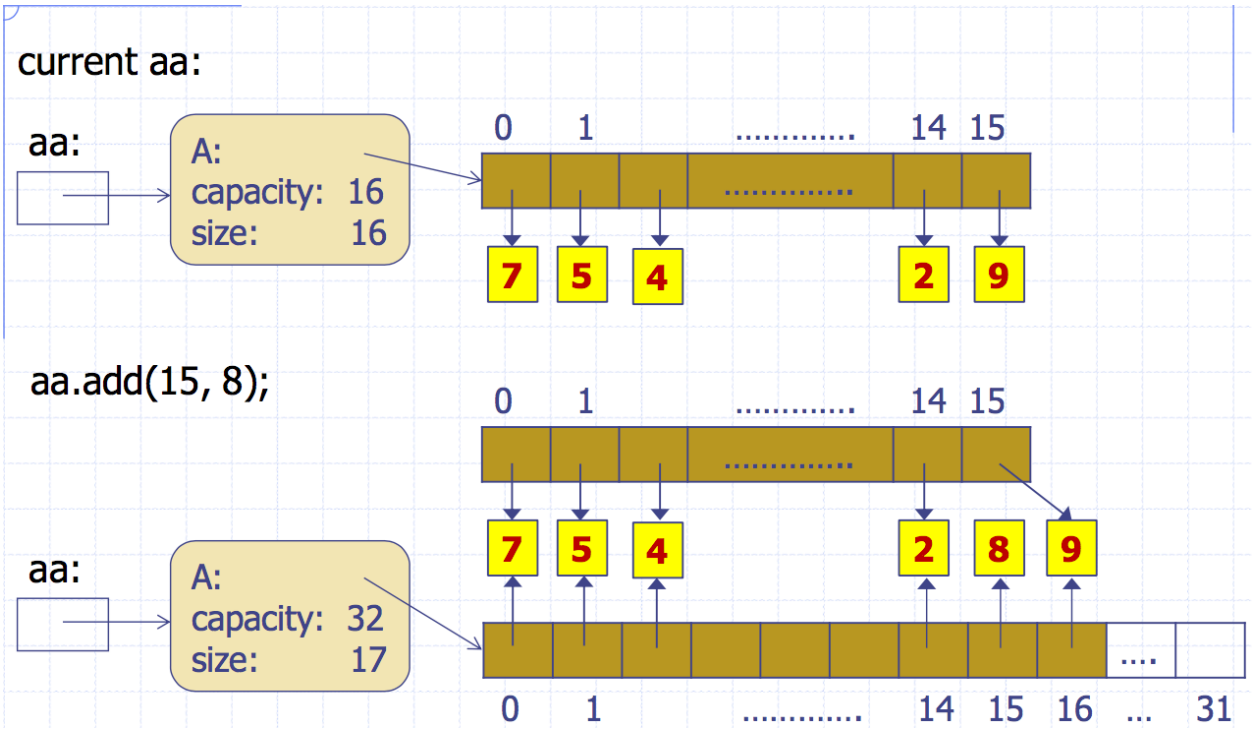
Class ArrayIndexList<E>

```
public class ArrayIndexList<E> implements IndexList<E> {
   private E[ ] A;              // array storing the elements of the indexed list
   private int capacity = 16;   // the current capacity (length) of array A
   private int size = 0;        // number of elements stored in the indexed list
   public ArrayIndexList() {
     A = (E[ ]) new Object[capacity];  // initial array with the default capacity 16
   }
   // implementations of the IndexList methods
   public void add(int r, E e) throws IndexOutOfBoundsException { ... }
   ...
   // an auxiliary method; checks if the given index is in the range [0,n-1]
   protected void checkIndex(int r, int n) throws IndexOutOfBoundsException
     { if ( r < 0 || r >= n ) throw new IndexOutOfBoundsException(" ... "); }
}
```

Method add(int r, E e)

```
/** Inserts an element at the given index. */
 public void add(int r, E e) throws IndexOutOfBoundsException {
    checkIndex(r, size() + 1);
    if ( size == capacity ) {                          // an overflow
        capacity *= 2;
        E[ ] B = (E[ ]) new Object[capacity];          // create a new array
        for (int i=0; i<size; i++) { B[i] = A[i]; }    // copy all elements
        A = B;                                 // the new array becomes the list array
    }
    for (int i=size – 1; i>=r; i--) { A[i+1] = A[i]; } // shift elements up
    A[r] = e;                                          // insert the new element
    size++;                                            // update the size of the list
 }
```

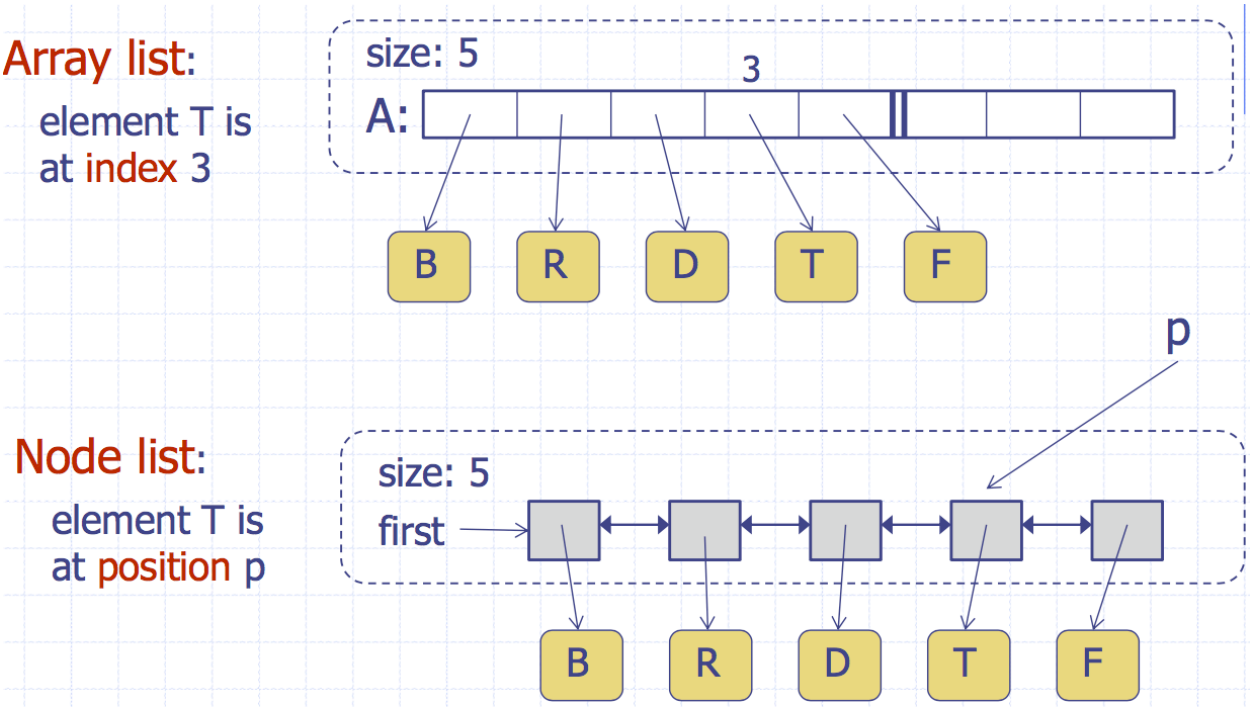

IndexList<Integer> aa = new ArrayIndexList<Integer>();
aa:
A:
capacity: 16
size: 0
0 1 ..... 13 14 15
15 x "aa.add(...);"
aa:
A:
capacity: 16
size: 15
0 1 ..... 13 14 15
7 4 2 9
aa.add(1, 5);
aa:
A:
capacity: 16
size: 16
0 1 ..... 14 15
7 5 4 2 9


current aa:
aa:
A:
capacity: 16
size: 16
0 1 ..... 14 15
7 5 4 2 9
aa.add(15, 8);
aa:
A:
capacity: 32
size: 17
0 1 ..... 14 15
7 5 4 2 8 9
0 1 ..... 14 15 16 ... 31

Performance

- The space used is $O(n)$.
- Operations size, isEmpty and get and set run in $O(1)$ time (constant time).
- Operation remove(i) runs in $O(n)$ time, or, more precisely in $\Theta(n-i)$ time.
- Removing the last element takes $O(1)$ time.
- Operation add(i,e) runs in $O(n)$ time.
- Adding a new element at the end of the list takes $\Theta(n)$ time in the worst case (full array), but only $O(1)$ time on average.

Node List



Position: a place where a single element is stored.

Knowing the position of the element, we might not know its index and if we know the index we might not know the position.

For a node list we will only use position (since it works with links not indexes). To get the element in a position we will use:

`p.element()` – returns the element stored in position p.

The Node List ADT

Accessor methods:

- `first()`
- `last()`
- `prev(p)`
- `next(p)`

Update methods:

- `set(p,e)` – replace the element at p with e, return the old element.
- `remove(p)` – remove and return element at p.
- `addFirst(e)`
- `addLast(e)`
- `addBefore(p,e)`
- `addAfter(p,e)`

NOTE: all the above must have error conditions.

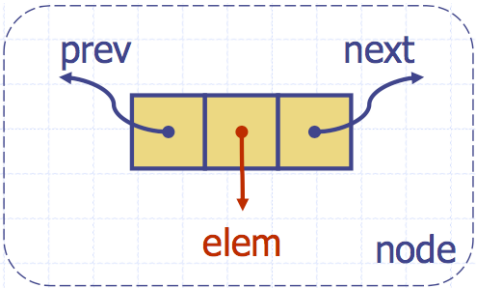
Generic methods:

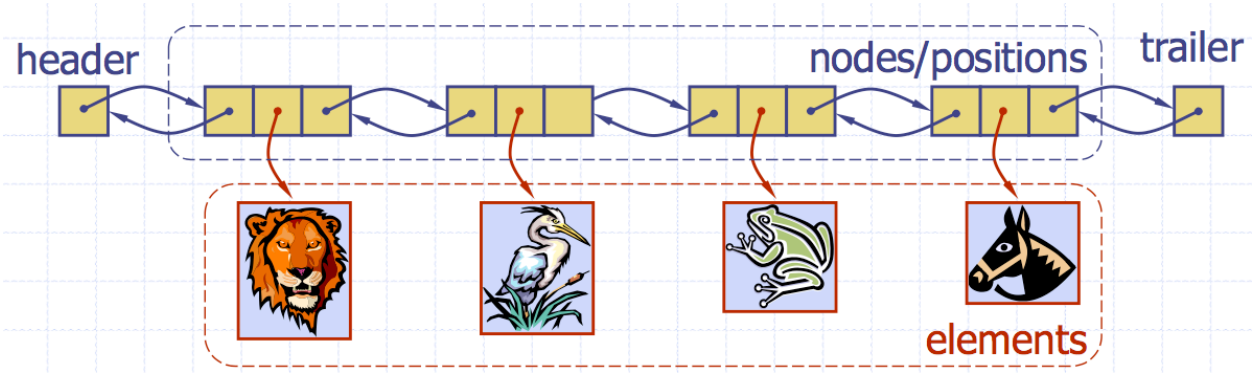
- `size()`
- `isEmpty()`

Doubly Linked list

A doubly linked list works in similar fashion as a singly linked one, it contains:

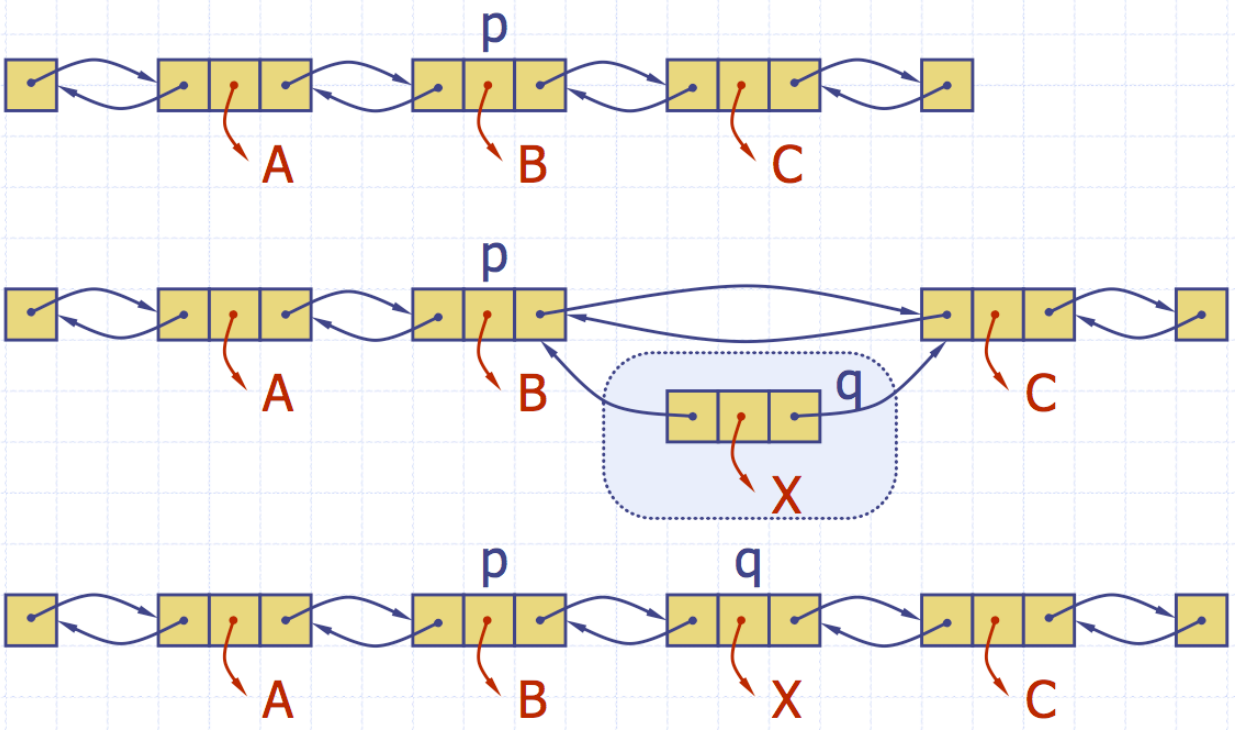
- element
- link to the previous node
- link to the next node





Insertion

We visualize operation addAfter(p, X)



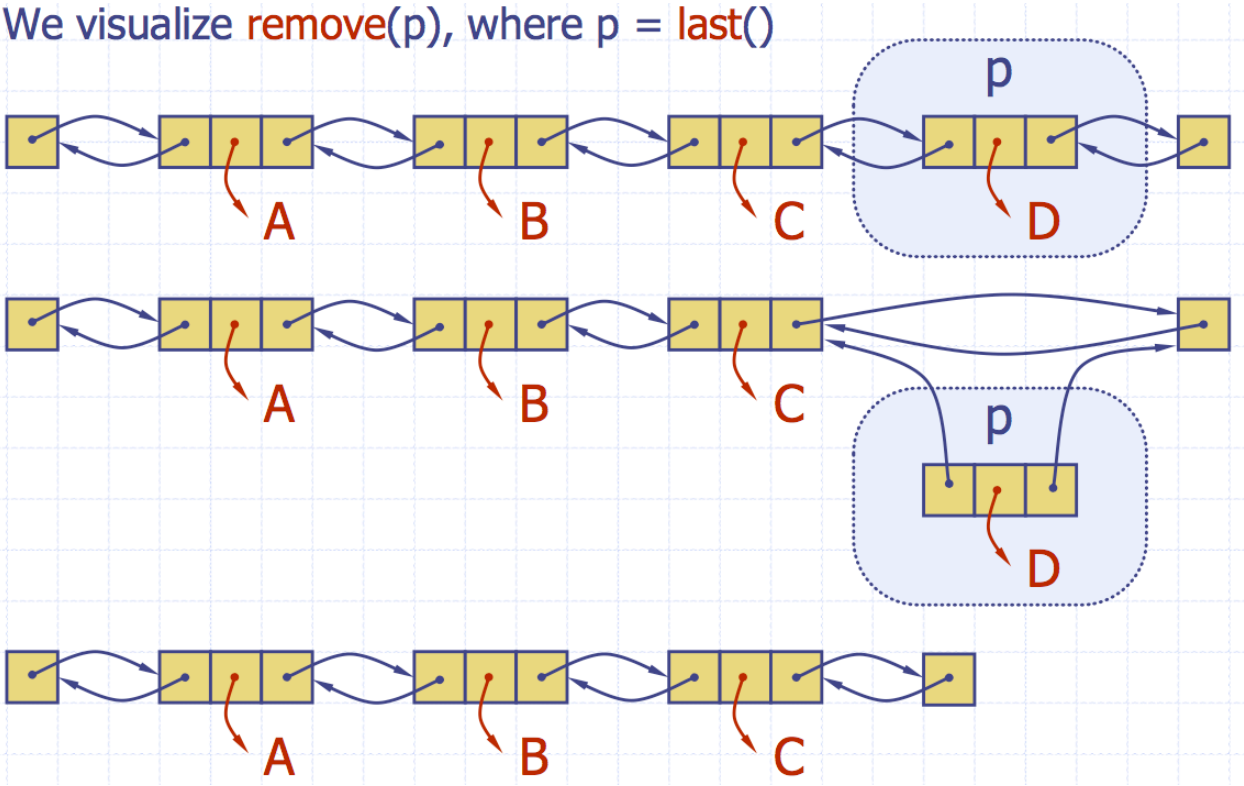
Algorithm addAfter(p,e):

Create a new node v

```
v.setElement(e)              { put e in the new node v }
v.setPrev(p)                 { link v to its predecessor }
v.setNext(p.getNext())       { link v to its successor }

(p.getNext()).setPrev(v)     { link p's old successor to v }
p.setNext(v)                 { link p to its new successor, v}

numberElements++
```

Deletion

We visualize remove(p), where p = last()



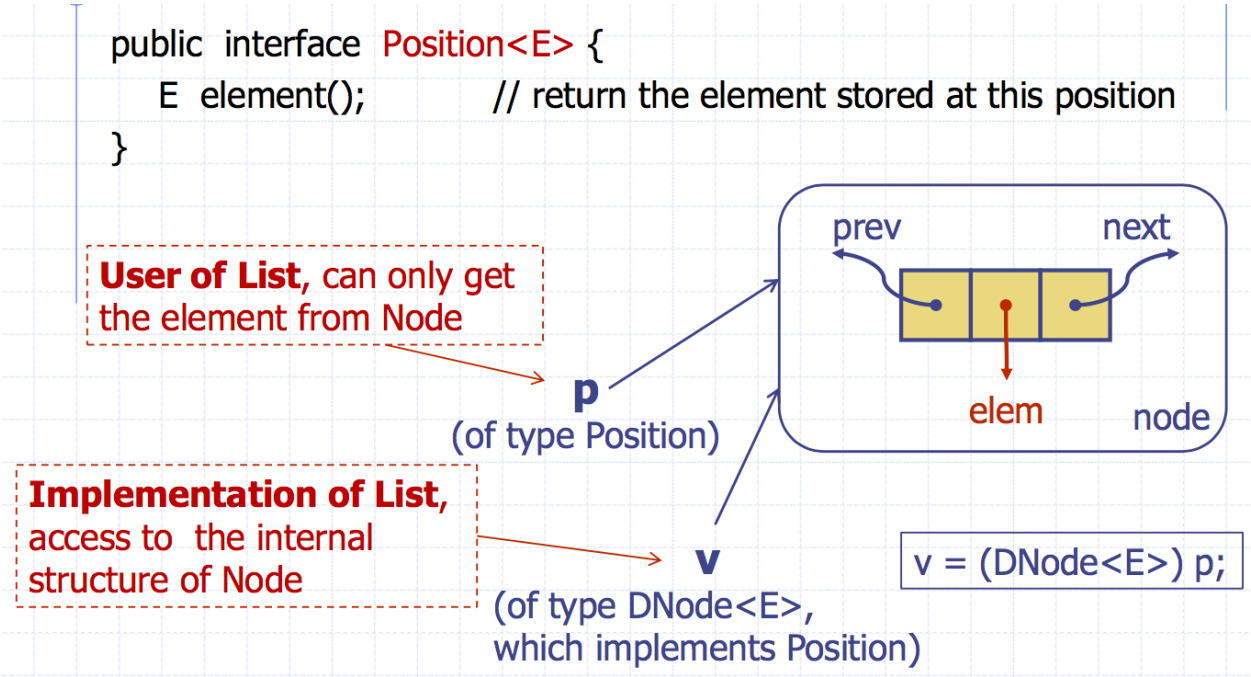
Algorithm remove(p):

```
t ← p.element          { a temporary variable to
                         hold the return value }

(p.getPrev()).setNext(p.getNext())     {linking out p}
(p.getNext()).setPrev(p.getPrev())

p.setPrev(null)        {invalidating the position p}
p.setNext(null)

numberElements++

return t
```

- The space used by each node of the list is $O(1)$
- The space used by a list with n elements is $\Theta(n)$
- All the operations of the Node List ADT run in $O(1)$ time
- The position operation element() runs in $O(1)$ time
- No efficient implementation of operations which refer to the index of an element of a list.

Interface Position<E>

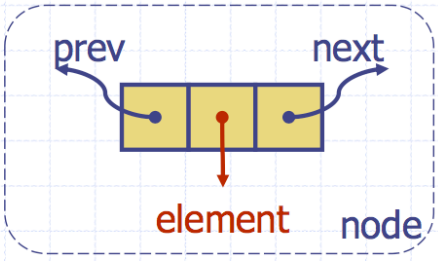
```
public interface Position<E> {
    E element();          // return the element stored at this position
}
```



```
public interface PositionList<E> extends Iterable<E> {
    public int size();
    public boolean isEmpty();
    public Position<E> first() throws EmptyListException;  // returns the first node
    public Position<E> last() throws EmptyListException;   // returns the last node
    public Position<E> next(Position<E> p)
            throws InvalidPositionException, BoundaryViolationException;
    public Position<E> prev(Position<E> p) throws ... ;
    public void addFirst(E e);  // inserts an element at the front of the list
    public void addLast(E e);   // inserts an element at the back of the list
    public void addAfter(Position<E> p, E e) throws InvalidPositionException;
    public void addBefore(Position<E> p, E e) throws InvalidPositionException;
    public E remove(Position<E> p) throws InvalidPositionException;
    public E set(Position<E> p, E e) throws InvalidPositionException;
    // not discussed:
    public Iterable<Position<E>> positions();   public Iterator<E> iterator();   }
```

Implementation of a Node

```
public class DNode<E> implements Position<E> {
    private DNode<E> prev, next;     // references to the nodes before and after
    private E element;               // element stored in this position

    // constructor
    public Dnode (DNode<E> newPrev, DNode<E> newNext, E elem) {
        prev = newPrev;
        next = newNext;
        element = elem;
    }

    // method from interface Position
    public E element() throws InvalidPositionException {
        if ( (prev == null ) && ( next == null ) )
                throw new InvalidPositionException("Position is not in a list!");
        return element;
```

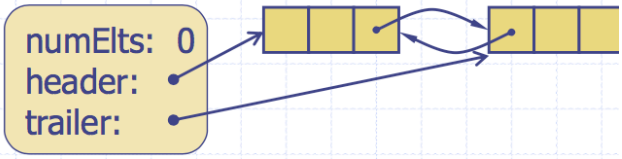


```
    // accessor methods
    public DNode<E> getNext() { return next; }
    public DNode<E> getPrev() { return prev; }

    // update methods
    public void setNext(DNode<E> newNext) { next = newNext; }
    public void setPrev(DNode<E> newPrev) { prev = newPrev; }
    public void setElement(E newElement) { element = newElement; }
}
```

Class NodePositionList<E>

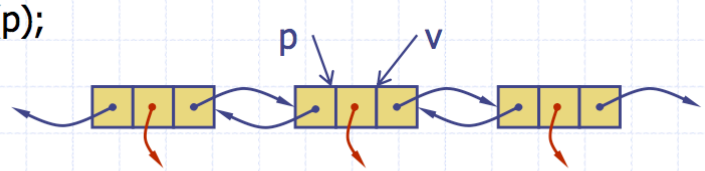
Class NodePositionList<E> (1)

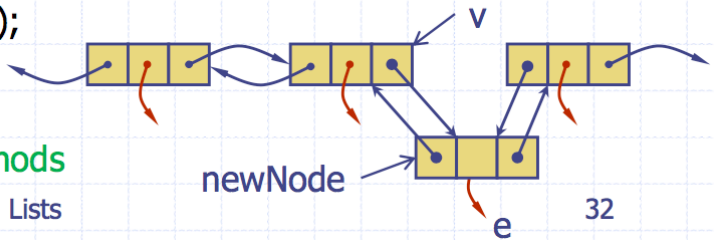
```
public class NodePositionList<E> implements PositionList<E> {
    protected int numElts;                  // number of elements in the list
    protected DNode<E> header, trailer;     // special sentinels

    // constructor
    // that creates an empty list
    public NodePositionList() {
        numElts = 0;
        header = new DNode<E>(null, null, null);          // create header
        trailer = new DNode<E>(header, null, null);       // create trailer
        header.setNext(trailer);     // header and trailer to point to each other
    }
    // checks if p is a valid position for this list and, if valid, converts it to DNode
    protected DNode<E> checkPosition(Position<E> p)
        throws InvalidPositionException { ..... }
    .....
```



```
// checks if p is a valid position for this list and, if valid, converts it to DNode
protected DNode<E> checkPosition(Position<E> p)
        throws InvalidPositionException {

    if (p == null) throw new InvalidPositionException
                                ("Null position passed to NodeList");
    if (p == header) throw new InvalidPositionException
                                ("The header node is not a valid position");
    ......
    DNode<E> temp = (DNode<E>) p;
    ......
    return temp;
}
```

```
// returns the first position in the list
public Position<E> first() throws EmptyListException {
      if ( isEmpty() ) throw new EmptyListException("List is empty");
      return header.getNext();
}
// insert the given element after the given position
public void addAfter(Position<E> p, E e) throws InvalidPositionException {
   DNode<E> v = checkPosition(p);
   numElts++;

   DNode<E> newNode = new DNode<E>(v, v.getNext(), e);
   v.getNext().setPrev(newNode);
   v.setNext(newNode);
}
// implementations of other methods
```





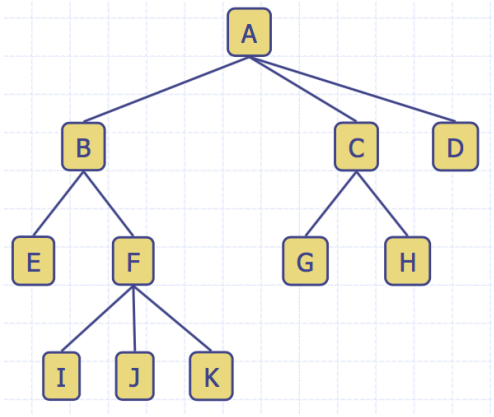
Example

```
PositionList<Integer>  L1 =  new  NodePositionList<Integer>();
L1.addFirst(5);
L1.addFirst(9);
L1.addBefore(L1.last(),3);
...
// print the elements of list L1; assuming L1 is not empty
Position<Integer>  p = L1.first();     // p will access consecutive positions of L1
System.out.print( p.element() + " ");
for ( int i = 0;  i < L1.size()-1;  i++ ) {
    p = L1.next(p);
    System.out.print( p.element() + "  " );
}
```

Trees Structures

Tree Terminology

Root: node without parent (A)
Internal node: node with at least one child (A, B, C, F)
External node (leaf): node without children (E,I,J…)
Ancestors of a node: parent, grandparent etc.
Descendant of a node: child, grandchild, etc.
Siblings: children of the same parent (I, J, K)
Depth of a node: number of ancestors (for F, 2)
Height of a tree: maximum depth of any node (3)
Subtree: tree consisting of a node and its descendants.



Formal Definition

A tree is a set of nodes storing elements such that the nodes have a parent-child relationship, that satisfies the following properties:

- If T is nonempty, it has a special node called the root of T, that has no parent.
- Each node v of T from the root has a unique parent node w . Every node with parent w is a child of w.
- A tree can be empty

Tree ADT

Accessor methods:

Uses the Position class.

- **root()** – returns the tree's root, error occurs if tree is empty – *Position*
- **parent(v)** – returns the parent of v, an error occurs if v is the root. - *Position*
- **Children(v)** – returns an iterable collection containing the children of node v. - *Iterable*

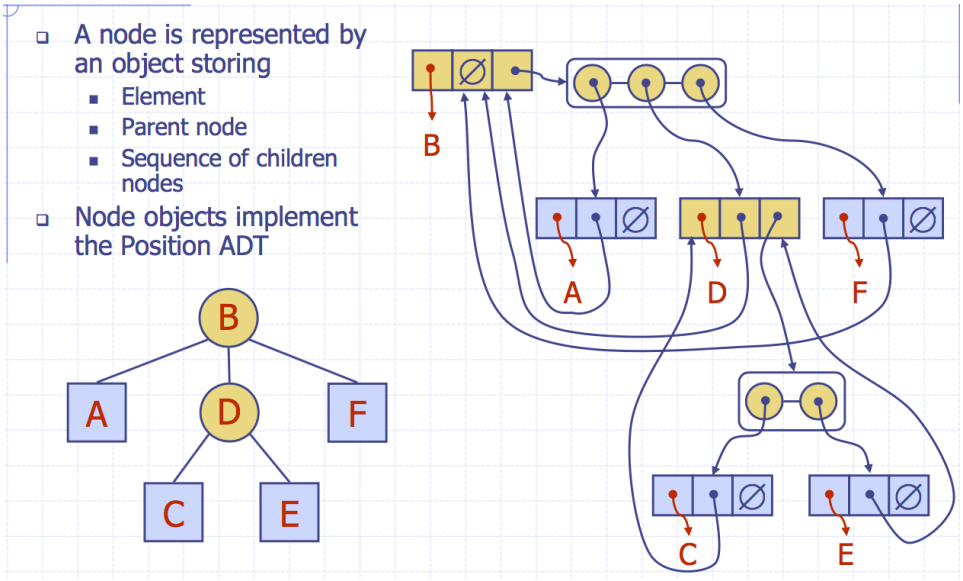
Query Methods:

- **isInternal(v)** – test whether the node v is internal - *Boolean*
- **isExternal(v)** – test whether node v is external – *Boolean*
- **isRoot(v)** – test whether node v is a root – *Boolean*

Generic Methods:

- **size()** – returns the number of nodes in the tree – *Int*
- **isEmpty()** – test whether the tree has any nodes or not – *Boolean*
- **iterator()** – return an iterator of all the elements stored at the nodes of the tree – *Iterator*
- **positions()** – return an iterable collection of all the nodes of the tree – *Iterable*
- **replace(v,e)** – replace with e and return the element stored at node v - *Element*

Linked Structure for Trees



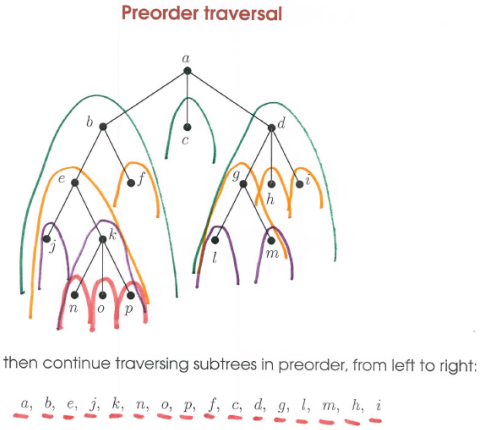
Tree traversal

Rooted trees are used to store information. We need systematic procedures for visiting each vertex of a rooted tree to access data, we call these procedures **traversal algorithms**.

Preorder traversal: visit the root, then continue traversing the subtrees in order, from left to right. Running time for tree with n node is O(n).

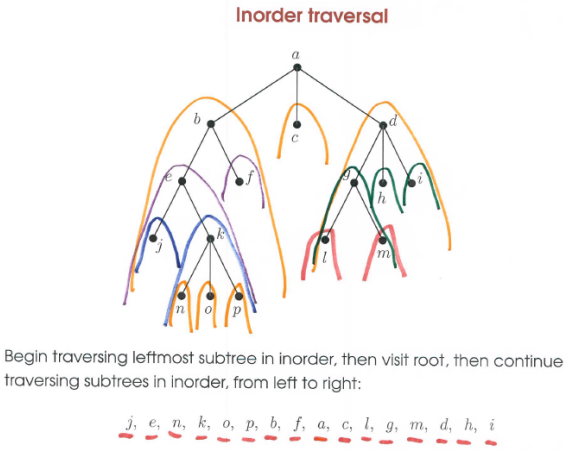
ROOT, LEFT, RIGHT

```
Algorithm preOrder(T, v)
    visit(v)
    for each child w of v in T
        preorder (T, w)
```



Inorder traversal: begin traversing the leftmost subtree in order, then visit the root, then continue traversing subtrees in order, from left to right.

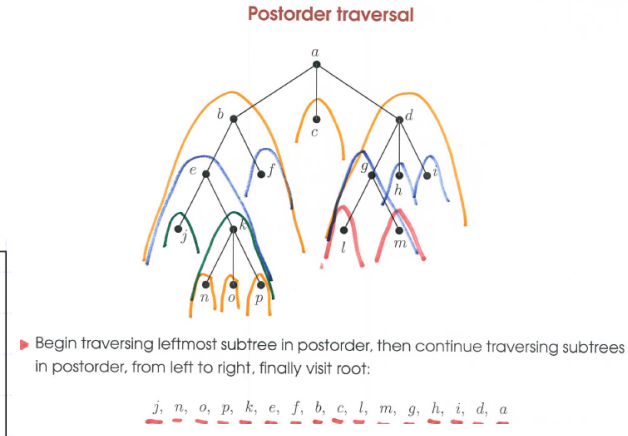
LEFT, ROOT, RIGHT



Postorder traversal: begin traversing leftmost subtree in postorder, then continue traversing subtrees in postorder from left to right, finally visit the root. Running time for tree with n node is O(n).

LEFT, RIGHT, ROOT

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



Binary Trees

- Each internal node has at most two children.
- Each child node is labelled as being either a left child or a right child.
- A left child precedes a right child in the ordering of children of a node.
- A binary tree is **full** if each node has either zero or two children and each internal node has two children.

Binary trees – recursive definition:

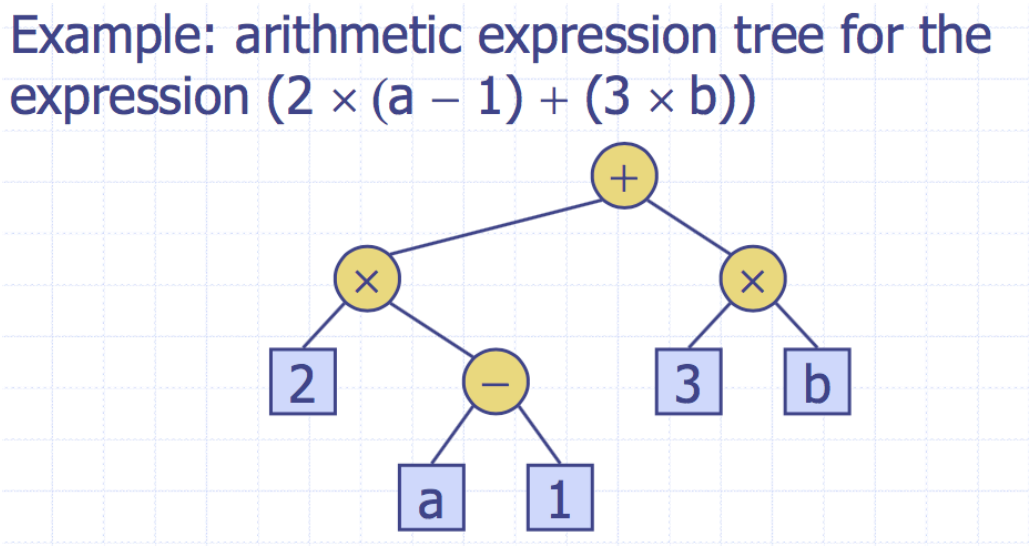
- A node r, called the root of T and storing the element
- A binary tree, called the left subtree of T
- A binary tree, called the right subtree of T

Applications:

- Arithmetic expressions
- Decision processes
- Searching

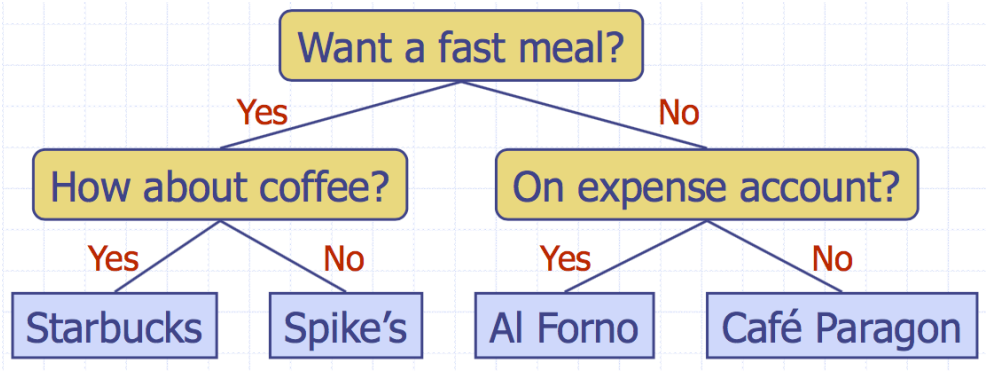
Arithmetic Expression Tree

- Internal nodes are operators (multiplication etc…)
- External nodes are operands (number or symbol)



Decision Tree

- Internal nodes are questions with yes/no answer
- External nodes are decisions



Properties of Binary Trees

- Notation
  - n – number of nodes
  - n_e – number of external nodes
  - n_i – number of internal nodes
  - h – height

Properties:

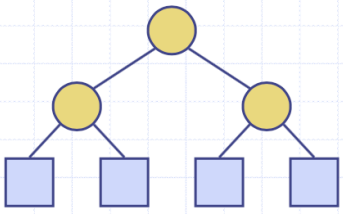
a) $h+1 \le n \le 2^{h+1} - 1$

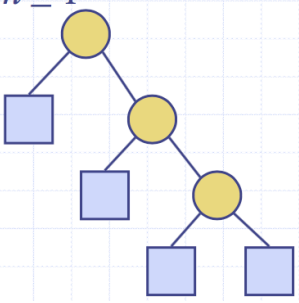
b) $1 \le n_e \le 2^h$

c) $h \le n_i \le 2^h - 1$

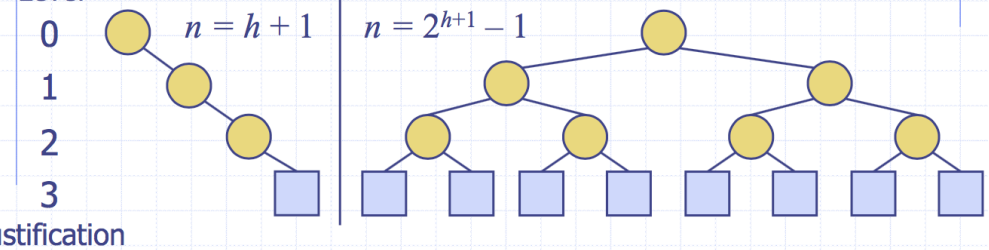
d) $\log_2 (n+1) - 1 \le h \le n - 1$

For $n \ge 1$





a) Let $n \ge 1$ be the number of elements in a binary tree of height $h \ge 0$ then: $h+1 \le n \le 2^{h+1} - 1$



Justification

- We must have at least one element at each level $0, 1, \ldots, h$, so $n \ge h+1$
- At each level i, for $i=0, 1, \ldots, h$ there are at most 2^i elements at this level, so we have:

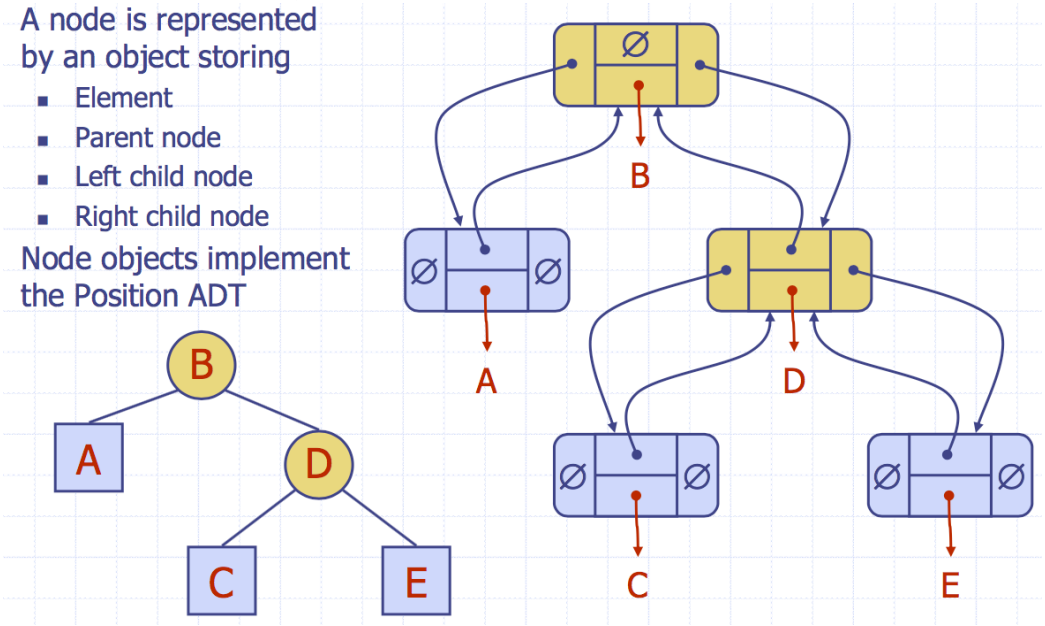
$$n \le \sum_{i=0}^{h} 2^i = 1 + 2 + 4 + \cdots + 2^h = 1 \cdot \frac{1 - 2^{h+1}}{1 - 2} = 2^{h+1} - 1$$

BinaryTree ADT

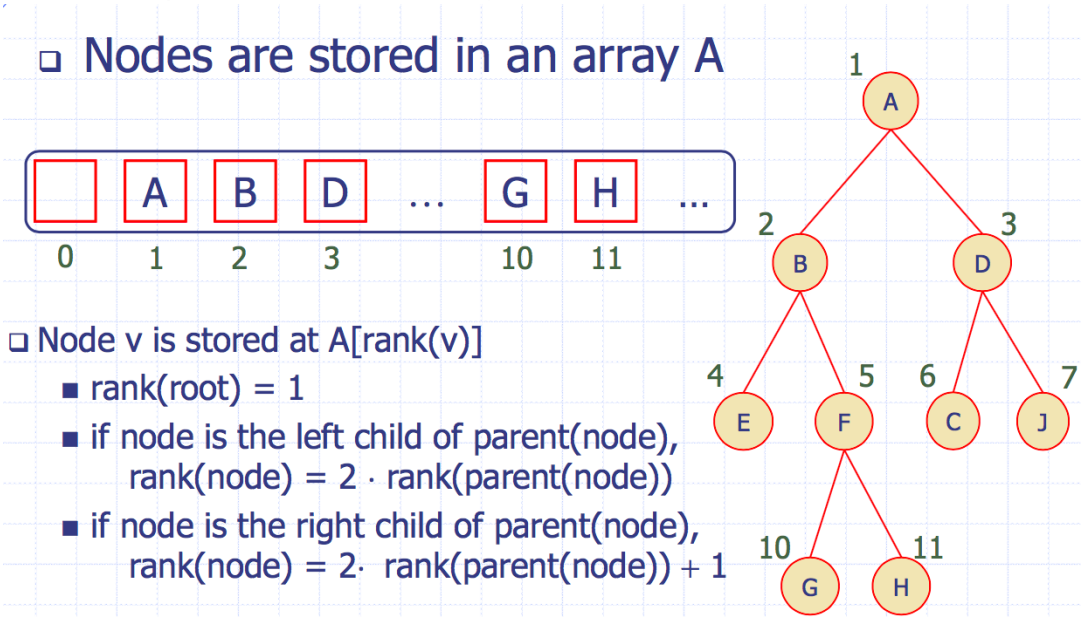
BinaryTree extends Tree and has the additional methods:

- `left(v)` – return the left child of v – *Position*
- `right(v)` – return the right child of v – *Position*
- `hasLeft(v)` – test whether v has a left child – *Boolean*
- `hasRight(v)` – test whether v has a right child – *Boolean*

Linked Structure for Binary Trees

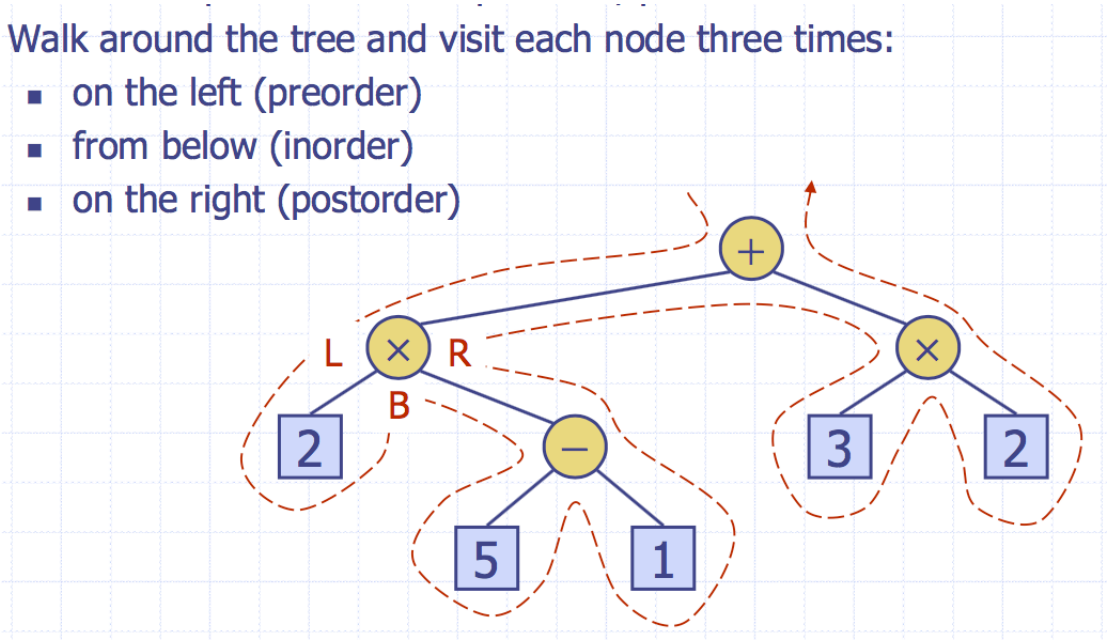


Array-Based Representation of Binary Trees



Euler Tour Traversal

Generic traversal of a binary tree



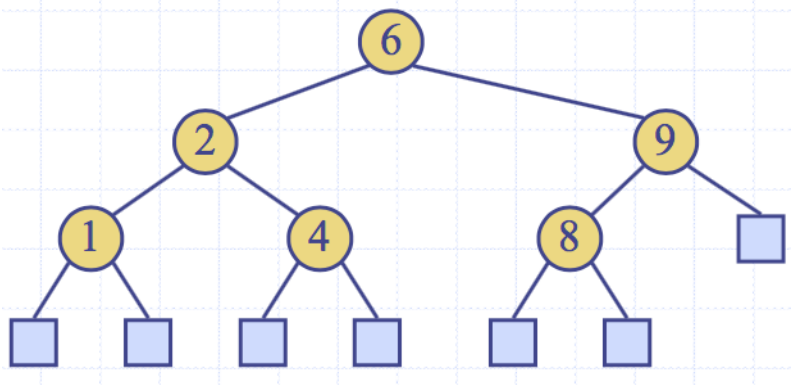
Binary Search Trees

A Binary tree is a tree with key-value entries at its internal nodes that:

- 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.

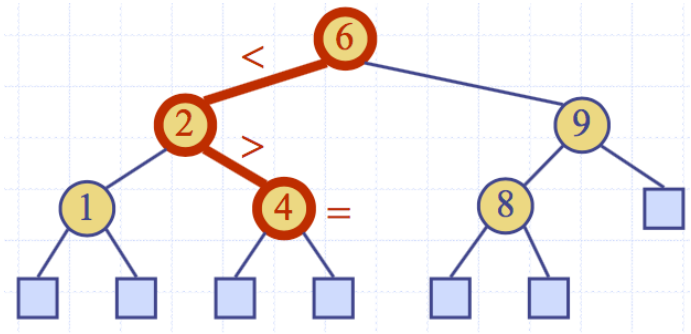
  We have $key(u) \leq key(v) \leq key(w)$

- External nodes do not store items.



Searching

```
Algorithm TreeSearch(k, v)
    if isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, left(v))
    else if k = key(v)
        return v
    else { k > key(v) }
        return TreeSearch(k, right(v))
```



Binary search trees are good to represent elements in a linearly ordered list.

Building Binary Search Trees

1. Put all members in linear order from left to right.
2. The first item is assigned as the **root**.
3. **Comparing**: we take the next item of the list and compare it to the *old* vertex. If is **less than** the one before the old vertex has a left child. If it's **more than** then we add a right child.

Adding: when the new item is **less than** the old vertex and the original one we insert a new left child and label it with the new item. For the **larger than** the same thing occurs with the exception that it creates a right child.

Priority Queue

Priority queue: is a collection of elements called values, each one has an associated **key** that is provided at the time the element is inserted.

Entry: each key-value pair inserted.

The **key** determines the priority used to pick entries to be removed.

Key: an object that is assigned to an element as a specific attribute for that element, which can be used to identify or weigh that element. Two distinct entries in a priority queue can have the same keys.

Total Order Relations

Mathematical concept with the following properties:

- Reflexive property: $x \leq x$
- Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
- Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- Comparison rule that satisfies these three properties will never lead to a comparison contradiction. Such a rule defines a linear ordering relationship among a set of keys.

Priority Queue ADT

Methods:

- **insert(k,x)** - inserts into the priority queue P an entry with key k and value x, returns the inserted entry. Error occurs if k is invalid (k cannot be compared with other keys). - *Entry*
- **removeMin()** – removes from P and returns an entry with the smallest key; an error occurs if P is empty – *Entry*
- **min()** – returns, but doesn't remove the entry with the smallest key. Error occurs if P is empty – *Entry*
- **size()** – returns the number of entries in the priority queue. – *Int*
- **isEmpty()** – tests whether the priority queue is empty. - *Boolean*

Applications:

- Standby flyers
- Auctions
- Stock Market

Entry ADT

An entry is simply a key-value pair.

```
public interface Entry<K,V> {
  public  K getKey();
  public  V getValue();
 }
```

Comparator ADT

A comparator encapsulates the action of comparing two objects according to a given total order relation. It compares two keys although it is external to these.

Methods:

- **compare(Object x, Object y)** – returns an integer i such that:
  - $i < 0 \; if \; x < y$
  - $i = 0 \; if \; x = y$
  - $i > 0 \; if \; x > y$
  - Error occurs if these cannot be compared.
- **equals(Object obj)** – compares a comparator to another comparator.

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

- size and isEmpty take $O(1)$ in both cases

Priority Queue Sorting

We can use a Priority Queue to sort a set of comparable elements.

1. Put the elements in sequence into an empty priority queue (insert operations).
2. Remove the elements in sorted order from the priority queue (removeMin operations) which puts them in order.
   Running time depends on the implementation of the method.

Selection-Sort

Selection-sort is a variation of PriorityQueueSort which is implemented without a sorted sequence.

1. Inserting elements into the priority queue with n insert statements takes O(n) times.
2. Removing the elements in sorted order using removeMin n times takes time

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

$$selection\ sort\ runs\ in\ O(n^2)\ time$$

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

In this case we are inserting a sorted sequence.

Running time of Insertion-sort:

1. Inserting the elments with n operations takes time proportional to:

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

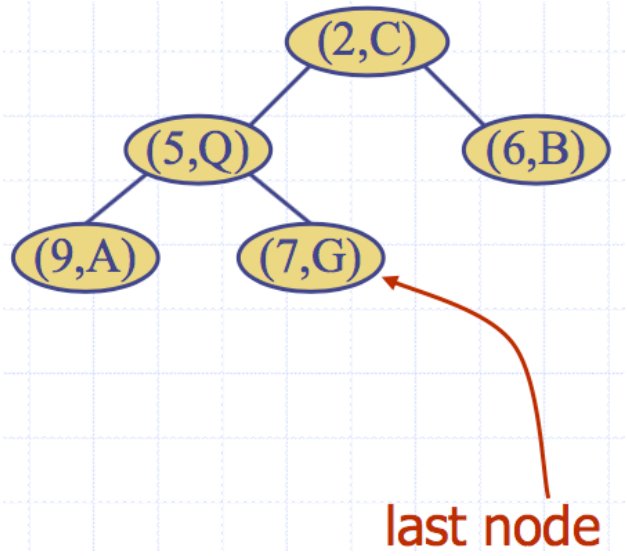
2. Removing the elements in sorter order from the priority queue with a series of n removeMin takes O(n) time.
   It runs in $O(n^2)$.

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Heaps

A heap is a binary tree storing keys at its nodes and satisfying the following properties:

- Heap-Order: for every internal node v other than the root $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap.
  - For $i = 0, \ldots, h-1$, $there\ are\ 2^i\ nodes\ of\ depth\ i$
  - At depth (h – 1), the internal nodes are to the left of the external nodes.
- The **last node** is the rightmost node of maximum depth.

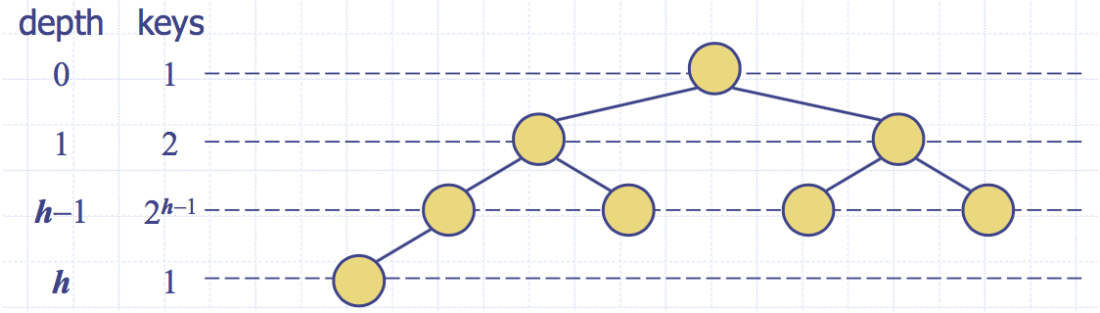


Height of a Heap

Theorem: A heap storing n keys has height $O(\log n)$

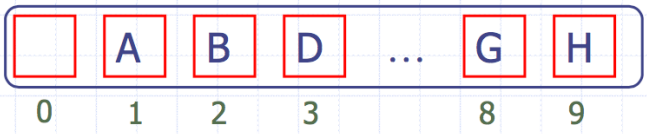
Proof: (we apply the complete binary tree property)

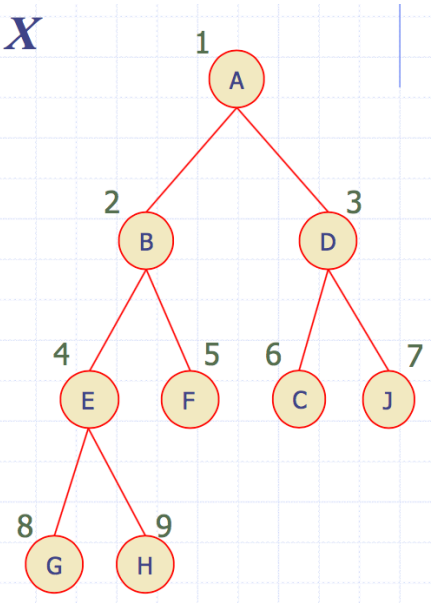
- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$



Array-Based Representation of Complete Binary Trees

- Nodes are stored in an array X





- Node v is stored at X[rank(v)]
  - rank(root) = 1
  - if node v is the left child of parent(v), rank(v) = 2 · rank(parent(v))
  - if node v is the right child of parent(v), rank(v) = 2· rank(parent(v)) + 1

Complete Binary Tree ADT

This includes all the methods for the binary tree ADT and the following:

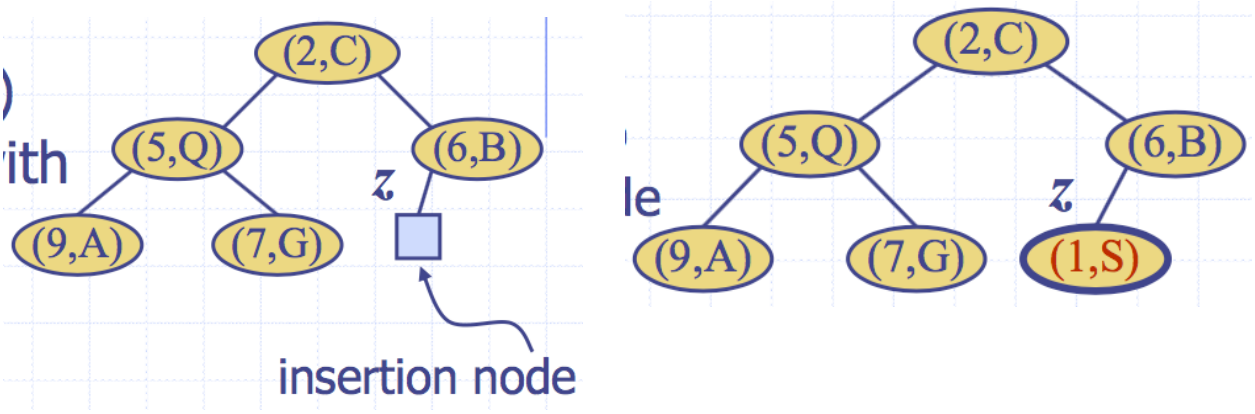
- `Add(o)` – add to T and return a new external node v storing element o that the resulting tree is a complete binary tree with last node v.
- `remove()` – remove the last node of T and return its element.

Heaps and Priority Queues

A heap can be used to implement a priority queue in which a key, value item is stored at a node. We need to keep track of the position of the last node.

Insertion into a heap

Consider the insertion of an entry (1,S) to the heap.

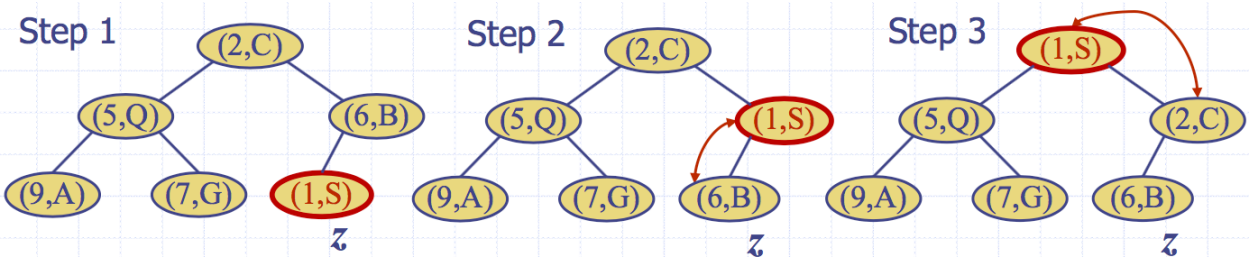


- We add a node to T that stored the entry.
- Then we restore the heap-order property that may be violated.

Up-heap bubbling

The algorithm *up-heap* restores the heap order property by **swapping the entry with key k** along an upward path from the insertion node. Up-heap terminates when the entry with the key **k reaches the root or a node whose parent has a key smaller than or equal to k.**

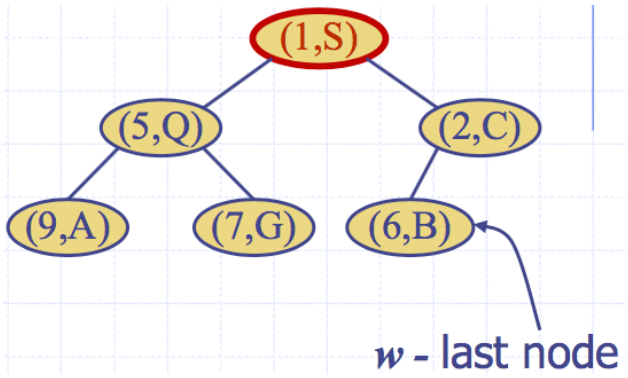
Since a heap has height O(log n), up-heap runs in **O(log n)** times.

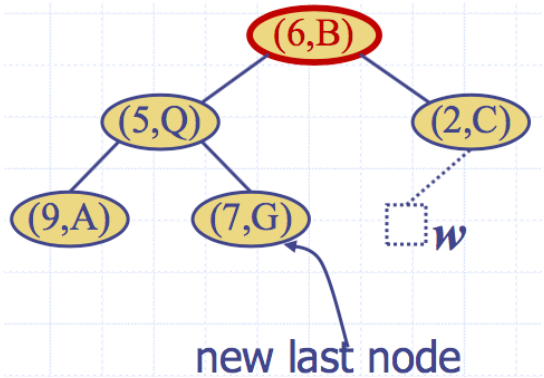


Removal from a Heap

The removeMin() of the priority queue ADT corresponds to the removal of the root key from the heap. It consists of the following steps:

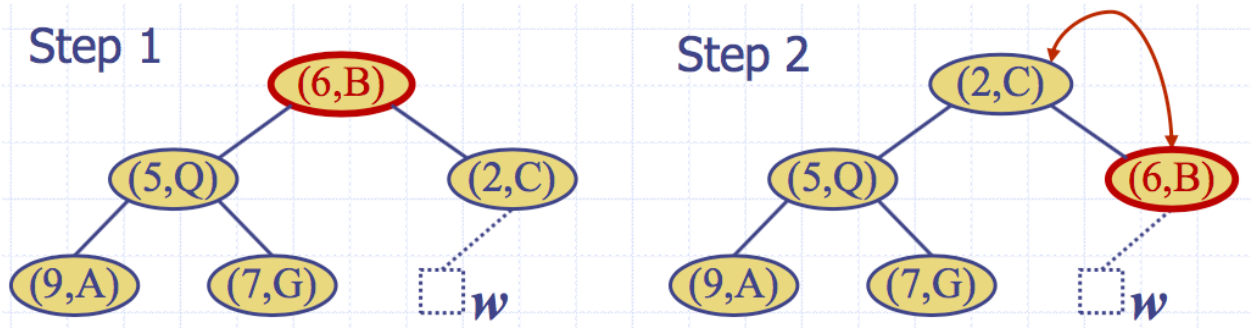
1. Replace the root element with the entry that is in the last node w.
2. Remove w.
3. Restore the heap-order property.





Down-heap bubbling
Algorithm down-heap restores heap-order property by swapping entry with key k along a downward path from the root (swap the entry with key k with its child with the smallest key). It terminated 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), down-heap runs in O(log n) time.



Heap-sort
Consider a priority queue with n items implemented by means of a heap.

- The space used is O(n).
- Methods *insert* and *removeMin* take O(log n) time.
- Methods size, isEmpty and min take time O(1) time.
- Sorting a sequence of elements takes O(n log n) time.
- Much faster than quadratic sporting algorithms such as insertion-sort and selection-sort.

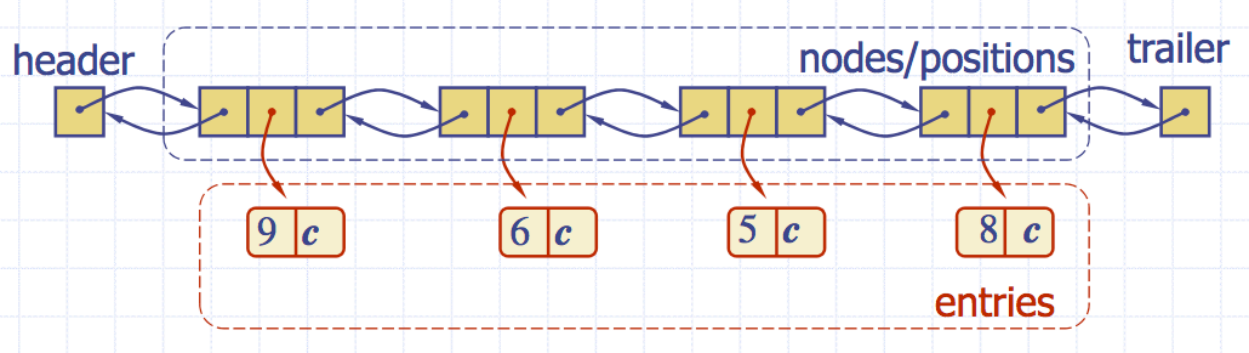
Maps
A map models a searchable collection of key-value entries.

1. Main Operations:
    - Searching
    - Inserting
    - Deleting
2. **Multiple entries** with the same key are **not** allowed.
3. E.g. address book, student-record database.

Map ADT

4. **get(k)** – if the map has an entry with key k, return its associated value v, else return null. – *Entry*
5. **put(k,v)** – if M doesn't have an entry (k,v) then add it to the map and return null, else replace v with the existing value with the same key and return the old value. – *Value*
6. **remove(k)** – if the map has an entry with key k, remove it and return its associated value. Else return null.
7. **size()** – return the number of entries in M – *Int*
8. **isEmpty()** – returns true if M is empty. – *Boolean*
9. **values()** – return an iterator of the values in M.
10. **entrySet()** – return an iterable collection of the entries in M.
11. **keySet()** – return an iterator of the values in M.

In a map, items are stored as a list (based on a doubly-linked list) in arbitrary order.




```
Algorithm get(k):
    B = S.positions() {B is an iterator of the positions in S}
    while B.hasNext() do
        p = B.next()   { the next position in B }
        if p.element().getKey() == k      then
            return p.element().getValue()
    return null {there is no entry with key equal to k}
```

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() == k  then
          t = p.element().getValue()
          S.set(p,(k,v))
          return t        {return the old value}
S.addLast((k,v))
n = n + 1       {increment variable storing number of entries}
return null     { there was no entry with key equal to k }
```

```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
          t = p.element().getValue()
          S.remove(p)
          n = n – 1      {decrement number of entries}
          return t       {return the removed value}
return null              {there is no entry with key equal to k}
```

Performance: put, get and remove take O(n) time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key.

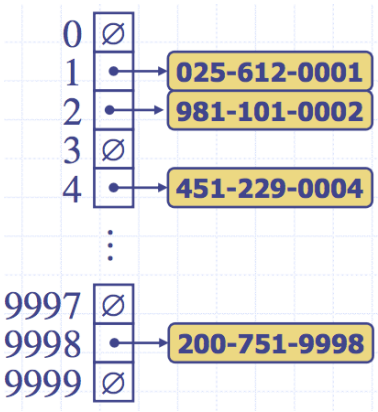
Hash Tables

Hash tables are used to implement a map.

1. They have two main components:
    a. **Bucket array -** an array A of size N, where each cell of A is thought of as a bucket (a collection of key-value pair).
    b. **Hash function h** – which maps keys of a given type to integers in a fixed interval [0, N-1].
        i. Example: h(x) = x mod N is a hash function for integer keys
        ii. The integer h(x) is called the hash value of key x

2. When implementing a map with has table, the goal is to store item (k,v) at index I = h(k).

Example of a hash table for a map storing entries as SSN with a array of size 10 000 and the hash-function: h(x) = last four digits of x.



There would be a problem if there were to be two people with the same SSN (which is not possible) if there was, we would have to select how we do **collision handling**.

Hash Functions

A hash-function is usually specified as the composition of two functions:

3. **Hash code function:** applied first and the compression function is applied next on the result.
4. $h(x) = h_2(h_1(x))$

Hash code:
h_1: keys $\rightarrow$ integers

Compression function:
h_2: integers $\rightarrow [0, N-1]$

Hash codes assigned to keys should avoid collisions. The goal of the hash function is to disperse the keys in an apparently random way.

Hash codes

Memory address:

We reinterpret memory address of the key object as an integer (default hash code of all Java objects).

Integer cast:

We reinterpret the bits of the key as an integer (which means that keys of the length less than or equal to the number of bytes of an integer type.

Component sum:

We partition the bits of the key into components of fixed length (16 or 32 bits) and we sum the components (ignoring overflows). Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type.

Compression Functions

A good hash function should ensure that the probability of two different keys getting hashed to the same bucket is 1/N.

Division:

- $h_2(y) = y \ mod \ N$
- The size n of the bucket array is usually chosen to be a prime.

Multiply, Add and Divide (MAD):

- $h_2(y) = [(ay + b) \ mod \ p] mod \ N$
- where N is the size of the bucket array
- p is a prime number greater than N
- a and b are integers chosen at random from the interval [0,p-1] with a > 0.

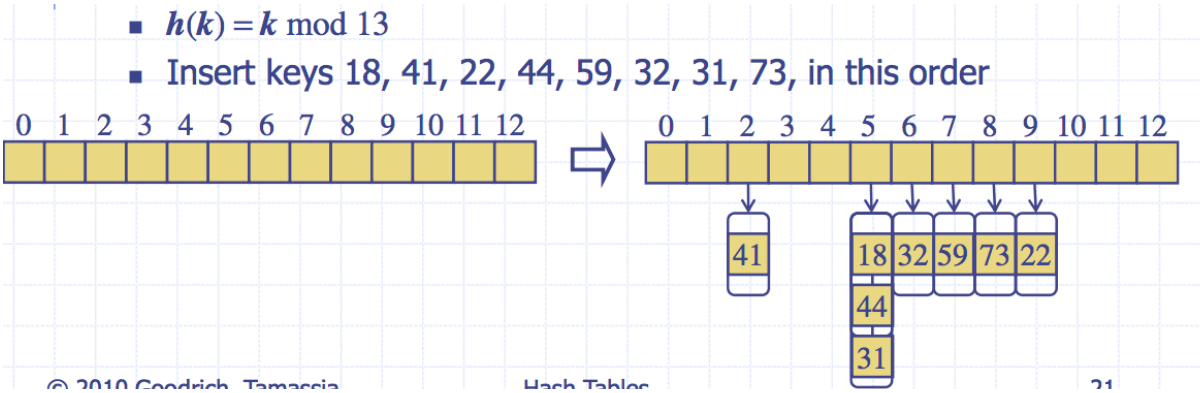
Collision Handling

Collisions occur when different elements are mapped to the same bucket.

Separate Chaining

Separate chaining: let each cell in the table point to a linked list of entries that map there.

This is simple but requires extra memory outside the table.



Open addressing

Open addressing: the colliding item is placed in a different cell of the table:

- Linear Probing
- Double Hashing
- Quadratic probing (individual study, p.396)

Each cell inspected is referred to as probe.
In this example colliding items lump together, causing future collisions to cause longer sequence probes.

Linear Probing

Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell:

- If we try inserting an entry in a cell that is already occupied then we try the next $A[(i + 1) \ mod \ N]$. If this is taken we take +1 and so on until we find an empty cell.

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

Search with linear probing

- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm get(k)
    i = h(k)
    p = 0
    repeat
        c = A[i]
        if c == ∅
            return null
        else if c.getKey () == k
            return c.getValue()
        else
            i = (i + 1) mod N
            p = p + 1
    until    p == N
    return null
```

Updates with linear probing

To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements

- remove(***k***)
  - We search for an entry with key ***k***
  - If such an entry (***k***, ***v***) is found, we replace it with the special item ***AVAILABLE*** and we return element ***v***
  - Else, we return ***null***
- put(***k***, ***v***)
  - We throw an exception if the table is full
  - We start at cell ***h(k)***
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores ***AVAILABLE***, or
    - ***N*** cells have been unsuccessfully probed
  - We store (***k***, ***v***) in cell ***i***

Double Hashing

Double hashing: uses a secondary hash function h'(k). In this case we just add the outcome of this function to the current key they should go to, if this is also taken we add that amount again or more formally:

- If $A[i],\ with\ i = h(k)\ is\ taken,\ then\ we\ iteratively\ try\ the\ buckets\ A[(i + g(j)\ mod\ N]\ for\ j = 1,2,\dots,N-1$
- Where $f(j) = j * h'(k)$
- The secondary hash function cannot have zero values. The table size N must be a prime to allow probing of all the cells.
- A common choice of compression function for the secondary hash function is:

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

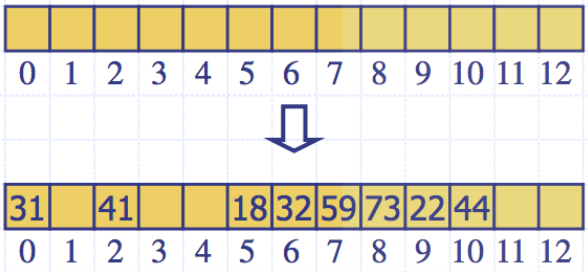
  Where $q < N\ and\ q\ is\ a\ prime$

Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $h'(k) = 7 - k \bmod 7$

Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$h'(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Performance of Hashing

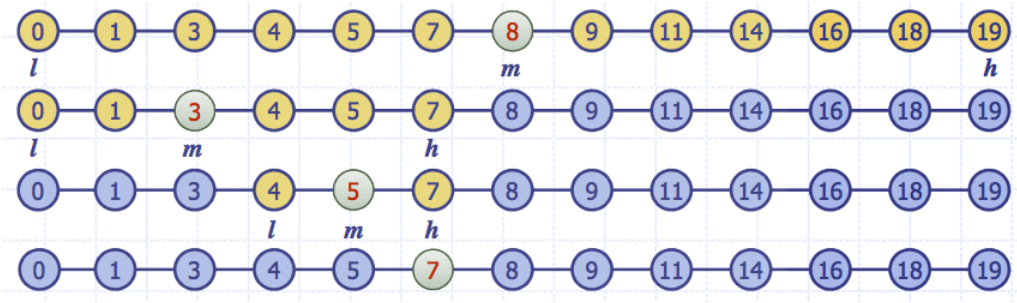
- In the worst cases, searches, insertions and removals on a hash table take O(n) time
- Worst case scenarios occur when all the keys inserted into the map collide
- The load faction a = n/N affects performance of a hash table.
- It can be shows that the expected number of probes for an insertion with open addressing is: 1 / (1 – a).
- In practise, hashing is very fast provided the load factor is not close to 100%
- **Applications:**
    a. Small databases
    b. Compilers
    c. Browser caches

Dictionaries

A dictionary models a searchable collection of key-value entries. The main operations of a dictionary are searching, inserting and deleting items. Multiple items with the same key **are allowed.**

Applications:

- word-definition pairs
- credit card authorizations
- DNS mapping of host names to internet IP addresses.



Dictionary ADT

- **get(k)** – if the dictionary D has an entry with key k, it returns it else returns null.
- **getAll(k)** – returns an iterable collection of all entries with key k.
- **put(k,v)** – inserts into the dictionary D and returns the entry (k,v).
- **remove(e)** – removes the entry e from the dictionary and returns the removed entry. An error occurs if entry is not in the dictionary.
- **entrySet()** – returns an iterable collection of the entries in the dictionary.
- **size(), isEmpty()**

Operation	Output	Dictionary
put(5,A)	(5,A)	(5,A)
put(7,B)	(7,B)	(5,A),(7,B)
put(2,C)	(2,C)	(5,A),(7,B),(2,C)
put(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
put(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(4)	**null**	(5,A),(7,B),(2,C),(8,D),(2,E)
get(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
getAll(2)	(2,C),(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
size()	5	(5,A),(7,B),(2,C),(8,D),(2,E)
remove(get(5))	(5,A)	(7,B),(2,C),(8,D),(2,E)
get(5)	**null**	(7,B),(2,C),(8,D),(2,E)

A log file or audit trail is a dictionary implemented by means of an unsorted sequence. Items are stored in a sequence (based on a doubly-linked list or array) in arbitrary order.

Performance

- put takes O(1) time since we can insert the new item at the beginning of the sequence.
- Get and remove take O(n) time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with a given key.
- The log file is effective only for small dictionaries in which insertions are the most common operations and searches and removals are rarely performed.

```
Algorithm getAll(k)
Create an initially-empty list L
for e: D do
   if e.getKey() == k  then
         L.addLast(e)
return L

Algorithm put(k,v)
Create a new entry e = (k,v)
S.addLast(e)   {S is unordered}
return e
```

```
Algorithm remove(e):
{ We don't assume here that e stores its position in S }
B = S.positions()
while B.hasNext() do
   p = B.next()
   if p.element() == e then
         S.remove(p)
         return e
return null     {there is no entry e in D}
```

Note that we can also create a hash-table dictionary implementation in which separate chaining to handle collisions Is used. Each operation can be delegated to a list-based dictionary stored at each hash table cell.

Ordered Search Table

A search table is a dictionary implemented by means of a sorted array. We store items of the dictionary in an array-based sequence, sorted by key.

Performance:

- Get takes O(log n) time, using binary search.
- Put takes O(n) time since in the worst case we have to shift n/2 items to make room for the new item.
- Remove takes O(n) time since in the worst case we have to shift n/2 items to compact the items after removal.

A search table is only effective for dictionaries of small size or for dictionaries which searches are the most common operations, while insertions and removals are rarely performed.

Search Tree Structures

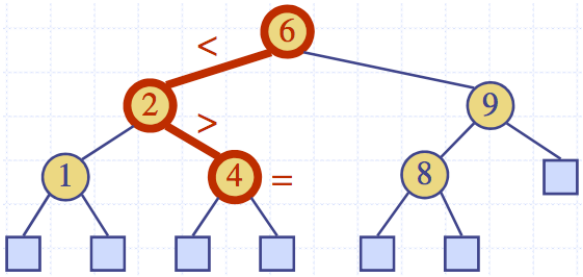
For the future reference assume that a binary tree supports the following operation:

- **`insertAtExternal(w,(k,o))`** – `insert the element (k,o) at the external node w, and expand w to be internal, having new (empty) external node children.`
- **`removeExternal(w)`** – `remove an external node w and its parent, replacing w's parent with w's sibling.`

Search

Example: TreeSearch(4,root)

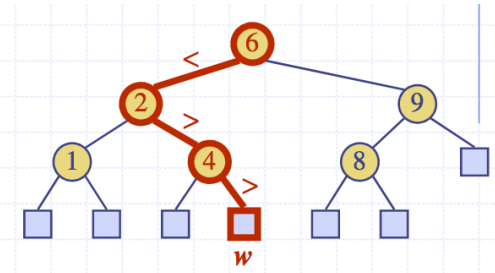
```
Algorithm TreeSearch(k, v)
    if isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, left(v))
    else if k = key(v)
        return v
    else { k > key(v) }
        return TreeSearch(k, right(v))
```

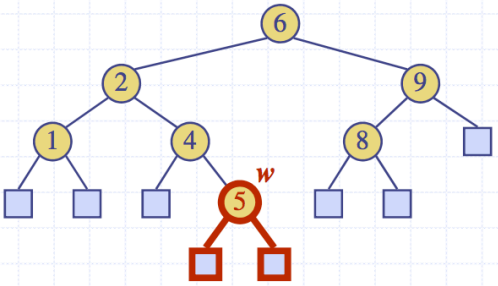


Insertion

- To perform operation put(*k*,*o*), we search for key *k* (using TreeSearch)
- Assume *k* is not already in the tree, and let *w* be the leaf reached by the search
- We insert *k* at node *w* and expand w into an internal node using insertAtExternal(*w*,(*k*,*o*))
- Example: insert 5

To check, we can operate the tree Inorder and it should give all the numbers in increasing order





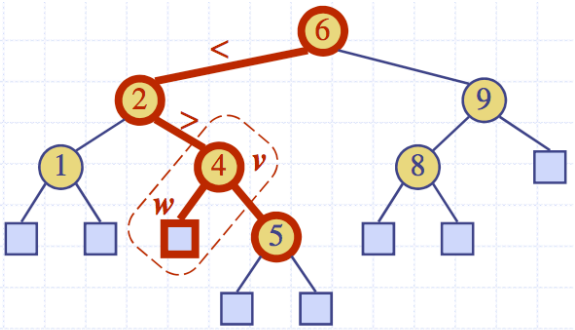
Deletion

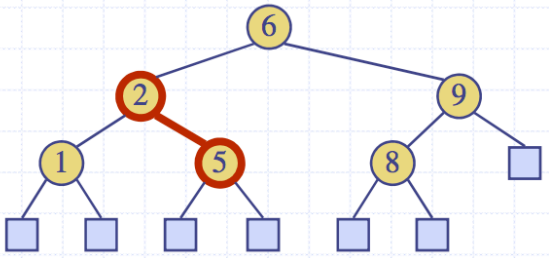
To perform operation remove(*k*), we search for key *k*

Assume key *k* is in the tree, and let let *v* be the node storing *k*

If node *v* has a leaf child *w*, we remove *v* and *w* from the tree with operation removeExternal(*w*), which removes *w* and its parent

Example: remove 4

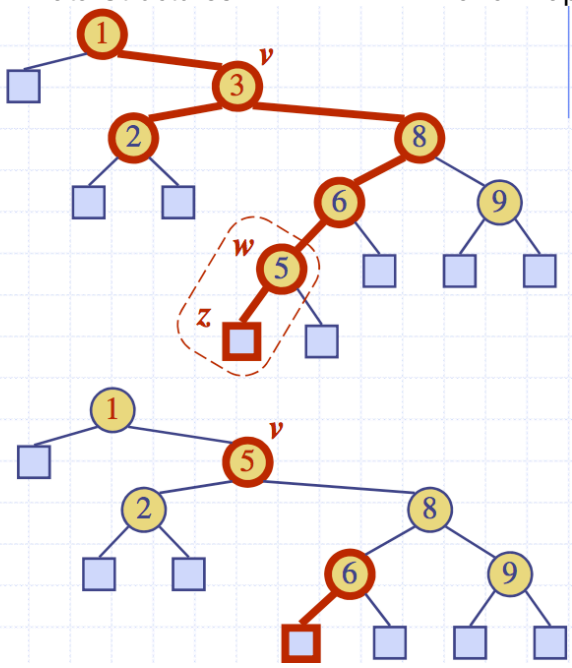




We consider the case where the key k to be removed is stored at a node v whose children are both internal

- we find the internal node w that follows v in an inorder traversal
- we copy $key(w)$ into node v
- we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)

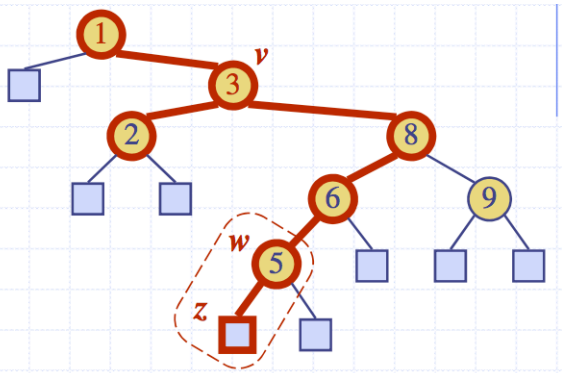
Example: remove 3

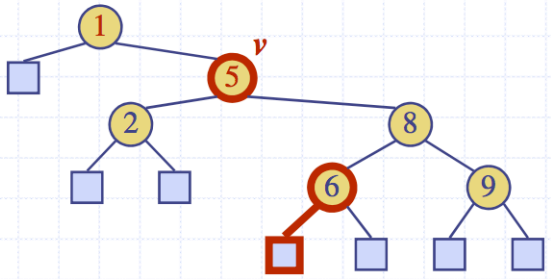


We consider the case where the key k to be removed is stored at a node v whose children are both internal

- we find the internal node w that follows v in an inorder traversal
- we copy $key(w)$ into node v
- we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)

Example: remove 3





Performance

Consider a map with n items implemented by means of a binary search tree of height h.

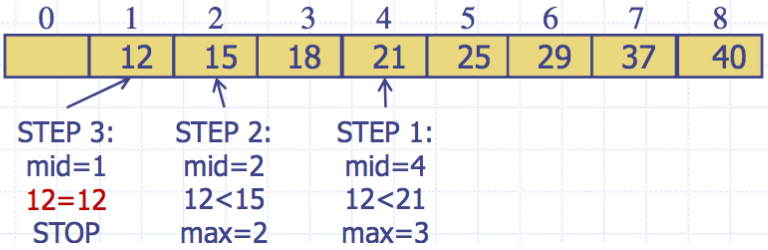
- Space used is O(n)
- Methods get, put and remove take O(h) time (assuming we spend O(1) time at each node)
- The height h is O(n) in the worst case and O(log n) in the best case.

Binary Search

Assume that we have an ordered array A of size N i.e. $[0..N]$ of integers and we want to find key k

Search for k=12

```
Algorithm BinarySearch(k, A, N)
   min := 1;
   max := N;
repeat
     mid := (min+max) div 2;
     if k > A[mid] then min := mid + 1;
          else max := mid- 1;
until (A[mid] = k) or (min > max);
```



In a sorted sequence

Linear search for the key **41** in a sequence with elements in arbitrary order:

46 9 11 27 59 14 17 3 33 63 37 41 52 7 53

Binary search for the key **41** in the sorted sequence:

3 7 9 11 14 17 27 33 37 41 46 52 53 59 63

Compare with the middle element:

3 7 9 11 14 17 27 [33] 37 41 46 52 53 59 63

Search recursively in that half (either the lower half or the upper half) which may contain the search key:

3 7 9 11 14 17 27 [33] 37 41 46 52 53 59 63
3 7 9 11 14 17 27 [33] 37 41 46 [52] 53 59 63
3 7 9 11 14 17 27 [33] 37 [41] 46 [52] 53 59 63

Number of comparisons in the worst case, for a sequence of n elements:

linear search: n **binary search**: [log n] + 1 (all log's with base 2).

Sorting Algorithms

Sorting is the most studied field in computer science as an initial sort can significantly enhance the performance of an algorithm, it is fundamental for computers.

Merge-Sort

Divide-and-Conquer

Is a general algorithm design paradigm:

- **Divide**: divide the input data into two disjoint sets S1 and S2.
- **Recur**: solve the sub-problems associated with S1 and S2
- **Conquer**: combine the solutions S1 and S1 into a solution for S.

Merge-sort

- **Divide**: if S has zero or one element, return S. Else remove all elements from S and put them into two sentences, S1 and S2.

S_1 contains the first $\lceil n/2 \rceil$ elements of S and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements

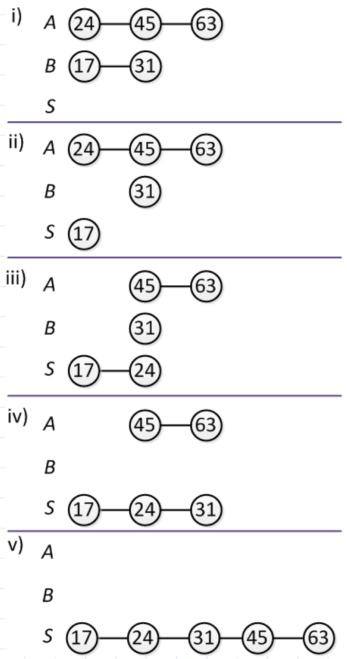
- **Recur**: recursively sort S1 and S2.
- **Conquer**: merge S1 and S2 into a sorted sequence.

The base case for the recursion are sub-problems of size 0 or 1.

```
Algorithm mergeSort(S, C)
    Input sequence S with n elements, comparator C
    Output sequence S sorted
        according to C
    if S.size() > 1
        (S1, S2) ← partition(S, n/2)
        mergeSort(S1, C)
        mergeSort(S2, C)
        S ← merge(S1, S2)
```

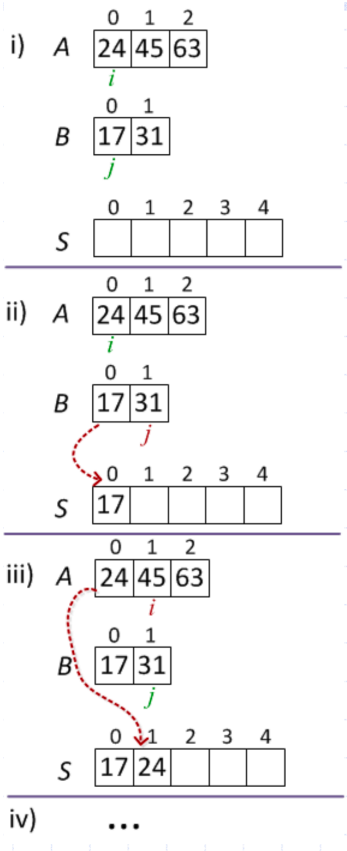
Merging Two Sorted List-based Sequences

```
Algorithm merge(A, B, S)
    Input sorted sequences A and B and an empty
sequence S implemented as linked list
    Output sorted sequence S = A ∪ B

    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        if A.first().element() <= B.first().element()
            {move the first element of A at the end of S}
            S.addLast(A.remove(A.first()))
        else
            {move the first element of B at the end of S}
            S.addLast(B.remove(B.first()))
    {move the remaining elements of A to S}
    while ¬A.isEmpty()
        S.addLast(A.remove(A.first()))
    {move the remaining elements of B to S}
    while ¬B.isEmpty()
        S.addLast(B.remove(B.first()))
    return S
```



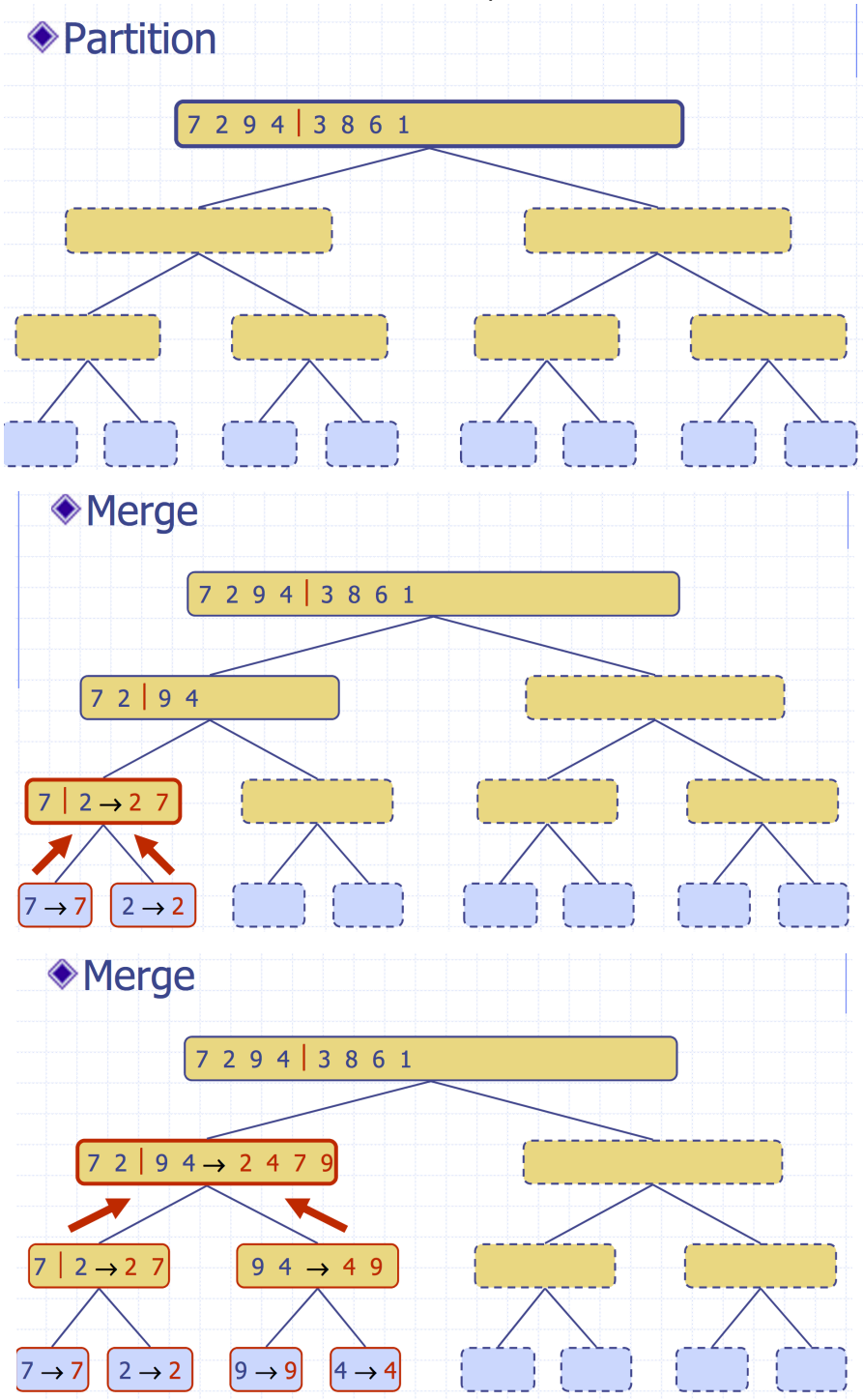
Merging Two Sorted Array-based Sequences

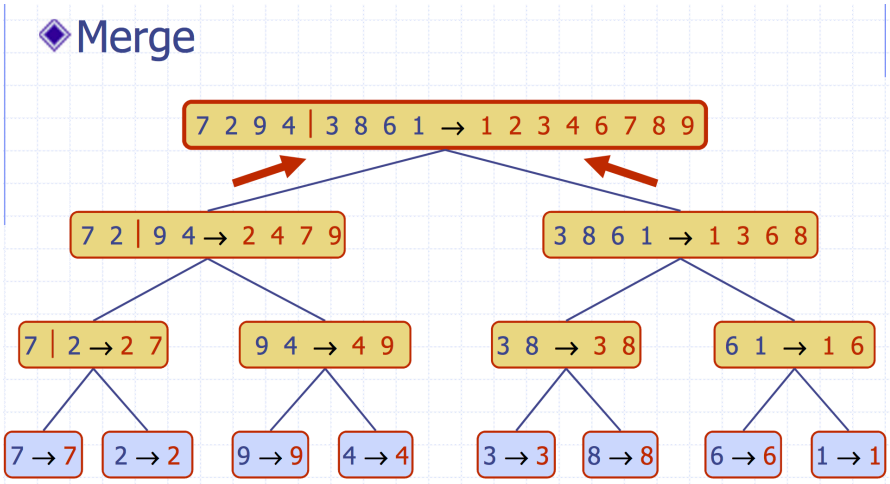
```
Algorithm merge(A, B, S)
    Input sorted sequences A and B and an empty
sequence S, all of which are implemented as arrays
    Output sorted sequence S = A ∪ B
    i ← j ← 0
    while i<A.size() ∧ j<B.size() do
        if A.get(i) <= B.get(j)
            {copy ith element of A to the end of S}
            S.addLast(A.get(i))
            i ← i+1
        else
            S.addLast(B.get(j))
            {copy jth element of B to the end of S}
            j ← j+1
    {move the remaining elements of A to S}
    while i<A.size() do S.addLast(A.get(i)); i ← i+1
    {move the remaining elements of B to S}
    while j<B.size() do S.addLast(B.get(j)); j ← j+1
    return S
```



Tree

Each node represents a recursive call of merge-sort and stores the unsorted sequence before the execution and the sorted sequence at the end of the execution. The root is the initial call and he leaves are calls on sub sequences of size 0 or 1





Analysis of Merge-Sort

The height h of the merge-sort tree is $O(\log n)$

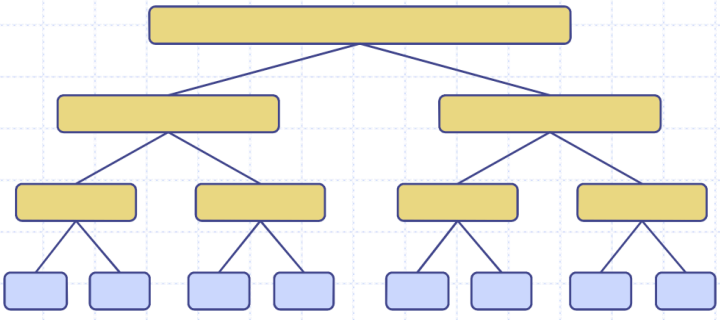
- at each recursive call we divide in half the sequence,

The overall amount or work done at the nodes of depth i is $O(n)$

- tree has 2^i nodes at depth i
- we partition and merge 2^i sequences of size $n/2^i$
- Overall time spent at all the nodes at depth i is $O(2^i \cdot n/2^i)$ which is $O(n)$

Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



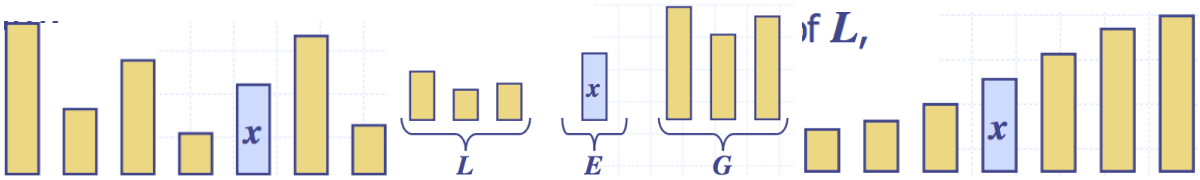
Quick-Sort

Quick sort: is a randomized sorting algorithm based on the divide-and-conquer paradigm.

- **Divide:** If S has at least two elements pick a random element x called **pivot**. Remove all elements from S into three sequences.
  a. **L** elements less than x
  b. **E** elements equal to x
  c. **G** elements greater than x
- **Recur:** sort L and G

- **Conquer:** put back the elements into S in order by first inserting the elements of L, then choose E then G.

Common practice is to **choose the pivot** to be the last element S.



Partition

We remove every element in S and we put it 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 the sequence takes O(1) time.
- Thus, the partition step of quick-sort takes O(n) times.

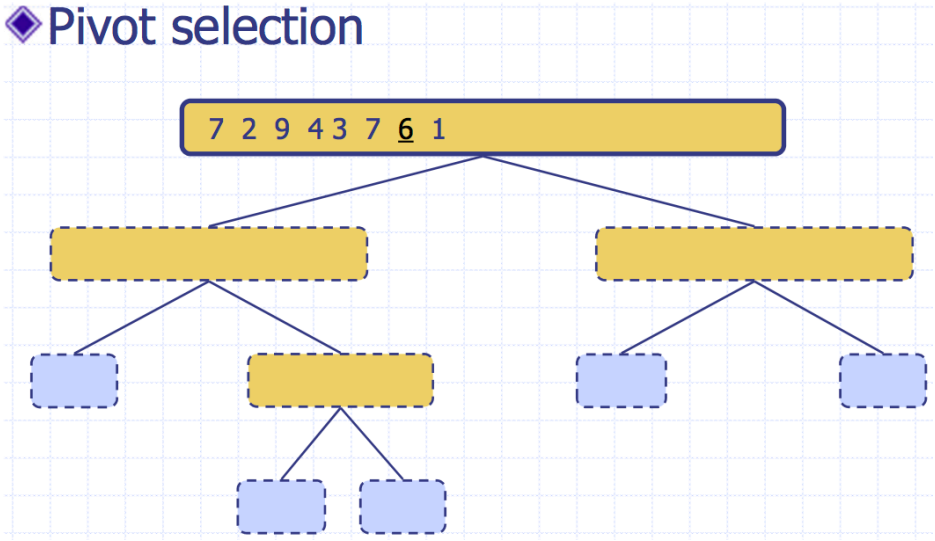
```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    E.addLast(x)
    while ¬S.isEmpty()
        y ← S.remove(S.first())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

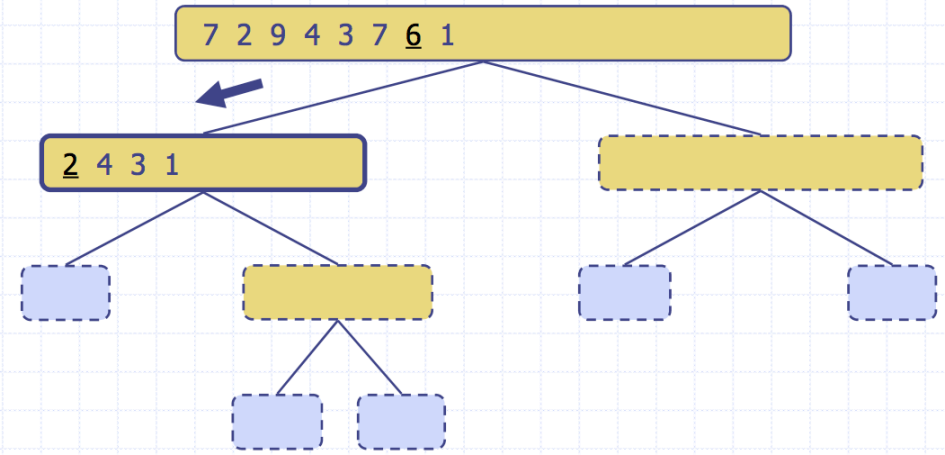
An execution of quick-sort is depicted by a binary tree. Each node represents a recursive call of quick-sort and stores an *unsorted sequence before the execution and its pivot* and *sorted sequence at the end of the execution.*

- The root is the initial call
- Leaves are calls on sub-sequences of size 0 or 1.

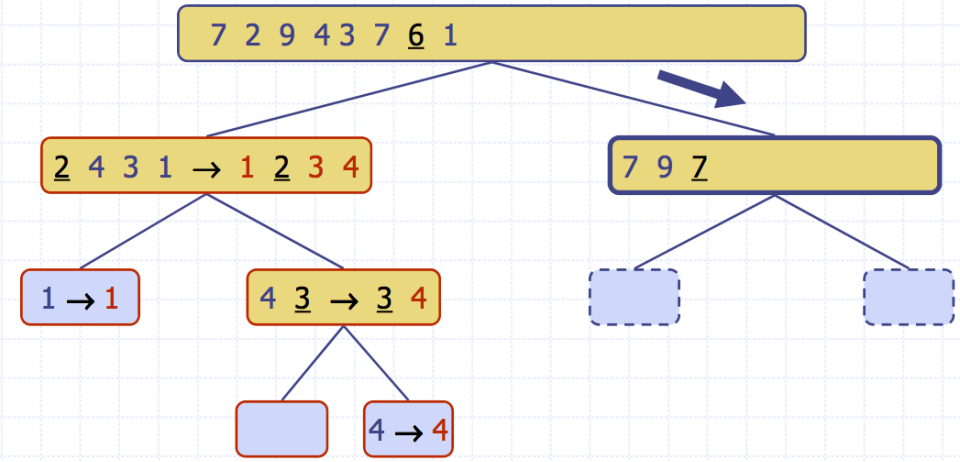
Execution Example (7,2,9,4,3,7,6,1)



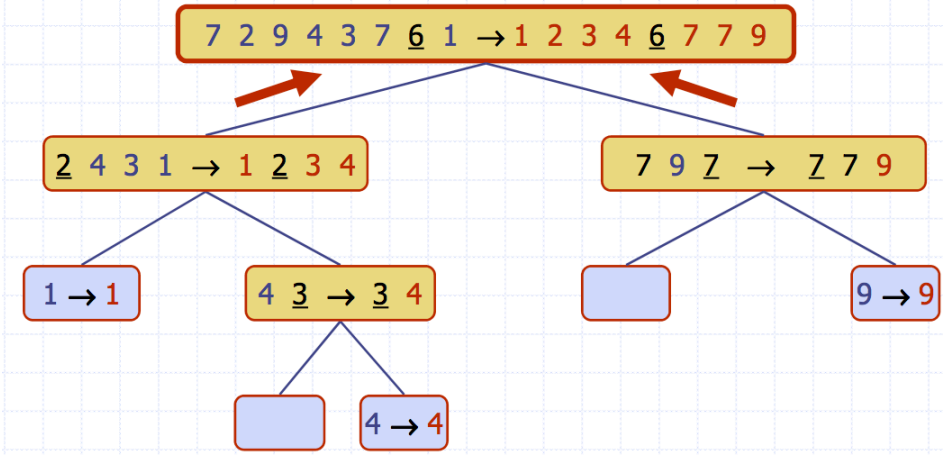
Partition, recursive call, pivot selection



Recursive call, pivot selection



Join, join



Worst-case Running Time

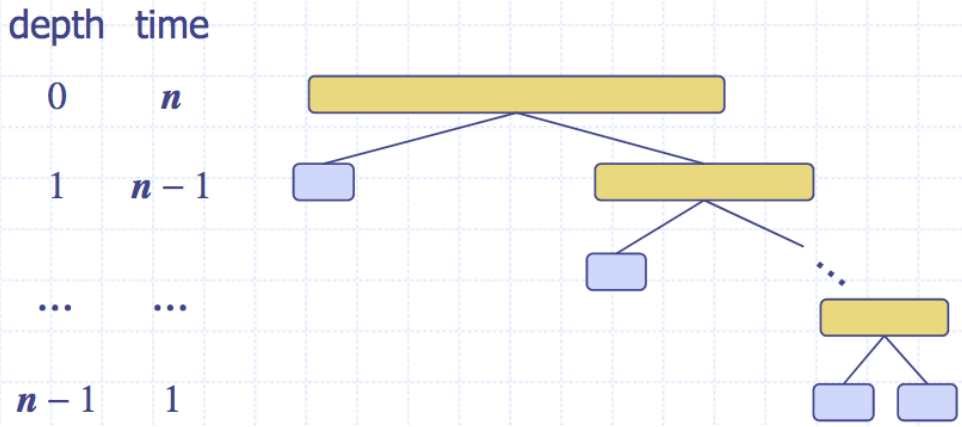
The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element

One of L and G has size $n - 1$ and the other has size 0

The running time is proportional to the sum

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

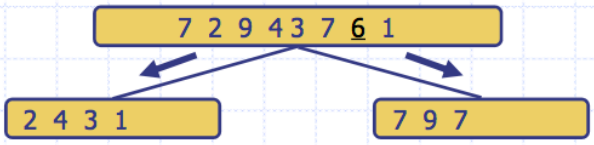
Thus, the worst-case running time of quick-sort is $O(n^2)$



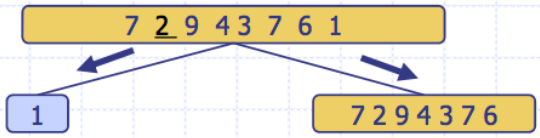
Expected Running Time

Consider a recursive call of quick-sort on a sequence of size s

- **Good call:** the sizes of L and G are each less than $3s/4$
- **Bad call:** one of L and G has size greater than $3s/4$



Good call



Bad call

A call is good with probability $1/2$

- 1/2 of the possible pivots cause good calls:

Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$

For a node of depth i, we expect

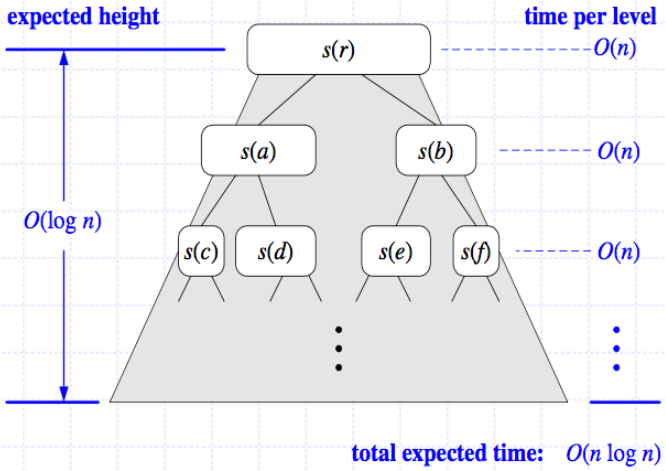
- $i/2$ ancestors are good calls
- The size of the input sequence for the current call is at most $(3/4)^{i/2}n$

Therefore, we have

- For a node of depth $2\log_{4/3}n$, the expected input size is one
- The expected height of the quick-sort tree is $O(\log n)$

The amount or work done at the nodes of the same depth is $O(n)$

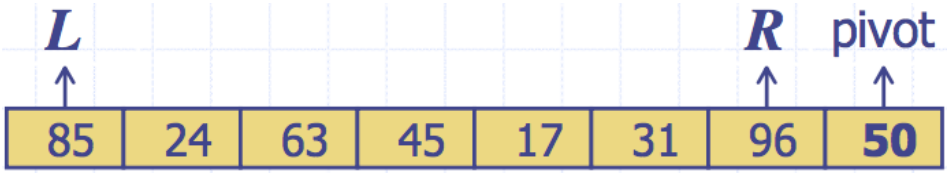
Thus, the expected running time of quick-sort is $O(n \log n)$



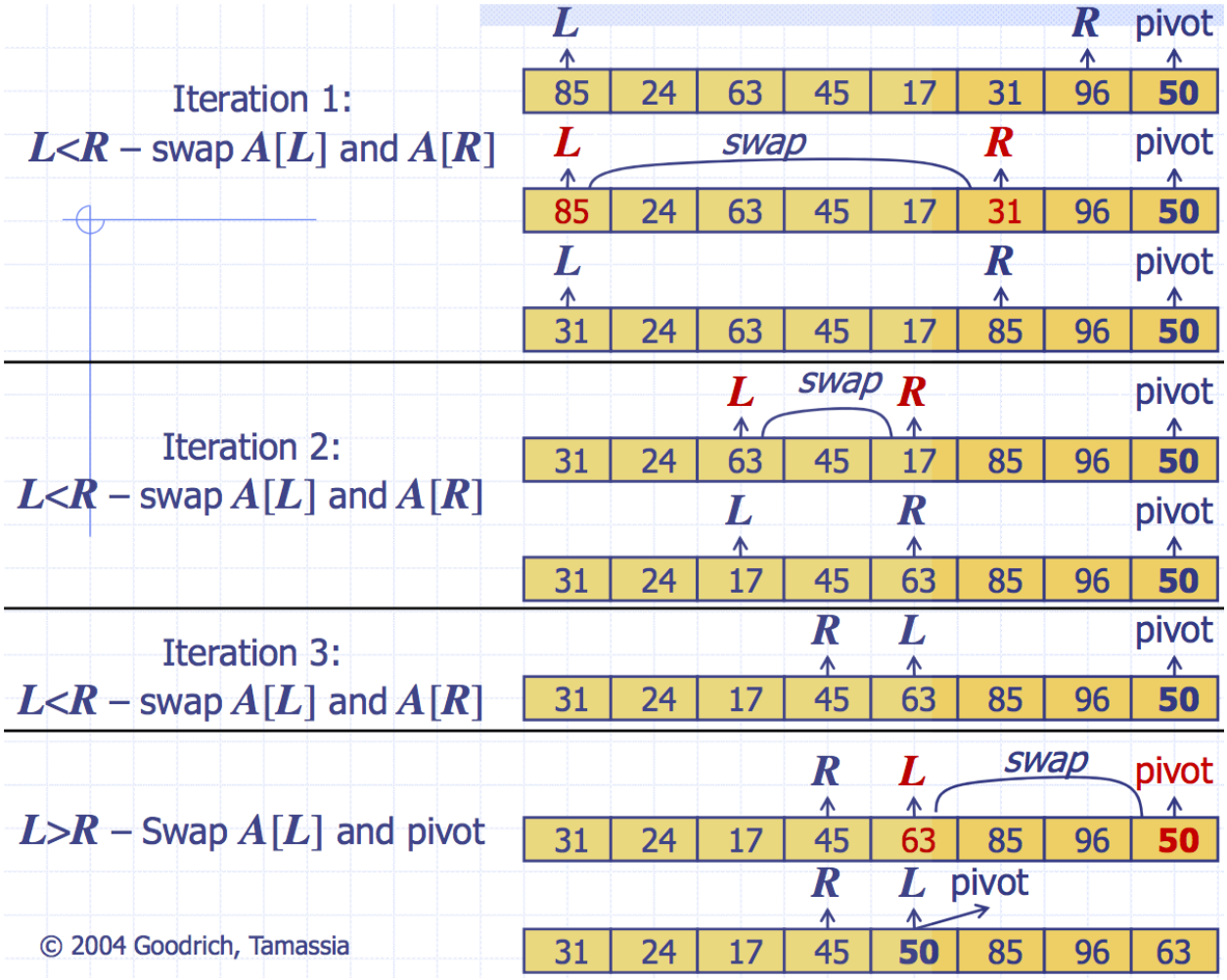
In-place Quick-Sort optimisation of "divide" step

We can make quick-sort "in-place" which is when we do the divide process without using any additional arrays:

Assume we want to devide A[leftEnd…rightEnd] with respect to pivot A[rightEnd]. We can choose L as leftEnd and R as rightEnd-1. The elements which have not been considered yet are in A[(L+1)…(R-1)].



- Iterate until $L<=R$:
  - Keep increasing L by 1 until an element $A[L]$ is smaller than the pivot and $L<=R$.
  - Keep decreasing R by 1 until an element $A[R]$ is larger than the pivot and $L<=R$.
  - If $L<R$ swap $A[L]$ and $A[R]$, and proceed to the next iteration.
- Swap $A[L]$ and pivot



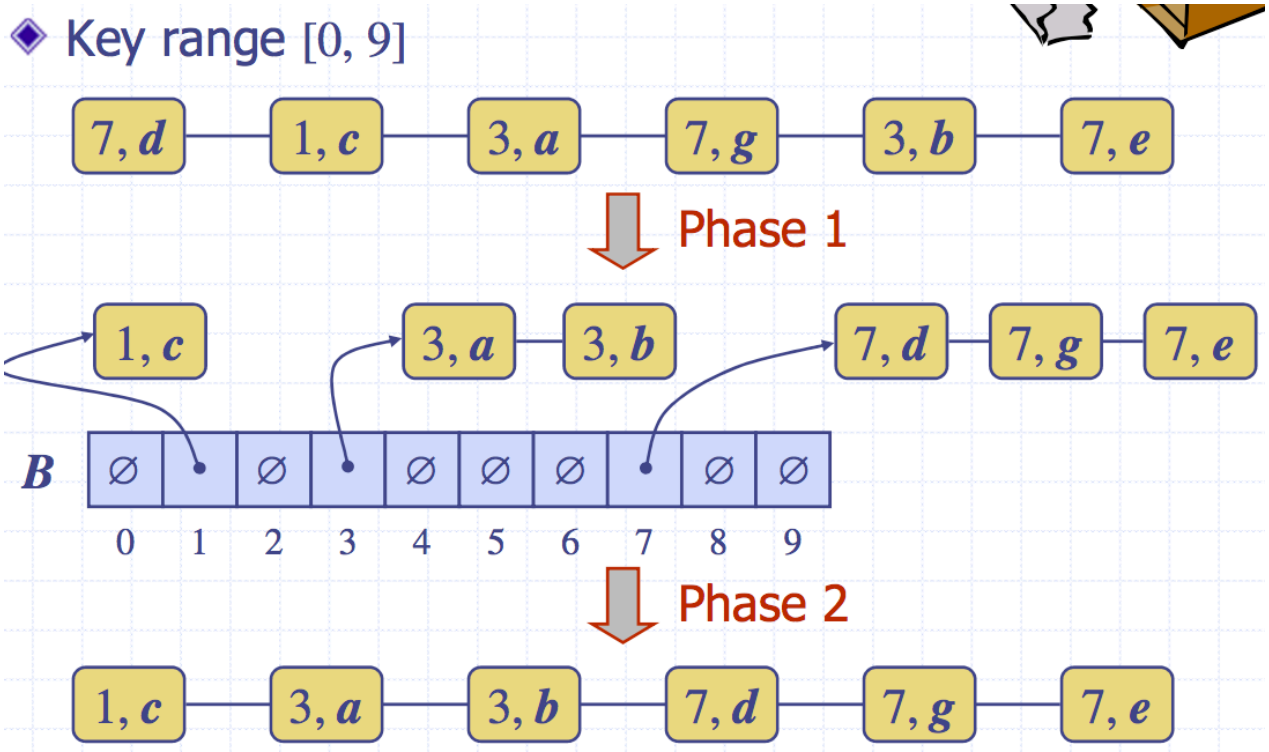
Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

Bucket-Sort

Bucket sort doesn't use comparison.

- Let S be a sequence of *n* (key, element) entries with keys in the range [0, N-1]
- Uses the keys as indices into an auxiliary array B of seuqences.
    a. Phase 1: empty sequence S by moving each entry (k,o) into its bucket B[*k*] – **takes O(n) time.**
    b. Phase 2: For *i = 0, …, N – 1,* move the entries of bucket B[*i*] to the end of sequence S. **– takes O(n + N) time**. **Bucket-sort takes O(n)**

Algorithm *bucketSort(S, N)*
Input sequence S of (key, element) items with keys in the range $[0, N-1]$
Output sequence S sorted by increasing keys
$B \leftarrow$ array of N empty sequences
while $\neg S.isEmpty()$
$f \leftarrow S.first()$
$(k, o) \leftarrow S.remove(f)$
$B[k].addLast((k, o))$
for $i \leftarrow 0$ **to** $N-1$
while $\neg B[i].isEmpty()$
$f \leftarrow B[i].first()$
$(k, o) \leftarrow B[i].remove(f)$
$S.addLast((k, o))$



Stability of sorting

Sorting algorithm is stable if the relative order of any two items with the same key in an input sequence is preserved after the execution of the algorithm.

Stable sorting algorithms:

- Merge-sort
- Bucket-sort

Unstable sorting algorithms:

- Quick-sort

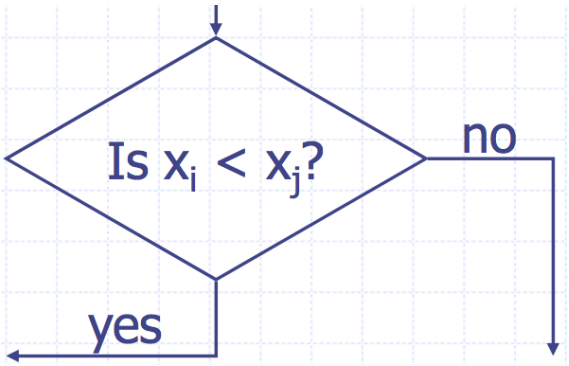
Sorting Lower Bound

Comparison-Based Sorting

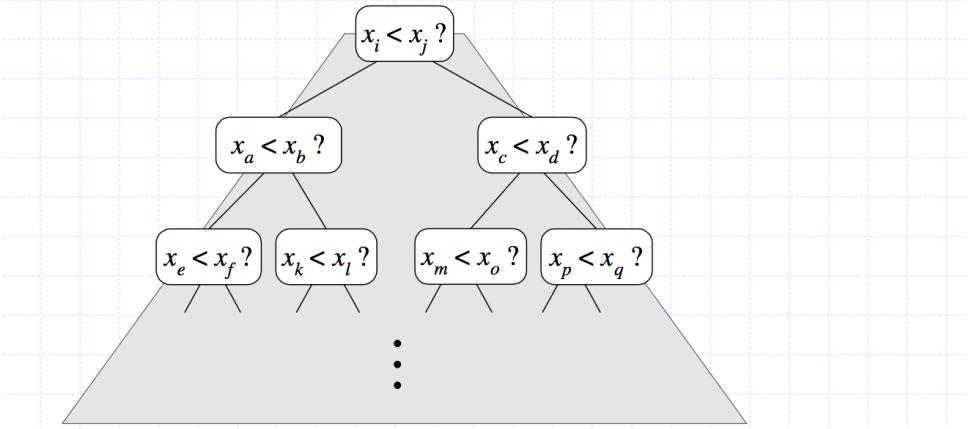
Many sorting algorithms are comparison based (they sort by making comparisons between pairs of objects).

Ex: selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort.

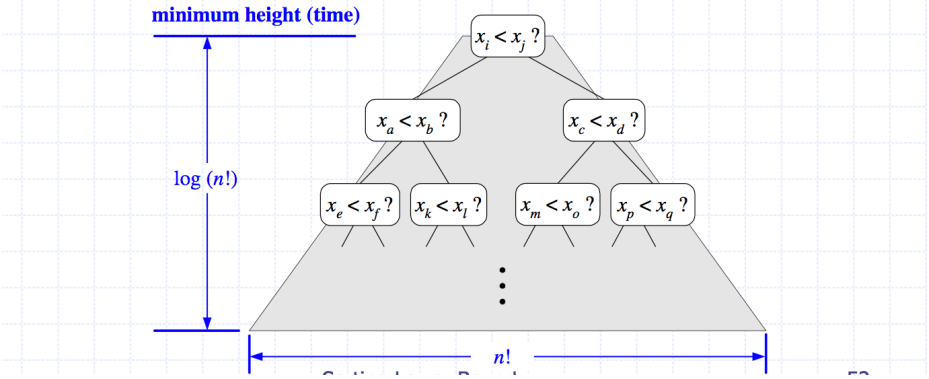
We can derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements.



Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree



The height of the decision tree is a lower bound on the running time
Every input permutation must lead to a separate leaf output
If not, some input …4…5… would have same output ordering as …5…4…, which would be wrong
Since there are n!=1·2 · … ·n leaves, the height is at least log (n!)



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) time.