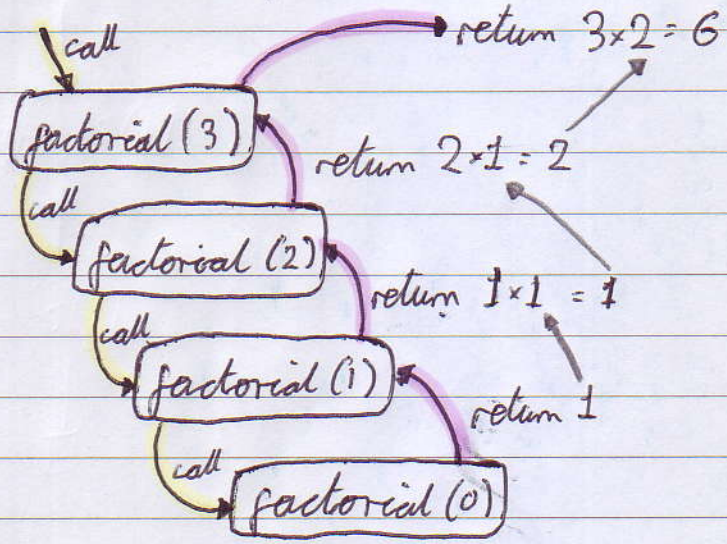
Recursion

- Recursion is when a method calls itself
- Recursive Methods have two main parts:
  - Base Case(s)
    - These are input values for which no recursive calls are made
    - Every possible chain of recursive calls must eventually reach a base case
    - These prevent infinite loops
  - Recursive Calls
    - Calls to the current method
    - Each recursive call should be defined to make progress towards a base case.
- Recursion Tracing
  - A box for each recursive call
  - An arrow from caller to callee
  - An arrow from callee to caller showing return value.
  - Example for factorials:

$$f(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times f(n) & \text{else} \end{cases}$$



• Types of Recursion

- Linear Recursion

- Each call makes only one recursive call if it is not a base case
- The basic process involves checking for a base case, then recurring once (if needed).

- Binary Recursion

- Exactly two recursive calls for each non-base case

- Tail Recursion

- A linearly recursive method makes its recursive call as the last step
- These methods can be easily converted to non-recursive methods, which often saves on resources.

Inheritance

- General classes can be specialised to more particular classes via the extends keyword.

- Allows re-use of code.

- More details on first page of PRA notes

- Polymorphism: an object can take different forms.
  - If the Shape class has been extended by several more then... Shape a = new Square();
    Shape b = new Circle();
    Shape c = new Triangle(); could all be valid.
  - See more notes on first page of PRA notes.

Abstract Classes and Interfaces

See notes in first section of PRA.

Casting

- Converting to "wider" types can be implied
  - Eg. Integer to Number.

- Converting to "narrower" types requires casting
  - Eg. Number to Integer
    Integer n = (Integer) k;

Data Structure Terms

- **Data Structure** - a systematic way of organising, accessing and updating data
- **Algorithm** - a step-by-step procedure for performing some tasks
- **Abstract Data Type** - a model of a data structure that specifies the type of data stored, the operations supported on them, and the type of parameters of the operations
- In Java, an **ADT** is specified by an interface.
- An **ADT** is realised by a concrete data structure, in Java implemented by a concrete class.

Analysis of Algorithms

• Running Time

- Most algorithms transform an input to an output.
- Running time typically increases with input size
- Average case time is often difficult to compute
- Worst case time is used
  - Easier to analyse
  - Easier to define
  - Crucial to some systems.

- Experimental Studies involve writing an implementation and recording execution time
  - Writing code takes time
  - " " may be difficult
  - Results may not consider outside factors
  - Identical hardware must be used for useful comparison.

• Theoretical Analysis

"←" is used for assignment.

- Uses a high-level description (pseudocode) instead of an implementation.
- Considers all inputs
- Works independent of hardware environment
- Characterises the running time as a function of the input size, n

- Counting Primitive Operations

- POs are counted as a function of n, the input size
  - The max number of times a PO could occur is counted.
- POs include assignment, comparison, basic arith., indexing into an array, calling a method, etc.

Eg:

```
1. Algorithm arrayMax(A, n)                    No. of Operations
2.       currentMax ← A[0]                     2
3.       for i ← 0 to n-1 do                   2n+1
4.            if A[i] > currentMax then        2(n-1)
5.                 currentMax ← A[i]           2(n-1)
6.            end if                           0
7.            increment counter i              2(n-1)
8.       return currentMax                     1
```

1. No operations
2. Index into array and assign
3. Initialise i, plus **n** lots of subtract and compare
4. **n-1** loops of index and compare
5. **n-1** loops of index and assign
6. No operations
7. **n-1** loops of increment and assign
8. Return

- arrayMax executes in $8n-2$ operations
- Let a be the time taken by the fastest operation
  b " " " " " " slowest "

- T(n) of arrayMax is therefore bound by two linear functions:

$$a(8n-2) \leq T(n) \leq b(8n-2)$$

• Growth Rate of Running Time

- Changing hardware will...
  - Affect $T(n)$ by a constant factor
  - Not change the growth rate of $T(n)$.

- The linear growth rate of $T(n)$ is an intrinsic property of the arrayMax algorithm.

- Seven Important Growth Functions:
  - Constant $\simeq 1$
  - Linear $\simeq n$
  - Logarithmic $\simeq \log n$
  - N-Log-N $\simeq n \log n$
  - Quadratic $\simeq n^2$
  - Cubic $\simeq n^3$
  - Exponential $\simeq 2^n$

- Growth Rate is not affected by
  - constant factor
  - lower order terms

Eg. $8n + 7$ is linear
$2n^3 + n^2$ is cubic

• Big-Oh Notation

- Given functions $f(n)$ and $g(n)$ we say that $f(n)$ is $O(g(n))$.

if there are +ve constants for c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$.

Eg. $2n + 10$ is $O(n)$

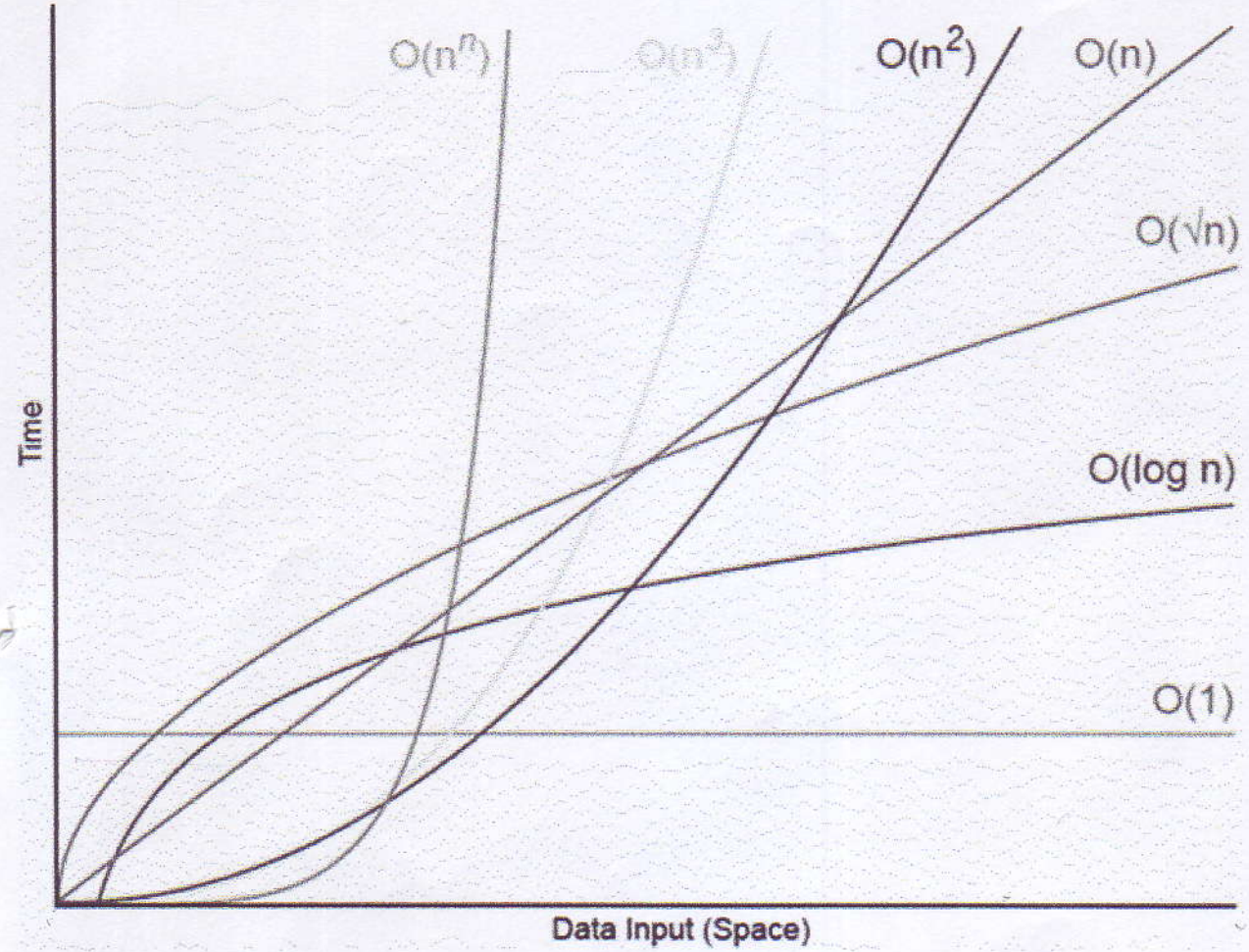
Because if $c = 3$ and $n_0 = 10$, $2n + 10 \leq 3n$ for all $n \geq 10$

- Big-Oh Examples:
  - $7n - 2$ is $O(n)$
  - $3n^3 + 2n^2 + 5$ is $O(n^3)$
  - $3 \log n + 5$ is $O(\log n)$

- Big-Oh provides an upper limit on growth rate of an algorithm or function.
  - The statement $f(n)$ is $O(g(n))$ means $f(n)$ does not grow faster than $g(n)$

- Big-Oh Rules:
  - Drop lower order terms
  - Drop constants
  - Use the smallest possible class of function
  - Use the simplest expression of the class

- Asymptotic Algorithm Analysis
  - Determines the running time in Big-Oh (or a "relative")
  - Steps:
    - Find the worst case and count primitive operations as a function of input size
    - Express the total in Big-Oh notation


O(n^n)
O(n^3)
O(n^2)
O(n)
O(√n)
O(log n)
O(1)
Time
Data Input (Space)

Linked Lists

- A singly linked list is a sequence of nodes
- Each node contains
  - The element (the data)
  - A link to the next node

• The Node Class

- Stores the element/data
- Stores the next node
- Has the appropriate accessor/mutator methods

• The SLinkedList Class

- Stores the head node
- Stores the tail node (if needed; needed to append)
- Stores the length/size (if needed)
- Has list operation methods:

- Insert at Head

- Create new node
- Have new node point to old head
- Update head to point to new node
- Update size
- Update tail if list was empty

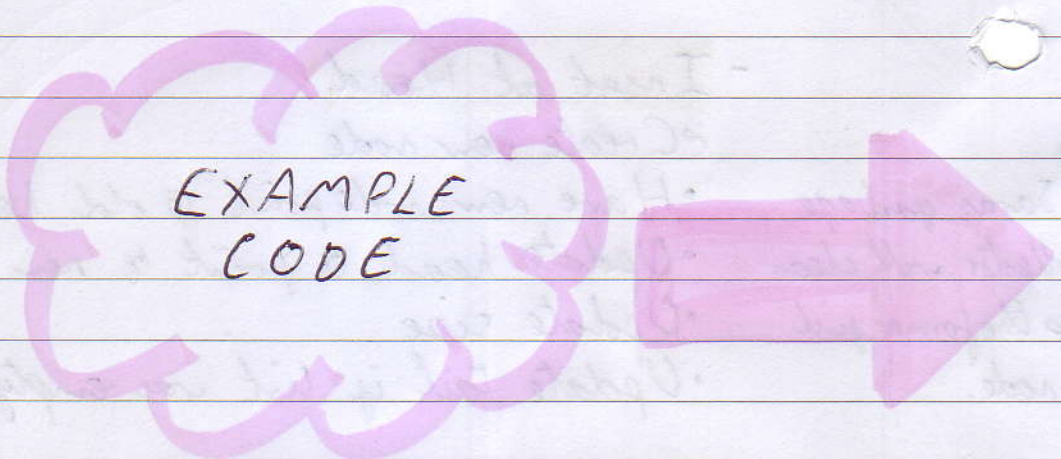
* Java's garbage collector will clean up the former first node.

- Removing at Head

- Update head to point to next node in list *
- Update size
- Update tail if list is now empty
- Return the removed element.

- Inserting at the Tail
  - Create a new node
  - Have the new node point to null
  - Have the last node point to the new node if the list wasn't empty
  - Update tail
  - Update size
  - Update head if the list was empty

- Removing at the Tail
  - The only way is to loop up from the start, because there is no "constant-time" way to update the tail to point to the previous node.



```
public class Node<E> {

      // instance variables
      private E element;
      private Node<E> next;

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

      // null constructor
      public Node() { this(null,null); {

      // accessors
      public E getElement() { ... }
      public Node<E> getNext() { ... }

      // mutators
      public void setElement( E newElem ) { ... }
      public void setNext( Node<E> newNext ) { ... }

}
```

```
public class SLinkedList<E> {

      protected Node<E> head;
      protected Node<E> tail;
      protected long size;

      // default constructor
      public SLinkedList() {
            head = null;
            tail = null;
            size = 0;
      }

      // operation methods
      ...

}
```

Lists

- A collection S of elements stored in a linear order:

S: e1 , e2 , e3 .... ei .... en

0 1 2 i-1 n-1

C index

- The index of an element is the number of preceding elements.

- Stacks and queues can be seen as restricted lists.

- ArrayList and LinkedList are two list types in Java - they both implement the List<E> interface.

• ADTs, Interfaces and Classes.

ArrayList ADT:
add(i,e), get(i)...

↑ specifies

Interface: IndexList<E>
methods: add(int, E)
get(int)
....

↑ implements

Class: ArrayIndexList<E>
implements IndexList methods

NodeList ADT:
addFirst(e), addBefore(i, e)...

↑ specifies

Interface: PositionList<E>
methods: addFirst(E)
addBefore(int, E)
....

↑ implements

Class: NodePositionList<E>
implements PositionList methods.

• Arraylist ADT

- An instance: a sequence S of elements
- Methods:
  - size()
  - isEmpty()
  - get(i) - get value at i, or error on invalid i
  - set(i, e) - set value at i,* or error on invalid i
  - add(i, e) - insert a new element to have index i,** or error if i<0 or i> size().
  - remove(i) - remove and return the element at i, or error for invalid i.

* and return the old value.

** end of list upwards if inserted inside the list.

- Array-Based Implementation

- add(i, e):
  shift all elements with index >= i up to make room for e at i

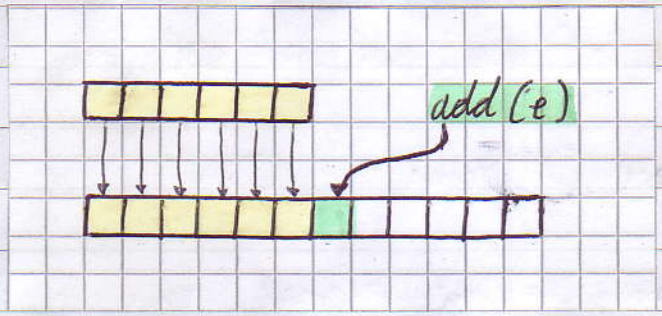
```
Algorithm add(i,e):
  for (j = n-1, n-2, ..., i) do
      A[j+1] ← A[j]
  A[i] ← e
  n ← n+1
```

- remove(i):
  shift down all elements with index >= i

```
Algorithm: remove(i):
  e ← A[i]
  for (j = i, i+1, i+2, ..., n-2) do
      A[j] ← A[j+1]
  n ← n-1
  return e
```

- Extending a Full Array.

- When inserting to a full array, a new larger array is allocated and all elements are copied to the new array.



- How big should the new array be?

- If the size increases by 1 each time, then n add() operations takes $O(n^2)$ time.

- If the size increases by some other constant, the time for n add() operations is still $O(n^2)$.

- If the size is doubled, running time is $O(n)$ for n add() operations

  - Assume the initial size is 2, and N is the final capacity:

$$O(4 + 8 + 16 + 32 + \dots + N)$$
$$= O(2N - 4)$$
$$= O(N) = O(n).$$

- Therefore, size is doubled when adding to a full array.

- Performance:

° In an array-based implementation of the Array-List ADT with n elements:

- size, isEmpty, get and set run in $O(1)$ time.
- remove(i) runs in $O(n)$, or more precisely $O(n-i)$
- removing the last element runs in $O(1)$.
- add(i, e) runs in $O(n)$
- adding an element at the end takes $O(n)$ in the worst case (full array) but only $O(1)$ on average.

• Position

- A place where an element is stored
- position ≠ index ← important!
- A user can only use a position*:

p.getElement(), addAfter(p, e), etc..

* positions are nodes in most implementations

• Node List ADT

- An instance: a sequence of positions, each storing an object
- Methods:
  - size()
  - isEmpty()
  - first()
  - last()
  - prev(p)
  - next(p)

  } Return a position (node)
  - set(p, e)
  - addFirst(e)
  - addLast(e)
  - addBefore(p, e)
  - addAfter(p, e)
  - remove(p)

p is a position/node.

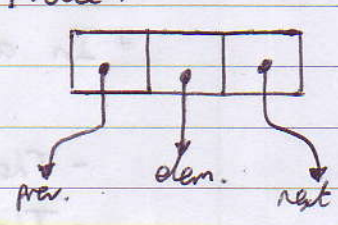
- Each position p has a relation to the nodes before/after.

- Doubly Linked Lists

- DLLs are good examples of the Node List ADT.

- Nodes implement Position and store:
  - the element
  - a link to the prev. node
  - a link to the next node.

Node:



- DLLs use special trailer nodes for head and tail

- Insertion

```
Algorithm addAfter(p, e):
    Create new node v
    v.setElement(e)
    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
    count++
```

- Deletion

```
Algorithm remove(p)
    t ← p.getElement()
    (p.getPrev()).setNext(p.getNext()).   Linking out p
    (p.getNext()).setPrev(p.getPrev())    Linking out p
    p.setNext(null)                       invalidate p
    p.setPrev(null)                       invalidate p
    count--
    return t
```

- Performance

- In a DLL implementation of a Node List ADT:

  - The space used by each node is $O(1)$
  - The space used by a n-sized list is $O(n)$
  - All operations run in $O(1)$

Tree Structures

- A tree consists of nodes with a parent/child relation
- Full definitions: see "Trees" in FC1 notes.
- NB: a tree can be empty.
- A tree is ordered if there is a linear order defined for the children of each node.

• Tree ADT

- Stores elements at positions that are defined relative to neighbouring positions
- Each position (node) has a • element() method
- Accessors:
  - position root() - root position, or error for empty tree
  - position parent(v) - parent of v, or error if v is root
  - Iterable children(v) - iterable collection of v's children
- Query Methods:
  - bool isInternal(v)
  - bool isExternal(v)
  - bool isRoot(v)

  } self-documented
- Generic Methods:
  - integer size()
  - bool isEmpty()
  - Iterator iterator() - iterator of elements
  - Iterator positions() - iterator of nodes/positions
  - element replace(v, e) - replace node v's content w/ e and

- Linked Structure:

  - A node implements the Position ADT.
  - Each node stores:
    - The element
    - The parent node (null for root)
    - A sequence of children nodes.

- Tree Traversal → NLr → LrN

  - See "Preorder" and "Postorder" in FC1 notes

- Binary Trees

  - Definition: see "2-ary / full 2-ary trees" in FC1 notes

  - Applications: arithmetic, decision processes, searching
    - Arithmetic: interior nodes = operators
      exterior nodes = numbers / variables
    - Decisions: interior nodes = questions w/ yes/no answers
      exterior nodes = decisions

  - Properties:

$$h+1 \leq n \leq 2^{h+1}-1$$

$$1 \leq n_e \leq 2^h$$

$$h \leq n_i \leq 2^h-1$$

$$\log_2(n+1)-1 \leq h \leq n-1$$

  - Notation:
    - n = number of nodes
    - n_i = number of interior nodes
    - n_e = number of exterior nodes
    - h = height

• Binary Tree ADT

- Extends the Tree ADT

- Additional Methods:
  - position left(v) } returns the left/right child of v
  - position right(v) }
  - boolean hasLeft(v) } check presence of left/right child for v
  - boolean hasRight(v) }

- Linked Structure:

  - A node implements the Position ADT
  - Each node stores:
    - The element
    - The parent node (null for root)
    - The left child node } can be null.
    - The right child node }

• Storing in Arrays

- Each node v is stored at A[rank(v)]

  - rank(root) = 1
  - rank(node) = { 2 × rank(parent) — if node is left child
    { 2 × rank(parent) + 1 — if node is right child

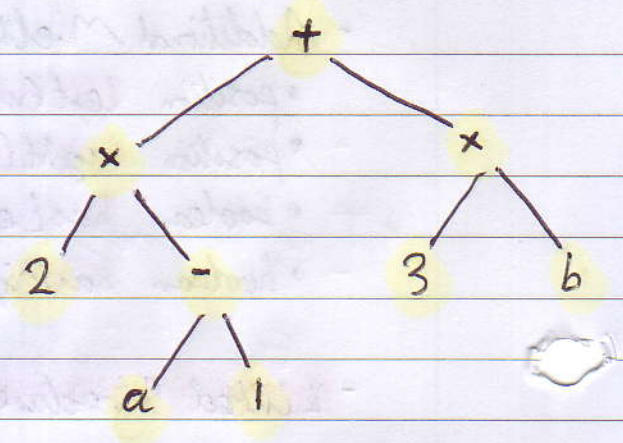
• In-Order Traversal

- See "In-Order" in FCI notes
  ↳ LNr

• Special Types of In-Order:

- Print an expression:

```
Algorithm printExpr(T, v)
  if hasLeft(v)
    print "("
    printExpr(T, left(v))
  print v.element()
  if hasRight(v)
    printExpr(T, right(v))
    print ")"
```

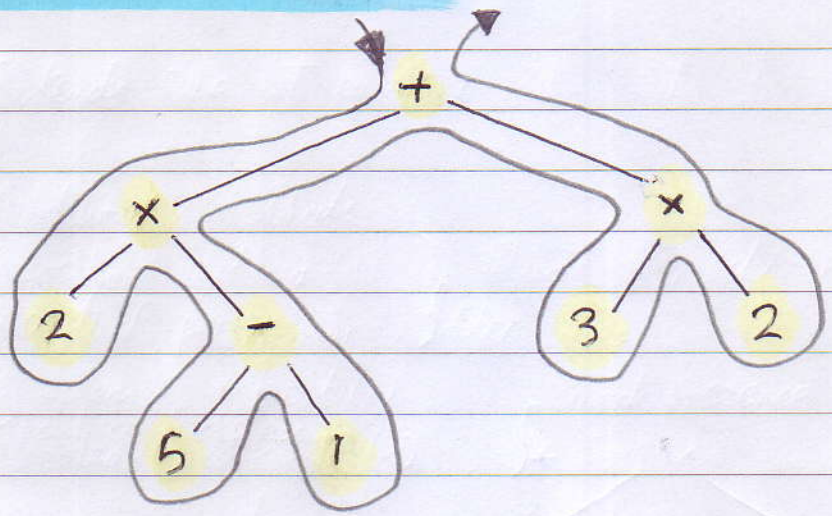


((2×(a-1))+(3×b))

- Evaluate an Expression

```
Algorithm evalExpr(T, v)
  if isExternal(v)
    return v.element()
  else
    x ← evalExpr(T, left(v))
    y ← evalExpr(T, right(v))
    ◇ ← symbol stored at v
    return x ◇ y.
```

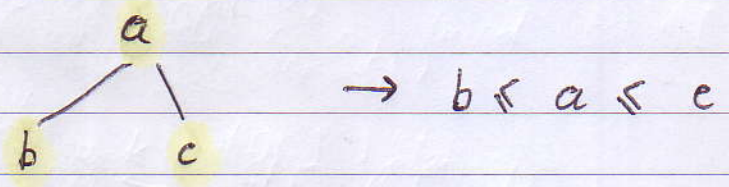
• Euler Tour Traversal



- Visits each node 3 times:
  - From the left
  - From below
  - From the right

• Binary Search Trees

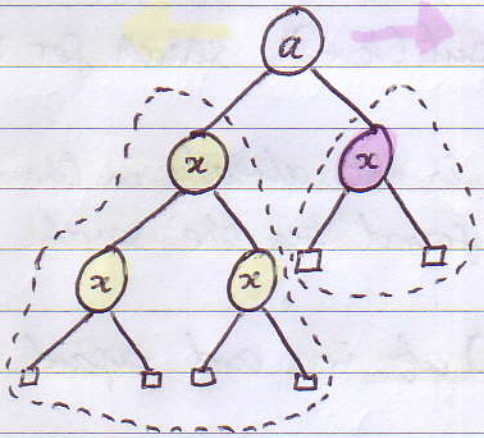
- A tree storing keys or key/value pairs in internal nodes
- External nodes do not store items



- Search:
  - start at root
  - compare each key visited with target key
  - if a leaf is reached, the key is not found

Search Tree Structures

- A binary search tree stores keys (or key/value pairs) at internal nodes.
- External nodes do not store items.
- In-order traversal visits keys in asc. order.
- Nodes are stored as follows:

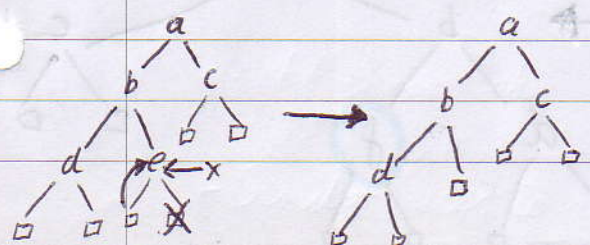


• ADT

- get(k)
- put(k, v)
- remove(k)

} The same as in the map ADT

- insertAtExternal(w, [k,v]) — inserts [k,v] at the external node w, and expands w to be internal
- removeExternal(w) — removes an external node w and its parent, replacing w's parent with w's sibling



• Search

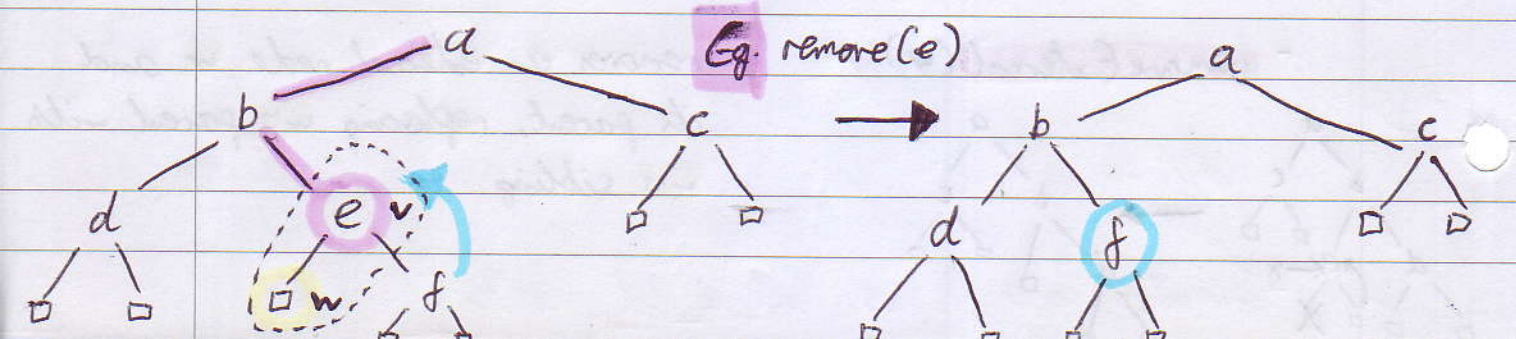
- To perform get(k), trace a downward path from the root.
- The next node visited depends on the comparison of k with the current node.
- If a leaf is reached, the key was not found.

• Insert

- To perform put(k,v), search for the key k.
- Assume k is not already in the tree, and let w be the leaf found by the search.
- Insert [k,v] at w, and expand w to be internal.

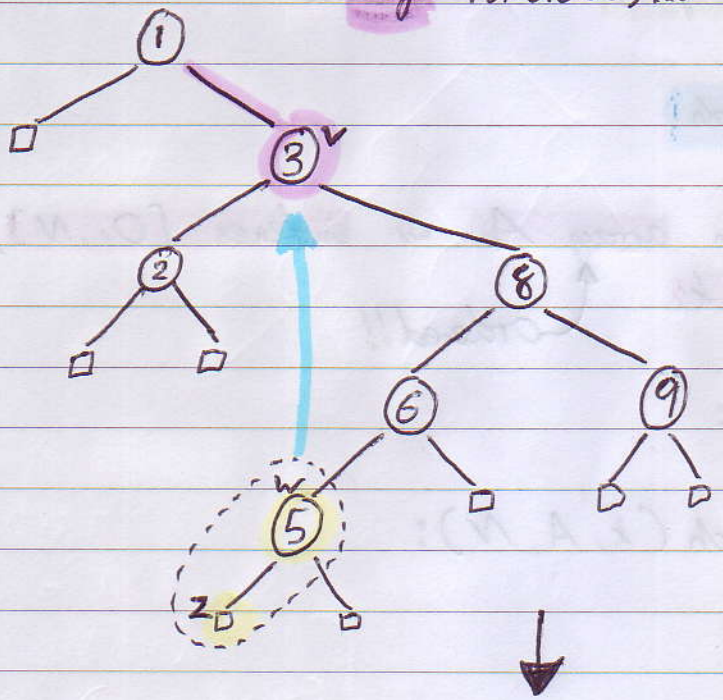
• Deletion

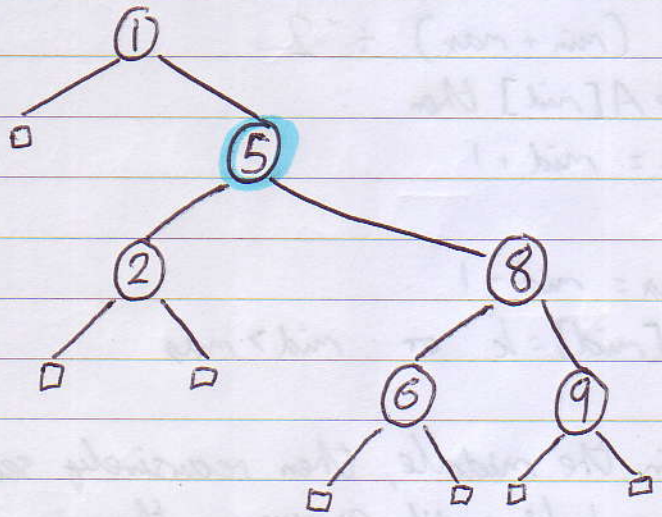
- To perform remove(k), search for the key k.
- Assume k is in the tree, and let v be the node storing k.
- If v has a leaf child w, we remove v and w with remove(w).



- If both children of v are internal...
  - Find the internal node, w, that follows v in an in-order traversal.
  - Copy w into v
  - Remove w and its left child, z (which will be a leaf) via removeExternal(z).

Eg. remove(3)...





• Performance

- Let tree height be h

- The space used is $O(n)$

- Get, put and remove take $O(h)$, assuming $O(1)$ is taken at each node.

- h is $O(n)$ in the worst case
- h is $O(\log n)$ in the best case.

• Binary Search

- Assume an array A w/ indexes $[0..N]$, searching for the key k. ← Ordered!!

- Pseudocode:

```
BinarySearch(k, A, N):
    min = 0
    max = N
    do
        mid = (min + max) ÷ 2
        if k > A[mid] then
            min = mid + 1
        else
            max = mid - 1
    until A[mid] = k or mid > max
```

TL;DR: Look in the middle, then recursively search the upper or lower halfs until success or there is nothing left to search.

Maps

- A searchable collection of key-value entry pairs.
- Multiple entries with the same key are not allowed.

• Map ADT

- Methods:

· get(k)	returns the value stored at the key k, or null if no entry has the key k
· ~~insert~~ put(k, v)	if there is no element with the key k, insert the entry (k, v) and return null. if there is an entry with the key k, update the value to v and return the old value.
· remove(k)	if the map has an entry with the key k, remove it and return the value, else ret. null
· entrySet()	returns an iterable collection of entries
· keySet()	" " " " " keys
· values()	an iterator of the values in the map
· size()	
· isEmpty()	

• List-Based Map

- A map can be efficiently stored in an unsorted list
  - Items are stored in any order on a DL-list
  - Each list node stores a key/value entry
- get, put and remove all work via straightforward list traversal

- Performance

  - get, put and remove all run in O(n) as the worse case requires that the node list is traversed.

  - Unsorted list implementations are effective for small maps where put operations are more common, where search and removal are less common (eg. a historical record of server log ins).

Hash Tables

- Used to implement a map.

- Two main parts:
  - **Bucket array**: array of size N, where each element is thought of as a bucket of key/value pairs
  - **Hash function**: maps keys of a certain type to integers in the range $[0, N-1]$.
    - the integer $h(x)$ is the hash value of the key x.

• Hash Functions

- Usually the composition of two functions:
  - **Hash Code**: h_1: keys → integers
  - **Compression Function**: h_2: integers → $[0, N-1]$

$$h(x) = h_2(h_1(x))$$

- Hash codes should avoid collisions

- The goal of a hash function is to disperse keys in a pseudo-random way.

- Sample Hash Codes:
  - **Memory address**: the memory address of the key object is reinterpreted as an integer. Not good for numeric/string keys.

- **Integer Cast:** the bits of the key are reinterpreted as an integer. Suitable for keys of length less than or equal to the number of bits of the integer type (byte, short, int, float in Java).
- **Component Sum:** the bits of the key are partitioned into components of fixed length and summed, ignoring overflow. Good for fixed-length numeric keys greater than the number of bits in the integer type.

- Sample Compression Functions

- A good compression function gives each bucket a $1/N$ probability of being selected for a given key.
- **Division:** $h_2(x) = x \bmod N$
  N is often chosen as prime
- **Multiply, Add, Divide (MAD)**
  - $h_2(x) = [(ax+b) \bmod p] \bmod N$
  - p is a prime greater than N
  - a and b are integers from $[0, p-1]$, $a>0$

• Collision Handling

- A collision occurs when two keys map to the same hash value.

- Separate Chaining

- Let each cell in the table point to a linked list of entries that map there.
- Simple, but requires extra space in memory.
- Operations are delegated to the list data type.

- Open Addressing - Linear Probing + Double Hashing.

- The colliding entry is placed in a different cell of the table.
- Each cell inspected is referred to as a probe.
- Colliding items lump together, so each subsequent collision consumes more time.

◦ Linear Probing:

- The colliding item is placed in the next (circularly) available cell.
- Search: start probing at index $h(k)$, then loop until an empty cell is found, or the whole array has been covered. (if the entry is not found)
- ~~The update method~~
- To handle insertions and deletions, a special object called 'AVAILABLE' is used to replace deletions.

◦ Double-Hashing:

- If $A[i]$ where $i = h(k)$ is occupied, we try...

$$A[(i + f(j)) \bmod N] \quad \text{for } j = 1, 2, 3, \ldots N-1$$

where $f(j) = j \times h'(k)$

- $h'(k)$ cannot give zero values
- N must be prime to allow for full probing.
- A common choice is:

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

where $q < N$ and q is prime.

• Hashing Performance

- In the worst case, when all keys collide, insertion, search and removal take $O(n)$ time.

- The load factor $a = n/N$ affects the performance

- Assuming a random distribution of hash values, the expected number of probes for insertion with open addressing is $\frac{1}{(1-a)}$

- Hashing is very fast, provided the load factor is not close to 100%.

Dictionaries

- The ADT models a searchable collection of key/value entries
- Main operations are searching, inserting and deleting.
- Multiple entries with the same key are allowed.

• Dictionary ADT

- get(k) — returns the first entry with the key k, or null if none exist
- getAll(k) — returns an iterable collection of all entries with the key k
- put(k, v) — inserts and returns the entry (k, v)
- remove(e) — removes and returns the entry e, or error if it does not exist in the dictionary
- entrySet() — returns an iterable collection of all entries
- size()
- isEmpty()

A dictionary can be implemented via a hash table w/ separate chaining

• Standard Performance

- put takes O(1) as the storage array/list is unsorted
- get/remove take O(n) as the worst case sees the whole list checked
- Similar uses to map, but a bit faster!

• Ordered Search Table.

- A dictionary implemented via a sorted array. (sorted by key).
- Performance:
  - get takes $O(\log n)$ using binary search
  - put takes $O(n)$ as the worst case is shifting $n/2$ items
  - remove takes $O(n)$ for the same reason.
- Useful for small data collections, or data where most actions are searches.

Stacks and Queues

- Both employ abstract data types (ADTs - see PRA notes).

• The Stack ADT

- The data structure consists of a sequence of elements, with one end designated as the top.

- Main operations (LI-FO principle):
  - Push (e) - add e at the top of the stack
  - Pop() - remove and return the top element
  - Top() - return the top element (don't remove)
  - Size()
  - isEmpty()

→ Throw an error if stack is empty (EmptyStackException)

- The Java Virtual Machine keeps track of active methods with a stack structure
  - When methods are called, they are pushed onto the stack, with their own program counter
  - When a method finishes, its frame is popped from the stack
  - This allows for recursion

- Array-Based Stack Implementation
  - Elements are added from left to right
  - A value tracks the index of the top element
  - Space used is O(n) for an n-sized stack
  - Operations run in time O(1)
  - Size must be predefined, and a FullStackException may occur.

• Parentheses Matching w/ Stacks

- ([{ must be properly matched

Algorithm ParenMatch(X, n)
- Input: An array X of n tokens, each of which is a mathematical operator, a number, a variable or a type of bracket.
- Output: true iff all brackets in X match

```
Let S be an empty stack.
For i=0 to n-1, do:
    if X[i] is an opening bracket then:
        S.push(X[i])
    else if X[i] is a closing bracket then:
        if S.isEmpty() then return false.
        if S.pop() does not match X[i] then return false
End for loop.
If S.isEmpty() then:
    return true
else
    return false
```

- A similar approach can be applied to HTML tag matching

• The Queue ADT

- Operates a FI-FO (First In First out) Scheme.
  - Insertions at the back of the queue, removals at the front.

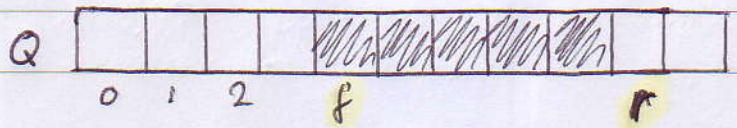
- Main operators:
  - enqueue(e) - adds e at the back of the queue
  - dequeue() - returns and removes the front element
  - front() - returns the front element without removing
  - size()
  - isEmpty()

- Throw an EmptyQueueException if called on an empty queue.

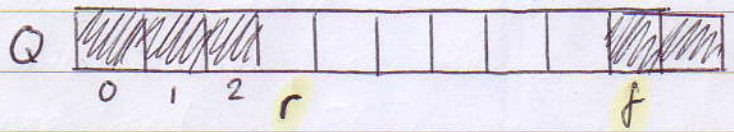
- Array-Based Queue Implementation

- Uses an array of size N in a circular fashion
- Variables keep track of the front and rear
  - f → index of the first element
  - r → index immediately past the rear element (empty)

- Normal config:



- Wrapped-Around Config:



- Operations:

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

  - isEmpty()
    return $(f == r)$

  - enqueue(e)
    if size() = n - 1
      throw FullQueueException
    else
      Q[r] ← e
      r ← (r+1) mod N

  - dequeue
    if isEmpty()
      throw EmptyStackException
    else
      e ← Q[f]
      f ← (f+1) mod N
      return e

Priority Queues and Heaps

- A priority queue is a collection of elements (values), each having a key that was provided at insertion
- An entry is a key/value pair
- keys determine an element's priority for removal.

• Keys

- keys are the parameter/properties by which objects are compared.
- keys in a priority queue can be any objects on which an order is defined.
- Multiple elements can have the same key!

• Priority Queue ADT

- Methods:
  - insert(k, x) — inserts the value x with the key k
    returns the inserted element
    error if k is not a valid key.
  - removeMin() — removes the element with the smallest key
    error if the queue is empty
  - min() — returns but does not remove, otherwise same function as removeMin()
  - size()
  - isEmpty()

• Entry ADT

- key/value pair

- Methods:
  - getKey()
  - getValue()

• Comparator ADT

- The object encapsulates the action of comparing two objects according to a given total order relation.

- The comparator is external to the keys being compared.

- A generic queue uses a default comparator.

- Methods:
  - compare(x, y)
    - ↳ returns an integer i such that:
      - $i < 0$ when $x < y$
      - $i = 0$ when $x = y$
      - $i > 0$ when $x > y$
  - equals(x)
    - ↳ compares a comparator to another comparator

Total Order:
- reflexive
- antisymmetric
- transitive

- Guarantees that comparison contradiction will not occur.

• Sequence-Based Priority Queues

Unsorted List	Sorted List
- insert takes O(1) as entries can go at the start or end.	- insert takes O(n) as one must find the place to insert the item.
- removeMin/min take O(n) as the entry must be found	- removeMin/min take O(1) as the lowest key is at the beginning

- size and isEmpty take O(1) for both.

• Priority List Sorting

1. Insert all elements into a PQ
2. Remove all elements with removeMin()
3. ???
4. Profit!

Running time depends on queue implementation

• Selection-Sort vs. Insertion Sort.

- Both implementations of PQ sorting, but....

• Selection-sort	• Insert-sort
- Uses an unsorted list	- Uses a sorted list
- n insert operations takes $O(n)$ time	- n insert operations take $O(n+(n-1)+\dots+2+1) = O(n^2)$
- n removeMin operations takes $O(n+(n-1)+(n-2)\dots 2+1) = O(n^2)$	- n removeMin operations take $O(n)$
- Runs in $O(n^2)$ time	- Runs in $O(n^2)$

• Heaps

- A binary tree storing entries at its nodes and satisfying:

  ◦ Heap-Order: for every non-root internal node, key(v) ≥ key(parent(v))

  ◦ Complete Binary Tree: if h is the height of the tree...
  - for $i = 0 \dots h-1$, there are 2^i nodes at depth i
  - at depth h, internal nodes are to the left of external nodes.

  ◦ Last Node: the last node of a heap is the rightmost node of maximum depth

• Heap Height

- A heap of n entries has a height of $O(\log n)$.

- Proof:
  - Using the complete binary tree property
  - Assume the height is h and we have n entries
  - There are always 2 keys at depths 0 to $h-1$ and at least 1 on depth h
  - So $n \geq 1 + 2 + 4 + 8 + \ldots + 2^{h-1} + 1$
  - $\therefore n \geq 2^h$
    ie. $h \leq \log n$

• Array Representation

- See array storage for binary trees, a few pages ago.

• Complete Binary Tree ADT

- Binary tree ADT, plus.....

- Methods:
  - `add(x)` add x to the tree and return a new ext. node v such that the resulting tree is a complete binary tree and v is the last node.
  - `remove()` remove ~~and~~ the last node of the tree and return its element.

• Heaps and PQs

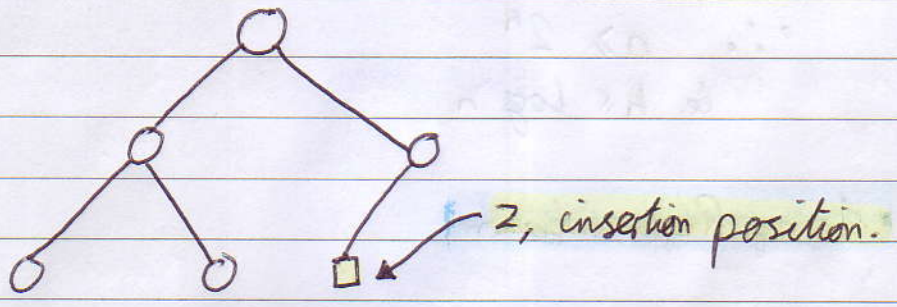
- Heaps can store a PQ: each node stores an entry, and the last node is tracked

• Insertion into a Heap. from the PQ ADT

- The insert(k, x) algorithm is as follows:

  - Add a node z to the tree with the add operation so that this node is the last node of the tree

  - Restore the heap-order property that may have been violated (see process after diagram).



- Up-Heap Bubbling:

  - Works to restore heap-order by swapping the inserted node on an upwards path

  - Stops when the new entry with key k reaches the root OR a node whose parent has a key ≤ k.

  - A heap has height of $O(\log n)$, so up-heap runs in $O(\log n)$ time.

• Removal from a Heap

- removeMin() removes the root of the tree. The algorithm has 3 steps:
  - Replace the root with the value in the last node w.
  - Remove w
  - Restore the heap-order property:

- Down-heap Bubbling:
  - Restores heap-order by swapping the node key k on a downwards path.
  - It swaps the element with key k with the child with the smallest key.
  - Stops when k reaches a leaf OR a node whose children have keys $\geq k$.
  - Heap height is $O(\log n)$, so down-heap runs in $O(\log n)$.

- Heap Sort:

  - Same principle as insert/selection sort.

  - insert and removeMin both take $O(\log n)$, so a heap-sort runs in $O(\log n)$.

  - Faster than quadratic sorts, like insert/selection sort.

$\lceil x \rceil$ = ceiling
$\lfloor x \rfloor$ = floor
$\lceil 1.2 \rceil = 2$
$\lfloor 2.9 \rfloor = 2$

Sorting Algorithms

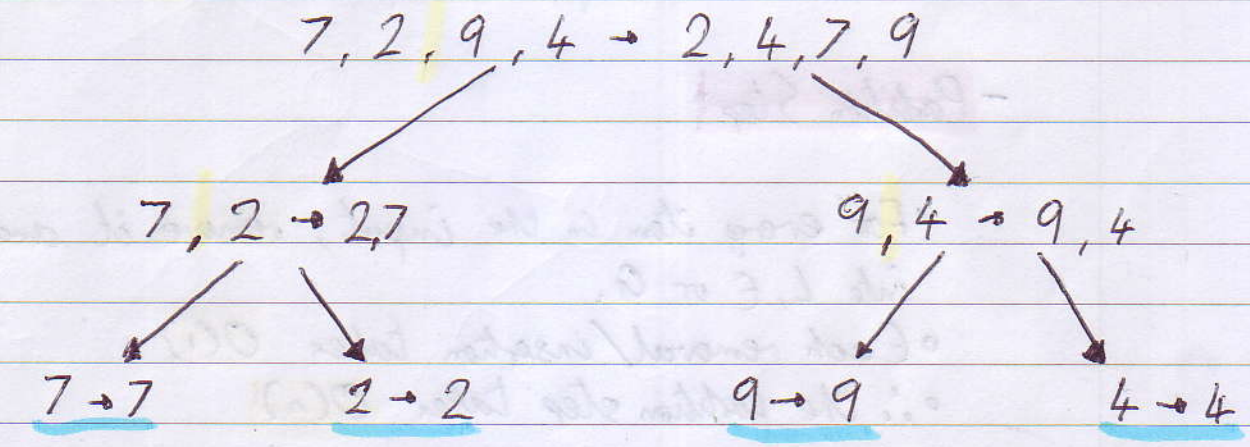
• Divide + Conquer

- A generic algorithm design pattern:
  - **Divide**: split a problem P into P_1 and P_2
  - **Recur**: solve P_1 and P_2
  - **Conquer**: recombine P_1 and P_2 to form a solution for P

• Merge Sort

- Algorithm for a sequence S with n elements:
  - **Divide**: split S into two lists - S_1 with the first $\lceil n/2 \rceil$ elements and S_2 with the last $\lfloor n/2 \rfloor$ elements.
  - **Recur**: recursively sort S_1 and S_2
  - **Conquer**: merge S_1 and S_2 into the sorted sequence S
  - **Base case**: subproblems of size 0 or 1.

Eg: Merge sort tree



= base cases.

• Performance of Merge Sort

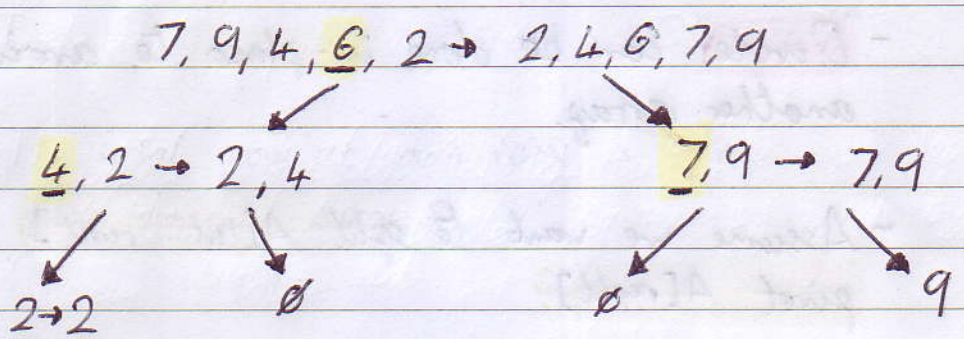
- The height h of the tree is $O(\log n)$
- The total work at each node of depth i is $O(n)$
  - Partition and merge 2^i sequences of $n/2^i$ length
  - 2^{i+1} recursive calls.
- Thus, total running time is $O(n \log n)$

• Quick Sort

- Algorithm:
  - Divide: pick an element x to be the pivot.
    Split S into 3 lists:
    - L elements $< x$
    - E elements $= x$
    - G element $> x$

    ↑ often chosen as the last element.
  - Recur: recursively sort L and G.
  - Conquer: recombine L, then E, then G.
  - Base cases: subproblems of size 0 or 1
- Partition Step:
  - For every item in the input, remove it and insert into L, E or G.
  - Each removal/insertion takes $O(1)$
  - ∴ the partition step takes $O(n)$

- Quick-Sort Tree



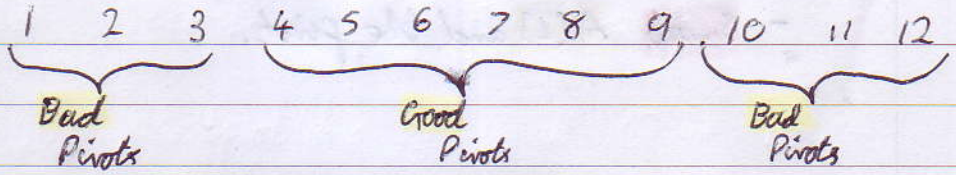
• Performance of Quick Sort

- Worst Case

- When the pivot each loop is the unique min or max.
- One of L / G has size is n-1, the other is 0.
- The running time is proportional to $n + (n-1) + \ldots + 2 + 1$
- ∴ the running time is $O(n^2)$

- Expected Performance

- Good : L and G are each less than $\frac{3}{4}n$
- Bad : L or G is bigger than $\frac{3}{4}n$
- Prob. of a good pivot choice is $\frac{1}{2}$.



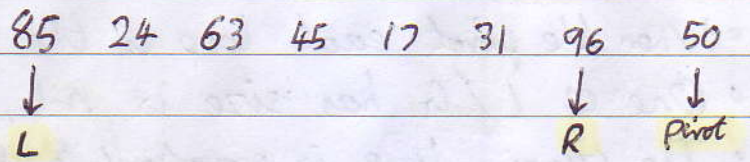
- For a node at depth i, we expect...
  - $i/2$ ancestors had good pivots
  - the size of the input is at most $(3/4)^{i/2} n$
- The expected height of the tree is $O(\log n)$.
- For a node at depth $2\log_{4/3} n$, the input size is 1
- The total work at each node is $O(n)$.
- So the run time is $O(n \log n)$.

• In-Place Variation of Quick Sort

- "Divide" can be done in-place to avoid creating another array.

- Assume we want to split A[left ... right] based on the pivot A[right]

- Maintain two pointers, L and R, for L=left and R=right-1
  - The unchecked elements are in A[(L+1) ... (R-1)]

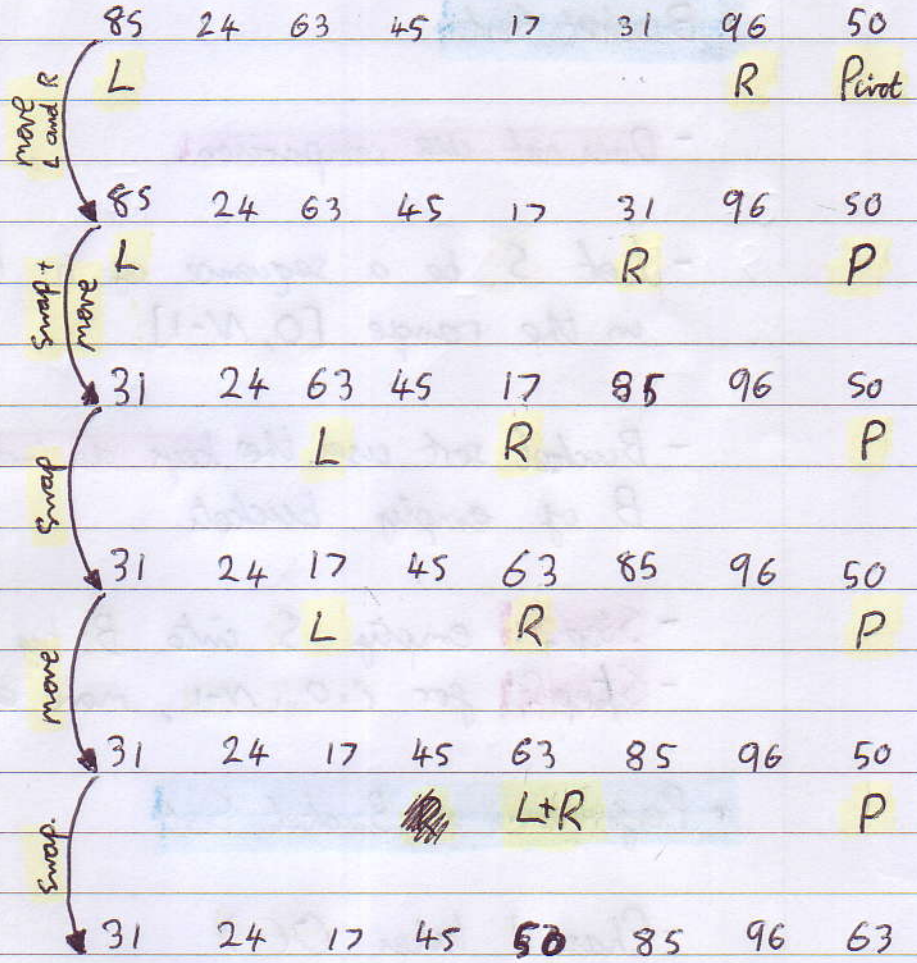


- Loop while L <= R:
  - keep increasing L by 1 until A[L] is larger than the pivot and L<=R.
  - keep decreasing R by 1 until A[R] is smaller than the pivot and L<=R.
  - If L<R swap A[L] and A[R] and proceed to the next loop.
- Swap A[L] and the pivot.

Example Over There →

Iteration 1:

- no move for L
- R moves down 1
- swap.



L>R so swap A[L]/pivot

• Bucket Sort

- Does not use comparison.

- Let S be a sequence of n key/val entries with keys in the range $[0, N-1]$

- Bucket sort uses the keys as indicies into an auxiliary array B of empty buckets

- Step 1: empty S into B by putting each (k, v) in $B[k]$
- Step 2: for $i = 0 \dots N-1$, move entries of $B[i]$ to the end of S.

• Performance of Bucket Sort

- Phase 1 takes $O(n)$
- Phase 2 takes $O(n+N)$

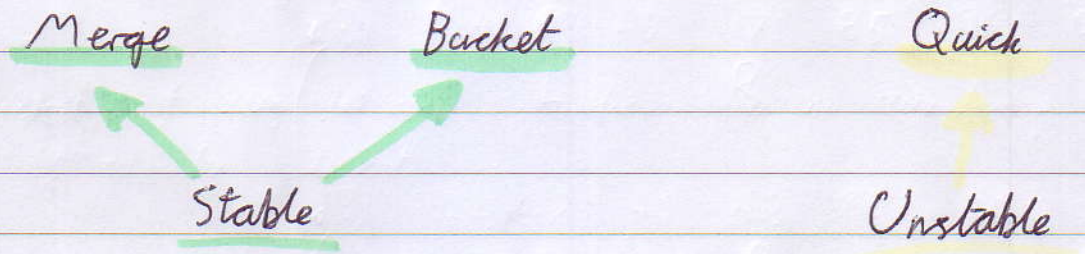
- Bucket sort takes $O(n+N)$

• Sort Summary

	Time	Notes
Selection Insertion	$O(n^2)$	• in-place • slow (okay for small inputs)
Quick	$O(n \log n)$ expected	• in-place • fast (good for large inputs)
Heap	$O(n \log n)$	
Merge	$O(n \log n)$	• sequential data access • fast (good for huge inputs)

• Stability of Sorting

- A sort algorithm is stable if two items with the same key are in the same order in the input as they are in the output.



• Sorting Lower Bound.

- Many sorts are comparison based
- Comparisons can be shown as a decision tree
- The height of this tree is a lower bound on run time
  - As there are $1 \times 2 \times \ldots \times n$ leaves, the height is at least $\log(n!)$.
- Any comparison-based sort takes at least $\log(n!)$ time.
- ∴ any such algorithm takes time at least...

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

- So any comparison-based sort runs in $\Omega(n \log n)$ at best.