COMP3506 NOTES

$\sum_{i=0}^{h} a + id = \frac{n}{2}(2a + hd)$

$\sum_{i=0}^{h} ar^i = \frac{a(r^h - 1)}{r - 1}$ (when $|r| > 1$)

$= \frac{a(1 - r^h)}{1 - r}$ (when $|r| < 1$)

$\sum_{i=0}^{\infty} ar^i = \frac{a}{r - 1}$ (when $|r| < 1$)

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_c a)/\log_c b$
5. $b^{\log_c a} = a^{\log_c b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a / b^c = b^{a-c}$

Week 1

ADT – abstract data type
- Specifies type of data stored
- Specifies supported operations
- Specifies types of parameters for operations
- What it does, not how it does it
- Java Interfaces

CDT – concrete data type
- Implementation of an ADT

Arrays
- Data structure consisting of a group of elements having a single name that are accessed by indexing
- Occupies a contiguous area of storage - Provides constant-time access to an indexed memory location
- Each element has the same data type

Week 1 & 2 – analysis

Data structure – systematic way of organizing and accessing data Algorithm – procedure for performing a task

Complexity

Experimental analysis
- Need to implement algorithm – may be difficult
- Comparing requires same hardware and software environment
- Not indicative of run times of other inputs

Theoretical Analysis
- Use high level description of algorithm
- Function of input size *n*
- Takes into account all possible inputs (including bad ones)
- Independent of software and hardware

How to analyse
- Express algorithm as pseudo-code
- Count primitive operations
- Describe algorithm as f(n)
- Perform **asymptotic analysis** describing limiting behaviour

Big-O notation describes an upper bound on a function f(n) is O(g(n)) if f(n) is asymptotically greater than or equal to g(n)

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 $f(n) \le c \cdot g(n)$ for $n \ge n_0$

Big-Omega Notation describes a lower bound on a function

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that $f(n) \ge c \cdot g(n)$ for all $n \ge n_0$

$f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$

Big-Theta notation describes a tight bound on a function (if one exists) if f(n) is asymptotically equal to g(n) — exist if the (big-O and big-Ω) are equal

Important, as practical experimental runtime can differ depending on hardware used! Thus, best to use a theoretical approach to compare algorithms.

Pseudo-Code Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration
  - **Algorithm** *method* (*arg* [, *arg*…])
  - **Input** …
  - **Output** …
- Method call
  - *method* (*arg* [, *arg*…])
- Return value
  - **return** *expression*
- Mathematical formatting
  - n^2
- Expressions
  - ← (Assignment) Like "=" in Java
  - = (Equality testing) Like "==" in Java

Week 2: Recursion

Express the following recurrence in big-O notation
T(n) = O(1) if n = 1; O(1) + T(n/3) otherwise
T(n) = O(1) + T(n/3) = O(1) + O(1) + T(n/9) = O(1) + O(1) + T(n/27) in O(log3n) base:3

Recursion: when a method calls itself
- Test for **base cases**
- Begin by testing for a set of base cases there should be at least one
- Every possible chain of recursive calls must eventually reach a base case
- Handle of each base case should not use recursion

Linear Recursion
- Perform a single recursive call
- Define each possible recursive call so that it makes progress towards a base case

Tail Recursion
- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step
- Easily converted into iterative forms

Binary Recursion
- Binary recursion occurs whenever there are two calls for each non-base case

Multiple Recursion
- It makes potentially many recursive calls not just one or two
- Multiple recursion is a way of enumerating all possible combinations of a set of elements

Divide and Conquer

Binary search uses the Divide and Conquer paradigm
- Divide - Break the problem into smaller subproblems of the same type
- Recur - Recursively solve these subproblems
- Conquer - Combine the solutions for the subproblems into a solution for the problem itself

Recursion Activity
- Use recursion to design an algorithm that sorts an array of n integers
- We will call this **Selection Sort**

Running time of *selectionSort*

$T(n) = O(n) + T(n-1)$
$= O(n) + O(n-1) + T(n-2)$
$= O(n) + O(n-1) + O(n-2) + T(n-3)$
…

n decreases by one each call, so there will be *n* recursive calls in total

$T(n) = 1 + 2 + 3 + \dots + n - 1 + n$
$T(n) = n(n+1)/2$
$T(n)$ is $O(n^2)$

Analysis of Binary Search

- We can express the running time of *binarySearch* as

$T(n) = \begin{cases} O(1) & \text{if } n = 0 \\ T(n/2) + O(1) & \text{if } n > 0 \end{cases}$

- The input size is halved on each recursive call
- At most $\log_2 n$ elements are searched
- Therefore the running time is $O(\log n)$

Algorithm *printIntegers(n)*
Input: an integer, n, representing the number of integers to print
Output: printed integers from 0 to n − 1 (no return value)

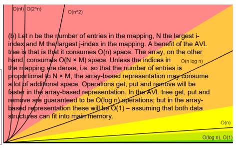
i ← 0
while i < n **do**
  println i

Sol: It is clear that the loop in printIntegers will iterate n times. We cannot make the assumption that printing has O(1) time complexity, as printing an integer is proportional to the number of digits. We can write the number of digits in a single number, k, as d(k) = floor(log10 k) + 1. Since d(k) is clearly O(log n), the overall running time of printIntegers is O(n log n).

$T(n) = \begin{cases} 3T(n/2) + O(n) & \text{if } n > 1 \\ O(1) & \text{if } n = 1 \end{cases}$

For example, the (2 × 1)-dimensional array A, represented pictorially below, could be used to represent the mapping {(0,0), v), ((1,0), V), ((1,2), u)}, where u, h and v are elements of the set V.

(a) Describe how an ADT tree representation could be used (efficiently) to implement mappings from pairs of non-negative integers to values.
(a) Each entry ((i, j), v) can be stored in the AVL tree with key (i, j) and value v. Keys can be ordered lexicographically (i.e., order by i first, then order by j if there are equal values of i)
(b) Compare the array-based implementation (described above) to the AVL tree implementation from (a). Identify the benefits and drawbacks of using each representation.



(b) Let n be the number of entries in the mapping, N the largest i-index and M the largest j-index in the mapping. A benefit of the AVL tree is that is that it consumes O(n) space. The array, on the other hand, consumes O(N × M) space. Unless the indices in the mapping are dense, i.e. so that the number of entries is proportional to N × M, the array-based representation may consume a lot of additional space. Operations get, put and remove will be faster in the array-based representation. In the AVL tree get, put and remove are guaranteed to be O(log n) operations; but in the array-based representation these will be O(1) – assuming that both data structures can fit into main memory.

Algorithm *insertionSort*(A, n)
for i ← 1 to n − 1 do
  key ← A[i]
  j ← i − 1
  while j >= 0 and A[j] > key do
    A[j + 1] ← A[j]
    j ← j − 1
  A[j + 1] ← key

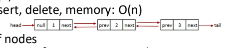
O(n) worst-case, O(1) best-case

Weeks 3&4 – Linear structures

Stacks, Queues, Lists, Vectors, Sequences
Implemented by arrays or linked data structures

Array
- Consecutive block of memory
- Size specified at creation
- Expensive to resize
- Access: O(1)
- Iterate, insert, delete, memory: O(n)

Linked list



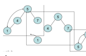
- Sequence of nodes
- Each node stores reference to next node
- Head – first node Tail – last node
- Access, insert, delete: O(1) Iterate, memory: O(n)
- In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Stack:
- LIFO / FILO
- push(T) – insert element
- T pop() – remove and return last inserted element
- T top() – return last inserted element, don't remove
- Array stack – keep track of last element
- O(1) operations
- Max size defined at creation

List:
- integer size() – number of elements in list
- T get(i) – get element at index i
- T set(i,e) – sets element at index I, returns old value
- add(i,e) – inserts at i, moves subsequent elements
- remove(i) – remove and return element at index i

Queue:
- FIFO
- enqueue (T) – insert element at end of queue
- T dequeue() – remove and return element at front of queue
- T first() – return front element, don't remove
- attempt to dequeue empty –null or NoSuchElementException
- Array based queue – array of size N in circular fashion
- keep track of start and end of queue
- O(1) operations
- Max size defined at creation

ArrayList:



- store list as array A
- element i at A[i]
- add, remove O(n) as may need to shift entire list

Growing arrays:
- Incremental
  - n push operations, replaced k = n/c times
  - T(n) = n + c + 2c + … + kc
  - T(n) = n + ck(k+1)/2
  - T(n) = O(n^2)
  - amortised to O(n) insert
- Doubling
  - n push operations, replaced k = log(n) times
  - T(n) = n + 1 + 2 + … + 2^k
  - T(n) = n + 2^{k+1} -1
  - T(n) = 3n-1
  - T(n) = O(n)
  - amortised to O(1) insert

Positional list

sequence of elements, ability to identify location of each
P – position class, only has P.getElement()
P first() – returns first position of list (or null)
P last() – returns last position of list (or null)
before(p) – returns position immediately before p (or null)
after(p) – returns position immediately after p (or null)
P addFirst(e) – add at start, return position
P addLast(e) – add at end, return position
P addBefore(p, e) – add e before p, return position
P addAfter(p, e) – add e after p, return position
e set(p, e) – replace elements at p, return old one
e remove(p) – remove and return element at p
obvious implementation is doubly-linked list

Q: Why might it be preferable to use a **doubly-linked list** (with head and tail pointers) in preference to a **singly linked list** (with head and tail pointers) to **implement** a **Position List?**
Sol: A DLL has the advantage of being able to iterate in reverse. It also allows constant-time removal provided we have a reference to the node being removed. A SLL, while occupying less memory and having slightly faster insert operations, cannot be iterated over in reverse efficiently, and has linear-time removal.

Q: **Choose a sorting algorithm that could be used to sort n decks of cards. Justify your choice.**
Sol: A bucket sort would likely be appropriate for this scenario. We know exactly what cards will be in the deck, so we can create 52 buckets (one for each card) and perform this sort in O(n) time complexity

Iterators
- Abstraction of scanning through a data structure
- boolean hasNext() – true if next element exists
- T next() – get next element

Week 4 – Trees

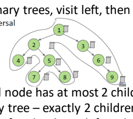
Abstract model of hierarchical data structure
Consists of nodes with parent-children relationships

Ex. How many arrangements of five nodes exist that form a proper binary tree? -- Sol: 2 since tree have 2 children and 2 since tree have 2 children and either left or right have 2 children

Terminology:
- root – node without parent
- internal node – at least one child
- external node / leaf – no children
- edge – link between nodes
- ancestor – parent, grandparent, etc.
- descendent – child, grandchild, etc.
- sibling – child of same parent
- degree – number of children
- level – root level 0, children of root level 1, …
- depth of node / tree – number of descendants from root
- height of node / tree – number of ancestors from leaf
- subtree – tree consisting of node + descendants
- k-ary tree – each internal node has at most k children
- Euler Tour Traversal - visit each node three times

Tree traversal
- Systematically visit every node in tree
- preorder – node is visited before its descendants
- postorder – node is visited after its descendants
- in order – binary trees, visit left, then node, then right

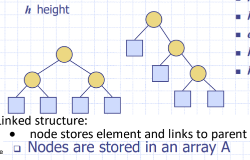
Example: Post-Order Traversal
4 7 8 5 2 9 6 3 1



Binary trees
- each internal node has at most 2 children
- proper binary tree – exactly 2 children for each internal node
- ordered pair of nodes, has a left and right child
- full level – level n is full if it contains 2^n nodes
- complete binary tree – every level except bottom one is full
- complete binary tree – leaf nodes as far left as possible
- proper/full binary tree – every level is full

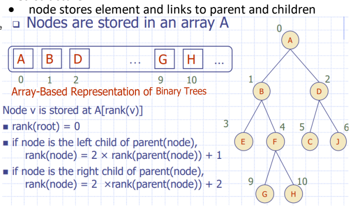
Properties of Proper Binary Trees

- Notation
  - *n* number of nodes
  - *e* number of external nodes
  - *i* number of internal nodes
  - *h* height
- Properties (no. external node of proper tree is (n+1)/2)
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \le i$
  - $h \le (n - 1) \div 2$
  - $e \le 2^h$
  - $h \ge \log_2 e$
  - $h \ge \log_2 (n + 1) - 1$



Linked structure:
- node stores element and links to parent and children

- Nodes are stored in an array A



Array-Based Representation of Binary Trees

Node v is stored at A[rank(v)]
- rank(root) = 0
- if node is the left child of parent(node), rank(node) = 2 × rank(parent(node)) + 1
- if node is the right child of parent(node), rank(node) = 2 ×rank(parent(node)) + 2

Week 5 – PQs

Priority Queues
- stores collection of entries
- entry is key value pair - getKey and getValue
- insert(k,v) inserts entry with key k, value v
- removeMin() remove and return entry with smallest key
- min() return but don't remove entry with smallest key
- priority assessed using a comparator
- keys can be arbitrary objects on which an order is defined
- distinct entries can have the same key
- total order relation: comparabilitiy, antisymmetric, transitivity
- finite set of key values implies a minimum
- compare entries with comparator

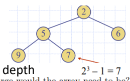
Comparator ADT

- Generic priority queue uses an auxiliary comparator
- Encapsulates the action of comparing two objects according to a given total order relation
- Comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator
- compare(x, y) – returns an integer *i* such that
  - *i* < 0 if a < b,
  - *i* = 0 if a = b
  - *i* > 0 if a > b
  - error occurs if a and b cannot be compared

Sequence based PQ
- sorted list – insert O(n), min, remove O(1)
- unsorted list – insert O(1), min, remove O(n)

Can use PQ to sort (insert all, remove smallest one by one)
selection sort – unsorted list PQ O(n^2)
insert sort – sorted list PQ O(n^2)
In-place insertion sort – initial portion of sequence is sorted use swaps instead of modifying sequence

HEAP



- binary tree, storing keys at nodes
- **heap-order** – children >= parent
- **complete binary tree**
- **last node** is rightmost node of max depth ($2^3 - 1 = 7$ how large would the array need to be?)
- height O(log(n))
- insertion: add to the last node,upheap to restore heap-order
- upheap – swap key upwards until larger than parent or root
- removal – swap root with last node. remove last node
- down heap – swap downwards with smaller child until larger than children or leaf

Insertion node can be found by traversing a path of $O(\log n)$ nodes
- go up until node you are traversing from is the left child of its parent, or the root is reached
- from that node go to its right child
- go down left until a leaf is reached

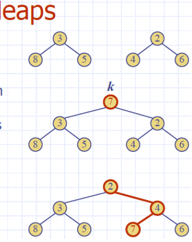
Updating the Last Node

- in array based heap, last element is last in array

Heap Sort
- PQ sort based around heap. O(nlog(n))
- Can construct heap in O(n) time – bottom up

Merging Two Heaps

- Given two heaps and a key *k*
- Create a new heap with the root node storing *k* and with the two heaps as subtrees
- Perform downheap to restore the heap-order property



Adaptable PQs
- Heap Implementation
  - replaceValue(e, x) takes constant time
  - remove(e) and replaceKey(e, k) run in O(log n) time
  - Location-aware entries increase the running time of insert and removeMin by constant factors
- List Implementation
  - remove(e) and replaceKey(e, x) take constant time
  - replaceKey(e, k) runs in O(n) time
  - Location-aware entries increase the running time by a constant factor

Performance

- Improved times thanks to location-aware entries are highlighted in red

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

Q: Are **hash tables** a suitable **implementation** of an **Ordered Map?**
Sol: An ordered map also has funcs like: Successor(k) - gets the key with value greater than or equal to k
Predecessor(k) - gets the key with value less than or equal to k
Rank(k) - number of keys less than k
Hash tables **won't** be able to provide these **efficiently** as we need to check every other key, resulting in linear performance.

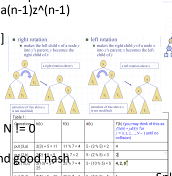
Week 5 & 6 – maps and sets

Map
- searchable collection of key-value entries
- duplicate keys not allowed
- don't need total ordering, but need equality test
- list based map – put, get and remove take O(n) time

Hash table – expected O(1), worst O(n)
- map keys to indices using hash function for [0, N-1]
- hash table has hash function and bucket array of size N
- hash function split into hash and compression

Hash codes h1: keys → integers
- memory address
- integer cast – reinterpret bits of key as integer
- component sum
- polynomial accumulation
  - partition key into sequence of components
  - evaluate p(z) = a0 + a1z + … + a(n-1)z^(n-1)
  - suitable for strings

Compression h2: integers → [0, N − 1]
- division – h(y) = y mod N
  - N should be prime
- Multiply add and divide
  - h(y) = ((ay+b) mod p) mod N
  - p prime > N
  - a, b random in [0, p-1], a mod N != 0



Collisions
- want low load factor (n/N < 0.5)

Collision handling
- separate chaining – linked list for each bucket
- Open addressing: colliding item is placed in a different cell of the table
- linear probing – place in next available bucket
  - Use DEFUNCT object when deleting
- Quadratic Probing -- Probe index calculated by quadratic function
  - Avoids linear probing clustering patterns
    - has secondary clustering issues
    - filled cells are not uniformly distributed
- Double hash – secondary hash function d(k)
- place in first h(k) + j*d(k) that's empty
- E # probes = 1/(1-α)

Set – unordered collection of elements, no duplicates
- Storing a Set in a List: Space used is O(n)

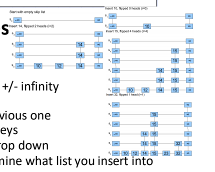
Multiset (bag) – like set, allows duplicates
Mutimap – key can map to multiple values
- can implement using map, storing list at each entry

Generic Merging

- Generalised merge of two sorted lists *A* and *B*
- Auxiliary methods
  - alsLess
  - blsLess
  - bothAreEqual
- Runs in $O(n_A + n_B)$ time
  - if the auxiliary methods run in O(1) time

Algorithm *genericMerge*(*A*, *B*)
S ← empty sequence
while ¬*A*.isEmpty() ∧ ¬*B*.isEmpty()
  a ← *A*.first().element(); *b* ← *B*.first().element()
  if *a* < *b*
    alsLess(*a*, *S*); *A*.remove(*A*.first())
  else if *b* < *a*
    blsLess(*b*, *S*); *B*.remove(*B*.first())
  else { *b* = *a* }
    bothAreEqual(*a*, *b*, *S*)
    A.remove(*A*.first()); *B*.remove(*B*.first())
while ¬*A*.isEmpty()
  alsLess(*a*, *S*); *A*.remove(*A*.first())
while ¬*B*.isEmpty()
  blsLess(*b*, *S*); *B*.remove(*B*.first())
return *S*

Week 9 – Search trees



Skip-list
- series of lists such that
  - each list contains special keys +/- infinity
  - each list is sorted
  - each list is a subset of the previous one
  - top list only contains special keys
- to search – scan forward and drop down
- perform "coin tosses" to determine what list you insert into
- low probability of having lots of lists
  remove element – delete additional empty lists
- Can implement with quad-node – stores links to prev, next, node, above, node below and entry
- Expected space: O(n), expected height O(log(n))
- Search expected to take O(log(n))

- Binary search – at each step, number of candidates is halved
- search - O(log(n)), remove and insert O(n)

Search Tables
An ordered map implemented by means of a sorted list

Search Tables: Performance

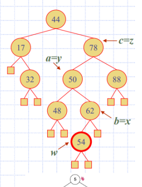
- Searches take $O(\log n)$ time
  - using binary search
- Inserting a new item takes $O(n)$ time
  - on average we expect to have to shift $n \div 2$ items to make room for the new item
  - worst case we have to shift *n* items
- Removing an item takes $O(n)$ time
  - on average we expect to have to shift $n \div 2$ items
  - worst case we have to shift *n* items

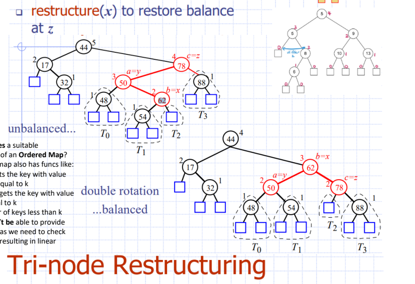
Binary search tree
- key-value entries at internal nodes
- left child <= parent <= right child
- external nodes do not store items
- in-order traversal visits nodes in increasing order
- search – go left or right depending on comparison with current key, if leaf reached, key not found
- insert – assume key not in tree, insert at leaf node reached in search for key
- delete – if node has leaf child easy. else replace with next node in in-order traversal
- space – O(n)
- get, put, remove take O(height) time, (n worst, log(n) best)

AVL trees
- Balanced – heights of children can differ by at most 1
- height of an AVL tree storing n keys is O(log n)

Insertion in an AVL Tree

- Insertion is as in a binary search tree, plus rebalance
- *z* is the first unbalanced node encountered while traveling up the tree from *w*
- *y* is the child of *z* with the larger height
- *x* is the child of *y* with the larger height
- restructure(x) to restore balance at *z*

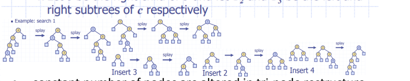




Tri-node Restructuring

Algorithm *restructure*(*x*)
Input Node *x* of a binary search tree *T* that has both a parent *y* and a grandparent *z*
Output Tree *T* after a tri-node restructuring (single or double rotation) involving nodes *x*, *y* and *z*

1. Let (*a*, *b*, *c*) be a left-to-right (in-order) listing of the nodes *x*, *y* and *z* and let (T_0, T_1, T_2, T_3) be a left-to-right listing of the 4 subtrees rooted at (*x*, *y*, *z*)
2. Replace the subtree rooted at *z* with a new subtree rooted at *b*
3. Let *a* be the left child of *b* and let T_0 and T_1 be the left and right subtrees of *a* respectively
4. Let *c* be the right child of *b* and let T_2 and T_3 be the left and right subtrees of *c* respectively

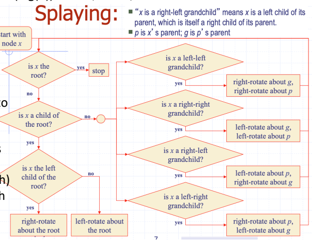


- constant number of nodes are altered in tri-node restructure
- removal requires rebalancing – have to rebalance all the way to the root
- Single restructuring takes O(1) time using a linked-structure binary tree
- O(n) space, O(log(n)) search, insert, remove

Splaying:

■ "*x* is a right-left grandchild" means *x* is a left child of its parent, which is itself a right child of its parent.
■ *p* is *x*'s parent; *g* is *p*'s parent

Splay tree
- binary search tree, no upper height bound on upper height
- worst case time O(n), amortised O(log(n))
- Splaying – move node to root following an operation on it
- splay by rotating nodes up the tree
- splaying costs O(h) - O(h) rotations, each of which is O(1)



Multi-way tree

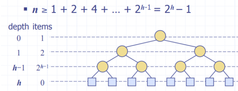
- each node stores d-1 key-element items (d children, d> 2)
- leaves store nothing, serve as placeholders
- children ordered between keys in parent
- in-order traversal can be extended to multi-way trees
- multi-way searching similar to binary searching – reaching leaf terminates search unsuccessfully

(2,4) tree

- multi-way tree, each node has 2-4 children
- Node-Size Property: internal node has at most four children
- Depth Property: all external nodes have same depth
- insert at parent of leaf reached by searching for k – may overflow. insert takes O(log(n))

Height of a (2,4) Tree

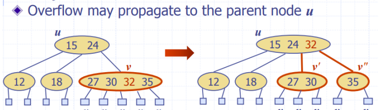
- Since there are at least 2^i items at depth i and no items at depth h, we have
  - $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$



- Thus $h \leq \log(n+1)$
- Searching takes $O(\log n)$ time

Overflow and Split

- Handle an overflow at a 5-node v with a split operation
  - let $v_1 \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
  - node v is replaced nodes v' and v''
    - v' is a 3-node with keys $k_1\ k_2$ and children $v_1\ v_2\ v_3$
    - v'' is a 2-node with key k_4 and children $v_4\ v_5$
  - k_3 is inserted into the parent u of v (a new root may be created)
- Overflow may propagate to the parent node u



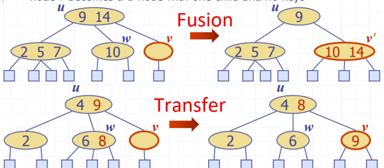
- if root overflows, add new root above existing one
- delete – if leaf, just remove. otherwise, replace by in-order successor. O(log(n))

Analysis of Deletion

- Let T be a (2,4) tree with n items
  - $O(\log n)$ height
- In a deletion operation
  - visit $O(\log n)$ nodes to locate the node from which to delete the entry
  - handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

Underflow and Fusion

- Deleting an entry from a node v may cause an underflow, where node v becomes a 1-node with one child and no keys

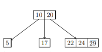


Comparison of Map Implementations

	Search	Insert	Delete	Notes
Hash Table	1 expected	1 expected	1 expected	no ordered map methods; simple to implement
Skip List	log n high prob.	log n high prob.	log n high prob.	randomised insertion; simple to implement
AVL and (2,4) Tree	log n worst-case	log n worst-case	log n worst-case	complex to implement

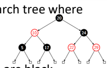
B-trees - Version of the **(a,b) tree** data structure, best-known method for maintaining a map in external memory
- B-tree of **order d** is an (a,b) tree with a = d÷2 and b = d
- external memory split into disk blocks.
- want to minimize disk transfer – I/O complexity
- (a,b) tree has between a and b children for each node where 2 ≤ a ≤ (b+1)÷2 and All external nodes have the same depth
- if parameters set appropriately, can get good disk performance operations similar to (2,4) tree
- each operation only requires one disk access at each level of tree
- **Height of an (a,b) tree** storing n entries

- Ω(log n ÷ log b), which is Ω(logb n)
- O(log n ÷ log a), which is O(loga n)



Red-black trees

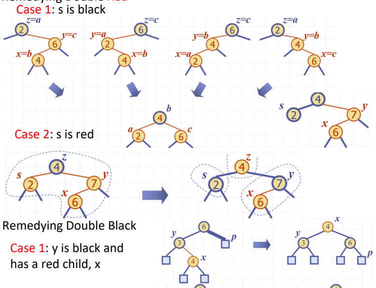
- representation of (2,4) tree by binary search tree where each node is red or black
- **Root property**: root is black
- **external property**: every leaf is black
- **internal property**: children of a red node are black
- **depth property**: all leaves have the same black depth
- O(log(n)) height as at most twice height of (2,4)



Remedying Double Red

Case 1: s is black

Case 2: s is red

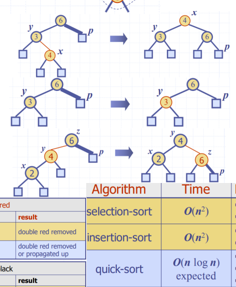


Remedying Double Black

Case 1: y is black and has a red child, x

Case 2: y is black and its children are both black

Case 3: y is red



- Deletion in a red-black tree takes O(log n) time

Insertion	remedy double red	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	change of 4-node representation	double red removed
recolouring	split	double red removed or propagated up

Deletion	remedy double black	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	transfer	double black removed
recolouring	fusion	double black removed or propagated up
adjustment	change of 3-node representation	restructuring or recolouring follows

Week 2 & 3 & 10 – Sorting and selection

E.g Give a list of 4 numbers that exhibits the best-case runtime when sorted using insertion sort - Sol: 1,2,3,4

Inserting sort process

- Select the first unsorted element
- Swap other elements to the right to create the connect position and shift the unsorted element
- Advance the marker to the right one element
- Worst-case (array is in descending order): O(n^2)
- Best-case (array is already sorted): O(n)

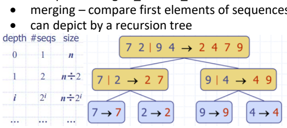
Selection Sort Process

- Scan each element of the array – find the largest (emax)
- Swap emax with the last element of the array
- Repeat this process on the first n - 1 elements

Merge sort

$$T(n) = \begin{cases} O(1) & \text{if } n < 2 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases}$$

- Guaranteed O(n log n) running time
- accesses data in sequential manner –good for sorting on disk
- Divide – split sequence S in half
- Recur – recursively sort S_1 and S_2
- Conquer – merge S_1 and S_2
- merging – compare first elements of sequences
- can depict by a recursion tree



- Height h of the merge-sort tree is O(log n)
- Overall amount of work done at nodes of depth i is O(n)
- Thus, the total running time of merge-sort is O(n log n)

Algorithm *merge(A, l, m r)*
Input an array A with two sorted halves
Output sorted union of $A[l..m]$ and $A[m..r]$

```
n1 ← m – l + 1      // size of first half of A
n2 ← r – m          // size of second half of A
L ← copy of A[l..m], R ← copy of A[m..r]
i ← 0, j ← 0, k ← l
while i < n1 and i < n2 do   // merge into A
    if L[i] <= R[j] then
        A[k++] = L[i++]
    else
        A[k++] = R[j++]
while i < n1 do // copy rest of L into A
    A[k++] = L[i++]
while j < n2 do // copy rest of R into A
    A[k++] = R[j++]
```

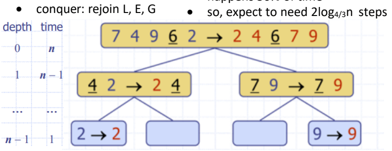
gorithm **inPlaceQuickSort(A, l, r)**
Input array A, indices l and r
Output array A with the elements of index between l and r rearranged in increasing order

```
if l ≥ r
    return
i ← random integer between l and r
x ← A[i]
(h, k) ← inPlacePartition(A, x)
inPlaceQuickSort(A, l, h – 1)
inPlaceQuickSort(A, k + 1, r)
```

in place quick sort: reorder sequence where it currently is

Quick sort

- O(n log(n)) expected, worst case O(n^2)
- divide – pick random pivot element x, partition S into:
  - L: elements less than x
  - E: elements equal to x
  - G: elements great than x
- recur: sort L and G
- conquer: rejoin L, E, G
- worst case: pivot is unique min or max element
- Good call: |L|, |G| < 0.75|s| - happens 50% of time
- so, expect to need $2\log_{4/3} n$ steps



Q1: Describe what would need to occur at each step of the algorithm to perform a **merge sort** of data in a file in **external memory**.
Sol: partitioning the array into 2 equal halves, we need extra memory for keeping a copy of every recursive call. i.e at every partition, the array is being copied into 2 parts, until the size of the array becomes 1. i.e we need to keep track of array elements log2N times i.e we need **N log2N extra memory** to completely store all recursive calls. After all, recursive calls are done until base case we perform merge routine i.e sorting 2 sorted lists and return to the called function. At each recursive call, we need to keep track of calls and memory allocated to them in the external memory to successfully complete the merge-Sort routine.
Q2: Exp time complex of **quick select** is O(n) vs.**quick sort** is O(nlogn)
Sol: Considering the position of this rand element is equally dist in N, there is 1/2 chance that it is between N/2 and 3N/4 th element. The recurrence relation as follows: T(N)=cN+T(3/4N), solving this recurrence relation we get O(N). as it is converging Taylor series. In this way, the theoretical time complexity of the quick-select algorithm is O(N) but in case of quick-sort, we select the pivot element in the order of O(N) every time in log N levels making the time complexity to a total of O(NlogN).

Comparison Sorts

- Sorted order determined by comparing keys - all keys are compared
- Examples - insertion, selection, merge and quick sorts

Decision Tree Height

- Height of decision tree is a lower bound on running time
- There are n!=1x2x…xn leaves – height is at least log (n!)

Lower Bound

- Comparison-based sorting algorithms takes at least log(n!) time
- Any comparison-based sorting algorithm must run in Ω(n log n) time

Bucket sort

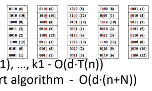
- Let S be a list of n (key, element) items with keys in the range [0, N − 1]
  - Bucket-sort uses the keys as indices into an auxiliary array B of lists (buckets)
  - Phase 1: Empty list S by moving each entry (k, o) into its bucket B[k] -- O(n)
  - Phase 2: For i = 0, …, N − 1, move the entries of bucket B[i] to the end of list S -- O(n + N)
  - move every element in S(k,o) into bucket B(k), then extract in-order to sort
- O(n+N) time (n = num elements, N = number of possible keys)
- stable sort – relative order of equal items is preserved
- keys can't be arbitrary objects – or need mapping from key to integer

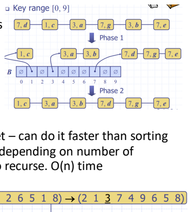
Pseudocode to determine whether or not a given Sequence S is full:

```
a = min(S)
b = max(S)
count = 0
initialise B with size (b - a + 1)
for k in S:
    if (B[k - a] is empty):
        B[k - a] = new Entry(k, new Object)
        count += 1
return count == B.size()
```

Lexicographic sort

- D-tuple – (k1, k2, …, kd)
- sorted in order of k1, then k2…
- to implement, stable sort by kd, k(d-1), …, k1 - O(d·T(n))
- **Radix-sort** – bucket sort as stable sort algorithm - O(d·(n+N))
- can sort integers in linear time
- e.g sequence S of n b-bit integers Runs in O(b·n) time
- Memory Usage
- Sort: 10, 999, 3, 100 000 000, 20
- Bucket Sort - O(5 + 100 000 000)
  Radix Sort - O(5 + 2)





Selection

- find kth smallest element in set – can do it faster than sorting
- Quick-select – split into L,E,G. depending on number of elements in each, may need to recurse. O(n) time
- recursion path representation:

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 (not really – call stack); randomized; fastest (good for large inputs)
merge-sort	$O(n \log n)$	sequential data access; fast (good for huge inputs)
radix-sort	$O(d \cdot (n + N))$	stable; fastest (with constraints on keys)
heap-sort	$O(n \log n)$	in-place; fast (good for large inputs)

k=5, S=(7 4 9 3 2 6 5 1 8) → (2 1 3 7 4 9 6 5 8)
k=2, S=(7 4 9 6 5 8) → (7 4 6 5 8 6)
k=2, S=(7 4 6 5) → (4 7 6 5)
k=1, S=(7 6 5) → (5 7 6)
5

- can do selection in O(n) expected time for random pivot selection
- O(n) worst time for deterministic pivot selection

Week 8 – Strings and pattern matching

- String – sequence of characters
- alphabet Σ – set of possible characters
- S[i, j] substring of S. S[0,i] prefix, S[i,m-1] suffix
- Pattern matching – find substring of text T equal to pattern P
- brute force – shift one character at a time, check whole string
- O(mn) time worst case

Boyer-Moore algorithm

- looking-glass heuristic – match from the end
- character-jump heuristic: mismatch at T[i]=c
  - if c in P, shift P to align last occurrence of c in P with T[i]
  - else, shift P to align P[0] with T[i + 1]
- preprocesses pattern and alphabet to build last occurrence function – L(c) = largest index i such that P[i] = c or -1
- O(nm+s) time – significantly faster than brute force on English text

Algorithm *BoyerMooreMatch(T, P, Σ)*

```
L ← lastOccurenceFunction(P, Σ)
i ← m – 1   { m is size of P }
j ← m – 1
repeat
    if T[i] = P[j] then
        if j = 0 then
            return i   { match at i }
        else
            i ← i – 1
            j ← j – 1
    else
        { character-jump }
        l ← L[T[i]]
        i ← i + m – min(j, 1 + l)
        j ← m – 1
until i > n – 1
return –1 { no match }
```

Algorithm *KMPMatch(T, P)*

```
F ← failureFunction(P)
i ← 0
j ← 0
while i < length(T)
    if T[i] = P[j] then
        if j = length(P) – 1 then
            return i – j { match }
        else
            i ← i + 1
            j ← j + 1
    else
        if j > 0 then
            j ← F[j – 1]
        else
            i ← i + 1
return –1 { no match }
```

Knuth-Morris-Pratt (KMP) -- runs in O(m+n)

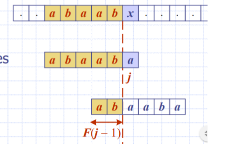
- compares text left-to-right
- what is most we can shift pattern so as to avoid redundant comparisons?
- Ans: largest prefix of P[0, j-1] that is a suffix of P[1, j-1]

KMP algorithm preprocesses the pattern

Failure function $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

KMP algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$



Computing the Failure Function

- Failure function can be represented by an array and can be computed in $O(m)$ time
- Construction is similar to the KMP algorithm itself
- Each iteration of the while-loop, either
  - i increases by one, or
  - shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

Algorithm *failureFunction(P)*

```
F[0] ← 0
i ← 1
j ← 0
while i < length(P)
    if P[i] = P[j] then
        {we have matched j + 1 chars}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        {use failure function to shift P}
        j ← F[j – 1]
    else
        F[i] ← 0 { no match }
        i ← i + 1
```

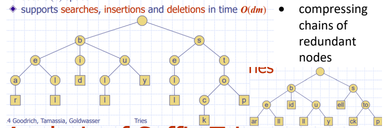
- if mismatch occurs at p[j] != T[i], set j = F(j-1)
- Failure function computed in O(m) time

TRIES -- Retrieval trees

- For pattern recognition, preprocess pattern. After preprocessing the pattern, KMP's algorithm performs pattern matching in time proportional to the text size
- for large immutable texts, may be better to preprocess text
- Standard Trie is an ordered tree – each node labelled with character -- children of node in alphabetically order
- Paths from root to external nodes yield strings

Analysis of Standard Tries

- n total size of the strings in S
- m size of the string parameter of the (e.g. search) operation
- d size of the alphabet
- uses $O(n)$ space
- supports searches, insertions and deletions in time $O(dm)$

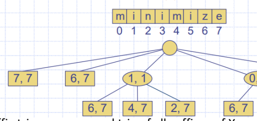


Compressed Trie:

- each internal node has degree of at least two
- compressing chains of redundant nodes

Analysis of Suffix Tries

- Compact representation of the suffix trie for a string X of size n from an alphabet of size d
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in X in $O(dm)$ time, where m is the size of the pattern
  - Can be constructed in $O(n)$ time



Suffix trie – compressed trie of all suffixes of X
Encoding Trie – mapping of each character of alphabet to binary code word, uses prefix codes
minimises size of encoding of X – greedy algorithm
Q: why is more efficient of using compact repre instead compressed repre for a suffix trie
Sol: compact repre don't need to store the character multiple times, just the start and end index
Huffman encoding: -- left:0 right:1

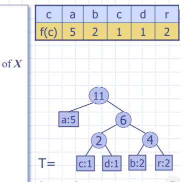
Huffman's Algorithm

X = abracadabra

Algorithm *Huffman(X)*:
Input: string X of length n
Output: optimal encoding tree for X

```
Compute frequency f(c) of each character c of X
PQ ← new empty Priority Queue
for each character c in alphabet of X do
    T ← single node binary tree storing c
    PQ.insert(f(c), T)
while PQ.size() > 1 do
    (f1, T1) = PQ.removeMin()
    (f2, T2) = PQ.removeMin()
    T ← a new binary tree T with
        left subtree T1 and right subtree T2
    PQ.insert(f1 + f2, T)
(f, T) = PQ.removeMin()
return T
```

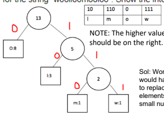
c	a	b	c	d	r
f(c)	5	2	1	1	2



- Prefixes of each code in the scheme should be unique.

Runs in O(n + d log d) – n size of X, PQ implemented using heap, d number of distinct characters

b. [6 marks] Using Huffman's algorithm, construct a Huffman tree that defines an optimal prefix code for the string "woolloomooloo". Show the intermediate states of the priority queue.



10	110	0	111
l	m	o	w

NOTE: The higher value should be on the right.

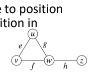
Q: A **Map** containing n elements may be **implemented using a Hash Table** where collisions are resolved by **separate chaining using unsorted linked lists**. What is the **worst-case** time complexity of the **put** method, which takes a key k and a value v and associates k with v, **updating** the value associated with the key if it was already mapped to another value? What is the **expected time complexity** of the **put** method? Are these the same or different? Explain why.

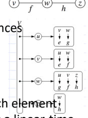
Sol: Worst-case complexity is O(n), when all elements are in a single bucket. We would have to iterate over all existing elements in that bucket to determine if we need to replace an element or not. Expected-case complexity is O(1) - we would expect all elements to be distributed somewhat evenly distributed across all buckets, so only a small number of elements (if any) would have to be iterated over to insert.

Week 9 & 10 – Graphs

Graph

- Pair (V,E) – V set of nodes, E collection of edges
- both vertices and edges can store elements
- edges directed or undirected – can have associated weight
- undirected graph – all edges undirected
- directed graph / digraph – all edges directed
- Terminology:
  - End vertices / endpoints of an edge
  - edges incident on a vertex
  - adjacent vertices – edge connects them
  - degree of vertex – number of edges connecting to it
  - parallel edges – two edges between same pair of vertices
  - self-loop – edge has same vertex on each end
  - path – sequences of alternating vertices and edges, starts and ends on a vertex
  - simple path – all vertices and edges are distinct
  - cycle – path that starts and ends on same vertex
  - simple cycle – simple path that's a cycle
  - o sum of deg(v) = 2* num edges
- Edge list – sequence of edges represented by vertex pairs
  - Store as sequences of edge and vertex objects
  - edges store element, vertices and reference to position
  - vertices store element and reference to position in sequence
- Adjacency list - Extends edge list
  - incidence sequence for each vertex
  - edges also store position in incidence sequences
- Adjacency Map - Adapts adjacency list
  Incidence list becomes a map
- Adjacency matrix - edge list structure
  - vertices have integer keys
  - 2d adjacency matrix, references to edges in each element
  - Finding all adjacent vertices to a single vertex is a linear-time
  - Determining of two vertices are adjacent is a constant-time





n vertices, m edges; no parallel edges; no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
outgoingEdges(v) incomingEdges(v)	m	deg(v)	n
getEdge(v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

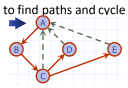
- Subgraph – vertices and edges are subset of graph
- spanning subgraph – contains all vertices of G
- connected – simple path between every pair of vertices
- connected component – maximal connected subgraph
- Tree T – T is connected, no simple cycles (unrooted tree)
- Forest – graph is composed solely of trees
- Spanning tree – spanning subgraph is a tree
- spanning forest – spanning subgraph is a forest

DFS

Algorithm *DFS(G, v)*:
Input Graph G and a vertex v of G
Output Collection of vertices reachable from v and their discovery edges and back edges

```
Mark vertex v as visited
for all e ∈ G.outgoingEdges(v) do
    if e is not explored then
        w ← G.opposite(v, e)
        if w has not been visited then
            Record edge e as a discovery edge for vertex w
            DFS(G, w)
        else
            Mark e as a back edge for vertex w
```

- O(n+m) time – n vertices, m edges
- Discovery edges form spanning tree
- DFS for entire graph – visit every vertex
- Can use DFS to find paths and cycle



Property 1 - DFS(G) visits all the vertices and edges in G
Property 2 - Discovery edges labelled by DFS(G) form a spanning tree T

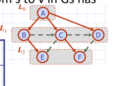
BFS

Algorithm *BFS(G, u)*
Input Graph G and a vertex u of G
Output Collection of vertices reachable from u and their discovery and cross edges

```
Q = new empty queue
Q.enqueue(u)
Mark vertex u as visited
while ¬Q.isEmpty() do
    v = Q.dequeue()
    for all e ∈ G.incidentEdges(v) do
        if e is not explored then
            w ← G.opposite(v, e)
            if w has not been visited then
                Record edge e as a discovery edge for vertex w
                Q.enqueue(w)
                Mark vertex w as visited
            else
                Mark e as a cross edge
```

- O(n+m) time – n vertices, m edges

Notation: Gs – connected component of s
Property 1: BFS(G, s) visits all the vertices and edges of Gs
Property 2: Discovery edges labeled by BFS(G, s) form a spanning tree Ts of Gs
Property 3: For each vertex v in Li
- path of Ts from s to v has i edges
- every path from s to v in Gs has at least i edges



Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	

DIGRAPHS

- each edge is directed
- Directed DFS – only go along edges in their allowed direction
- **transitive closure** – G' same as G but with edges added – edge added between u and v if there is a path between u and v
- Directed Acyclic Graph – no directed cycles
- topological ordering – order in which to do actions represented by nodes iff it is a DAG
- Strong Connectivity: Each vertex can reach all other vertices

Algorithm *FloydWarshall(G)*:
Input digraph G
Output transitive closure G^* of G

```
i ← 1
for all v ∈ G.vertices()
    denote v as v_i
    i ← i + 1
G_0 ← G
for k ← 1 to n do
    G_k ← G_{k−1}
    for i ← 1 to n (i ≠ k) do
        for j ← 1 to n (j ≠ i, k) do
            if G_{k−1}.areAdjacent(v_i, v_k) ∧
                    G_{k−1}.areAdjacent(v_k, v_j) then
                if ¬G_k.areAdjacent(v_i, v_j) then
                    G_k.insertDirectedEdge(v_i, v_j, k)
return G_n
```

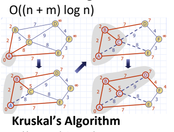
Algorithm *topologicalDFS(G, v)*
Input graph G and a start vertex v of G
Output labeling of the vertices of G in the connected component of v

```
Mark vertex v as visited
for all e ∈ G.outgoingEdges(v) do
    w ← G.opposite(v, e)
    if w has not been visited then
        { e is a discovery edge }
        topologicalDFS(G, w)
    else
        { e is a forward or cross edge }
Label v with topological number n
n ← n - 1
```

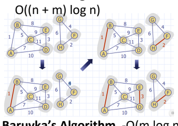
-- O(n+m) time

FIND MINIMIM SPINNING TREE

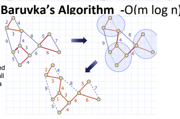
Prim-Jarnik Algorithm
O((n + m) log n)



Kruskal's Algorithm
O((n + m) log n)



Baruvka's Algorithm -O(m log n)



Dijkstra's algorithm – shortest path

- grow cloud of vertices starting with initial node
- store with each vertex distance from initial node
- O((n+m) log(n))

Algorithm *DijkstraDistances(G, s)*:

```
PQ ← new heap-based priority queue
for all v ∈ G.vertices()
    if v = s
        setDistance(v, 0)
    else
        setDistance(v, ∞)
    PQ.insert(getDistance(v), v)
while ¬PQ.isEmpty()
    u ← PQ.removeMin()
    for all e ∈ G.incidentEdges(u)
        { relax edge e }
        z ← G.opposite(u, e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(z)
            setDistance(z, r)
            PQ.replaceKey(getLocator(z), r)
```