Design Of Information Structures

07 January 2020 15:25

50% coursework
- 10% lab sessions
- 40% programming exercise (due week 10)

50% examination (1.5 hours in May)
- Past papers are online

An Introduction to Abstract Data Types, Data Structures and Algorithms – W. Watt and D.F Brown
Data Structures and Algorithms in Java – M.T. Goodrich and R. Tamassia

Maps

01 February 2020 18:27

A searchable collection of key value entries
Allows for searching, inserting and deleting
Multiple entries with the same key are not allowed

Get(k) - get the value with key k or return null if no value exists with that key
Put(k, v) - add a new pair into the map with key k and value v
EntrySet() - return an iterable collection with the entries in the map
KeySet() - return an interable collection of the keys in m
Values() - return an iterator of the values in the map

Performance

Maps can be implemented using linked lists

Get, put and remove operations have a linear time complexity as they require searching through the map to verify the existense of keys

The unsorted map is only useful when the map is small

Hash Tables

Hash Functions – composition of two functions; hash code to convert the key to an integer and a compression function to restrict the key value in the range of 0 to n –1 where n is the number of elements in an array

The goal of the hash function is to disperse the keys randomly

Hash Code

Polynomial Accumulation
Partition the of the key into a sequence of components of fixed length
Turn the bits into a polynomial with the bit values as the coefficients
Evaluate the polynomial in O(n) time using Horners Rule

Compression Functions

mod N - where N is the size of the hash table. Hash tables are usually have a prime size.
MAD – multiply, add and divide – ay + b mod N where a,b >= 0 and a mod b != 0

Handling Collisions

Separate chaining – create a link to a list which stores all of the entries that collided at that key. Requires additional memory outside of the table
Linear probing – keep moving to the right and try to enter the value there, if that is also full then keep moving until an empty slot is found. Causes long blocks (lumping)
Double Hashing – another hash function is used if the first hash function causes a collision

Performance of Hashing

Worst case, insertion, searches and removals take O(n)
Worst case occurs when all of the keys inserted into the map collide
Collisions occur most when load factor (elements stored divided by max size of hash table) approaches 1 p

Keeping the hash table size large uses more memory but increases speed of operations on the table

Sets and Multimaps

01 February 2020 19:27

Sets are unordered collections of elements without duplicates – typically supports efficient membership tests

Elements of a set are like keys in a map but without the auxiliary values

Multiset (bag) is a set-like container that allows duplicates

Multimap is like a traditional map where keys can be associated to multiple values

- Contains(e) - check S for e
- Add(e)
- AddAll(T) - S union T
- RetainAll(T) - delete anything not in T - S and T
- RemoveAll(T) - delete all elements in T from S – S / T

Generic Merging

Takes two lists that represent lists of objects that are sorted – walk along the two lists and observe the elements in both lists and consider the smallest element in the lists

Any of the set operations can be implemented using a generic merge

E.g. intersection – only copy elements that are duplicated in both lists

All methods run in linear time

Multimaps

For every key in M we have a list of entries that can be accessed with E(k)
So the entries of m are (k, E(k))

Trees

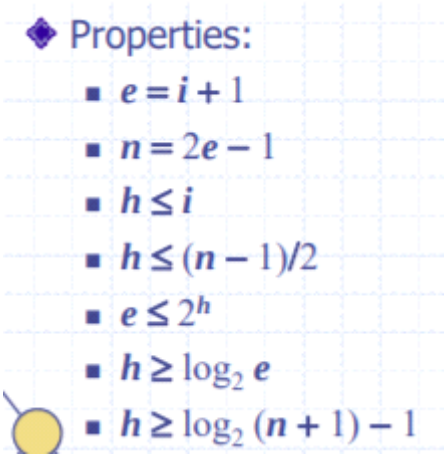
01 February 2020 19:59

Consist of vertices and edges (nodes and links)

Abstract model of a hierarchical structure
Nodes have a parent-child structure

Root – node without a parent
Leaf – node with no children
Depth – number of ancestors
Ancestors – parent, grandparent
Descendent – child / grandchild
Sub Tree – tree consisting of a node and its descendents

Root()
Parent(p)
Children(p) - iterable
NumChildren(p)
IsInternal(p) - leaf or not



In Order Traversal of Binary Trees
Recursively explore the tree. Nodes visited after its left subtree and before its right subtree

```
If left(v) != null
        InOrder(left(v))
Visit(v)
If right(v) != null
        InOrder(right(v))
```

Printing Arithmetic Expressions

- Specialisation of an inorder traversal
- Print operand or operator when visitng node
- Print (before traversing left subtree
- Print) after traversing right subtree

Linked Structure for Trees
A node is represented by an object string an element, a parent node and a sequence of children nodes

Node objects implement the position ADT

Analysis of Algorithms

16 January 2020 22:17

Average time is very hard to compute so we measure worst case time complexity

Experimental Evidence

1. Algorithm must be implemented
2. Impossible to try all inputs
3. Same hardware and software must be used in comparisons

Theoretical Analysis

1. High level description instead of implementation
2. Characterises running time as a factor of input size n
3. Takes into account all n
4. Allows evaluation independent of the hard / software environment

Pseudo Code

- High level description
- More structured than prose
- Less detailed than a program
- Preferred notation for algorithms
- Hides design issues

7 Time Complexities

1. Constant time – 1
2. Logarithmic – Logn
3. Linear – n
4. N Log N – n Logn
5. Quadratic – n^2
6. Exponential – 2^n
7. Cubic time – n^3

Primitive Operations

Basic computations that are languages independent and occur in constant time
Examples include:

- Assigning variables
- Evaluating expressions
- Calling methods
- Returning from a method

By inspecting pseudo code we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

If t(n) is the worst case time complexity of a function;

- A = time take by the fastest primitive operation
- B = time taken by the slowest primitive operation

If we determine that an algorithm performs 4n + 5 operations in the best case scenario
And the same algorithm performs 5n + 5 operations in the worst case scenario

Then we can see that T(n) is bounded by two linear functions

$A(4n+5) < T(n) < B(5n+5)$

Big O Notation

16 January 2020 22:32

Insertion sort is n^2 / 4
Merge sort is 2 n log n

Sorting a million items takes 70 hours for an insertion sort and 40 seconds for a merge sort

Growth rates are not affected by:

- Lower order terms
- Constant factors

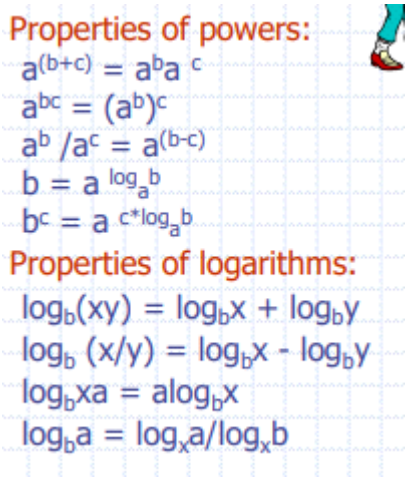
Growth Rate

Big O notation gives an upper bound on the growth rate fo a function
We can use big O to rank functions based on their growth rate

- Always use the smallest class of functions – e.g. 2n is O(n) not O(n^2)
- Use the simplest expression of the class - 3n + 5 is O(n) instead of 3n + 5 is O(3n)

Asymptotic Algorithm Analysis
The asymptotic analysis of an algorithm determines the running time in big O notation
To perform the analysis
Find worst case number of primitive operations executed as function of the input size
Express this function with big O notation

Useful Maths



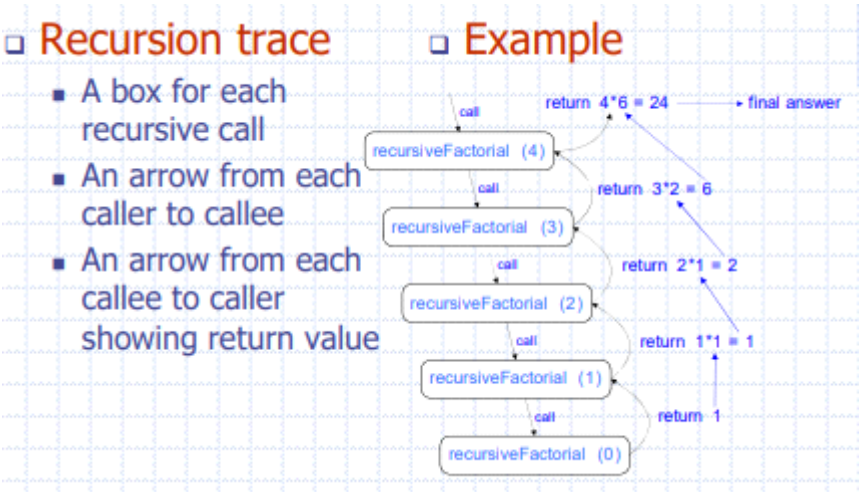
Relatives of Big O
f(n) is Big Omega if time complexity is greater than or equal to g(n)
f(n) is Big Theta if time complexity is equal to g(n)

Recursion

16 January 2020 22:43

A method calling itself
Computation representation of mathematical induction

Base case – all calls need >= 1 base case
Recursive Calls – call current method – defined so that it makes progress towards the base case



Linear Recursion

Test for Base Case

- begin by testing for a set of base cases
- Every possible chain of recursive calls must eventually reach a base case and handling of base case should not use recursion

Recur Once

- Perform a single recursive call
- This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
- Define each possible recursive call so that it makes progress towards a base case

Example: Reversing an Array

```
if i < j then
    Swap A[i] and A[ j]
    reverseArray(A, i + 1, j – 1)
return
```

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes it recursive call as its last step
- These methods can be easily converted to non-recursive methods to save on some resources
- Tail recursive functions can be optimised by the compiler – since the call is the last thing in the function and there is nothing left to do inside said function, there is no need to save the current functions stack frame

Binary Recursion

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

Singly Linked Lists

21 January 2020 15:08

The size can vary as we are using them, unlike arrays, which for most programming languages have a fixed size – all space allocated by the compiler when the array variable is declared. This memory is unavailable to other parts of the program that may need it.

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

First node is head
Last node points to null so we know it is last

Inserting at the Head

- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node

Inserting at the Tail

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

Removing at the Head

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

If no other variables point to the first object, then nothing is pointing to it and the compiler is not aware of it – nothing points to it. Java VM's will identify these values with no variables pointing to them. There is no pointer and so this variable cannot be used again. This is detected and the garbage collector removes the value (allows for the location to be overwritten)

Removing at the Tail

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

Removing at the tail is therefore inefficient. This is why we use doubly linked lists

Doubly Linked Lists

21 January 2020 15:46

A doubly linked list can be traversed forward and backwards – solving the problem of removing from the tail – if we remember the tail, then we can traverse backwards – we can find the second last element in the list.

The problem of this system lies in the 2n number of pointers rather than having n. This can make coding these systems confusing

Nodes store:

- Element
- Link to the previous node
- Link to the next node

Allowing us to insert into the list in constant time anywhere in the list

Stacks

23 January 2020 17:07

A stack is an abstract data type and its implementation using arrays or linked lists are quite straight forward.

What is an Abstract Data Type?

An abstraction of a data structure
ADT's specify:

- Data stored
- Operations on the data
- Error conditions associated with operations

Stack – Abstract Data Type

- Stacks store arbitrary objects
- Insertions and deletions are LIFO
- Operations: push, pop, top, size, isEmpty

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Assumes null is returned from top and pop when stack is empty
- Different from the built-in java class java.util.Stack

Method Stack in the JVM

- The JVM keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushed on the stack a stack frame containing local variables and return value, program counter, keeping track of the statement being executed

- When a method ends, its frame is popped from the stack and control is passed to the method on the top of the stack
- This allows for recursion

Space complexity is O(n)
Time complexity of operations is O(1)

Stacks can be used to evaluate postfix notation

Spans

Stacks can be used to compute the spans of a graph in linear time, as opposed to quadratic time that is required when computing normally (with a for loop nested in a while)
The reasoning behind this is that when using a stack, the element is only pushed to the stack once and popped once

Linear Time Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

```
Algorithm spans2(X, n)                                  #
    S ← new array of n integers                         n
    A ← new empty stack                                 1
    for i ← 0 to n − 1 do                               n
        while (¬A.isEmpty() ∧
                X[A.top()] ≤ X[i] ) do                  n
            A.pop()                                     n
        if A.isEmpty() then                             n
            S[i] ← i + 1                                n
        else
            S[i] ← i − A.top()                          n
        A.push(i)                                       n
    return S                                            1
```

Queues

25 January 2020 12:15

FIFO ADT

Enqueue(obj) appends obj to the queue
Dequeue(obj) removes the first item from the queue

First() - peeks the first object in the queue
Size()- peeks the last object in the queue

Boundaries – dequeuing on an empty queue results in a null return

Array-Based Queue

An array with two auxiliary pointers can create a queue. Pointer 1 holds the index of the first element in the queue and pointer 2 holds the number of elements

When the queue has fewer than N elements, the location of the first free slot = (f + size) mod N

Lists

28 January 2020 15:13

- Get() returns the element at index I
- Set(I, e) - Replaces the element at index I and returns the old element that was replaced
- Add(I, e) inserts a new element at index I, moving all elements index later in the list
- Remove(I) removes the elements at index I and shifts all the subsequent elements one index earlier in the list

Array Lists

Using an array is an obvious choice to store

Performance

In an array base implementation of a dynamic list, the spacr used is O(n)
Indexing the element at I takes O(I) time
Add and remove run in O(n) time
In an add operation, when the array is full, instead of throwing an exception we can simply replace the array with a larger one

Incremental Increase vs Double Array Size

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

Incremental

Over n push operations we replace the array k = n/c times where c is a constant
The total time of a series of n push operations is proportional to n + ck(k+1)2 = O(n^2)
Amortized time = O(n)

Doubling

We replace the array k = logn times
The total time t(n) of a series of n push operations is proportional to 3n – 1
The amortized time of a push is therefore O(n)

Positional Lists

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

Iterators

An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements
.hasNext() returns true if there is at least one additional element in the sequence

Next() returns the next element in the sequence

The Iterable Interface

```
for (ElementType variable : collection) {
  loopBody                                    // may refer to "variable"
}
```

is equivalent to:

```
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
  ElementType variable = iter.next();
  loopBody                                    // may refer to "variable"
}
```

Binary Search Trees

06 February 2020 17:26

Ordered Maps

Keys are assumed to come from a total order.
Items are stored in order by their keys

This allows us to support nearest neighbour queries:
Item with largest key less than or equal to k
Item with smallest key greater than or equal to k

Binary Search

Binary search can perform nearest neighbour queries on an ordered map that is implemented with an array, sorted by key.
At each step the number of candidate items is halved and as such has a time complexity of O(log n)
External nodes do not store items (design decision)

Search Tables

A search table is an ordered map implemented by means of a sorted sequence
We store the items in an ordered array, ordered by key

Searches take O(log n) time, using binary search

- Inserting a new item takes O(n) time, since in the worst case we have to shift n/2 items to make room for the new item
- Removing an item takes O(n) time, since in the worst case we have to shift n/2 items to compact the items after the removal
- The lookup table is effective only for ordered maps of small size or for maps on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search Trees

Let u, v and w be three nodes such that u is in the left subtree aof v and w is in the right subtree of v, we have key(u) leq key(v) leq key(w)
External nodes do not store items

Searching

To search for a key k we trace a downward path starting at the root
The next node visited depends on the comparison of k with the key of the current node
If we reach a leaf, the key was not found

Insertion

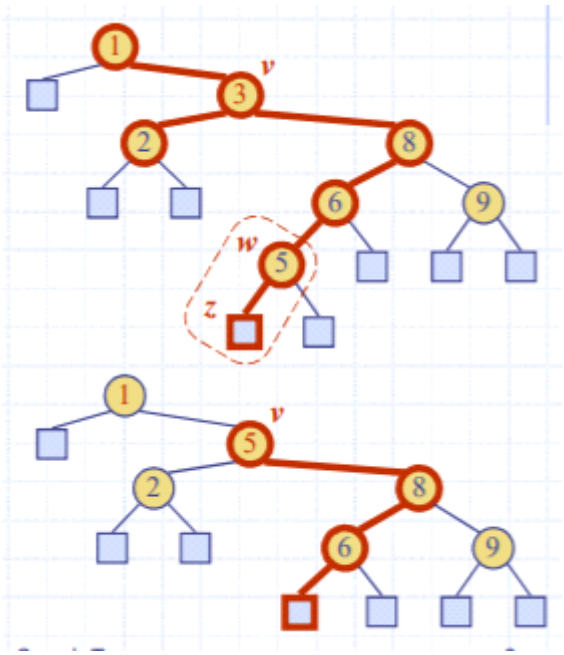
To perform opration put(k,o) we search for key k using searchTree
Assume k is not already in the tree and let w be the leaf reached by the search
We insert k at the node

Deletion

Search for key k
Assum k is in the tree, and let v be the node storing k
If node v has a leaf child w, we remove v and w from the tree with removeExternal() and plug the

remaining child into the void

FInd the node you want to delete
Traverse left until you find a node with only a single child
Swap that node with the one you want to delete



Performance
Space is O(n)
Methods get, put and remove take O(h) time
The height h is O(n) in the worst case and O(log n) in the best case

Priority Queues

07 February 2020 10:07

Stores a collection of entries
Each entry is a pair – key, value

- Insert(k, v)
- RemoveMin() - removes and returns the entry with the smallest key, or null if the queue is empty
- Min() - returns but does not remove the entry with the smallest key
- Size()
- Empty()

Total Order Relations

Keys in a priority queue can be arbitrary objects on which an order is well defined and linear
Two distinct entries in a priority queue can have the same key

We can use an Entry<K,V> and Comparator ADT to construct the priority queue.

Sequence-Based Priority Queue

One implementation of the PQ data type (collection of objects where one can insert, remove min by comparing keys in the kv pair) is the sorted and unsorted list.

Unsorted List

Performance – O(1) time since we can insert at the beginning or end of the sequence
RemoveMin and Min take O(n) time since we have to traverse the entire sequence to find the smallest key, as no order is maintained

Sorted List

Performance – reversed compared to unsorted list – insertion is slow O(n) but removeMin and min are fast at O(1), since smallest key will always be at the beginning.

Priority List Sorting

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

Selection Sort

- Variation of the PQ-sort where the PQ is implemented with an unsorted sequence
- Inserting the elements into the priority queue with n insert operations takes O(n) time
- Remove the elements in sorted order from the priority queue with n removeMin operations takes time proportional to n

Time complexity is O(n^2)

Insertion Sort
Insertion-Sort is the variation of PQ-sort where the queue is implemented with a sorted sequence

Inserting the elements into the priority queue with n insert operations takes time proportional to 1 + 2 + 3 … + n
Removing the elements takes time proportional to n, making the total complexity O(n^2)

Heaps

07 February 2020 10:44

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

Heap-Order – for every internal node v other than the root, key(v) geq key(parent(v))
Complete Binary Tree – let h be the height of the heap. For x = 0 to h-1, there are 2^x nodes of depth x
At depth h – 1 the internal nodes are to the left of the external nodes

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

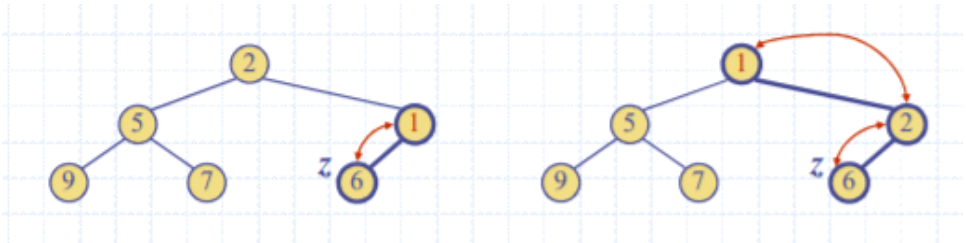
Theorem:
A heap storing n keys has height O(log n)
Proof: (we apply the complete binary tree property)
Let h be the height of a heap storing n keys
Sicne there are 2^I keys at depth i = 0, … h –1, and at least one key at depth h, we have n >= 1 + 2 + 4 + 2^(h-1) + 1
THus n >= 2^h, I.e. h <= log n

We can use a heap to implement a priority queue
We store a key element item at each internal node
We keep track of the position of the last node

Insertion into a heap
1) find the insertion node z (new last node)
2) store k at z
3) Restore heap order property

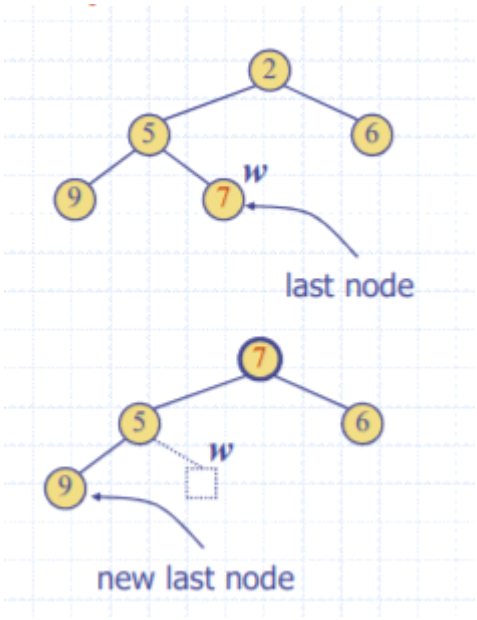
Upheap

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



Removals from a heap

Method removMin() removes the root key from the heap
Replace the root key with the key of the last node
Remove w
Restore heap order property



Updating the Last Node

The insertion node ca be found by traversing a path of O(logn) nodes

- Go up until a left child or the root is reached
- IF a left child is reached, go to the right child
- Go down left until a leaf is reached

Similar algorithm for updating the last node after a removal

Heap Sort

Consider a priority queue with n items implemented by means of a heap
The space used is O(n)
Methods insert and removeMin take O(logn) time
Methods size, isEmpty, and min take time O(1)

Array-Based Heap Implementation

We can represent a heap with n keys by means of a n array of length n
For the node at rank I

- The left child is at rank 2i + 1
- The right child is at rank 2i + 2

Links between nodes are not explicitly stored
Operation add corresponds to inserting at rank n + 1
Operation remove_min corresponds to removing at rank n
Yields in place heap-sort

Merging Two Heaps

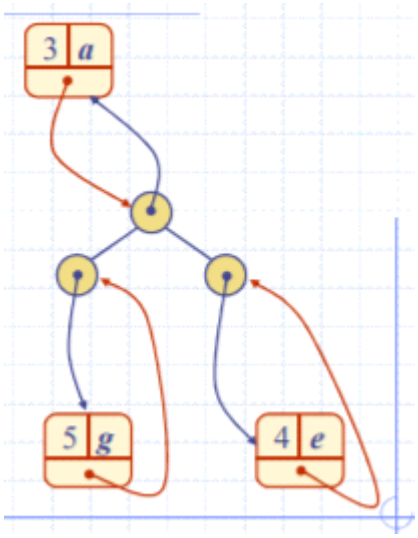
We are given two heaps and a key k
We create a new heap with the root node storing k and with the two heaps as subtrees
We perform down heap to restore the heap-order property

Adaptable Priority Queues

13 February 2020 17:11

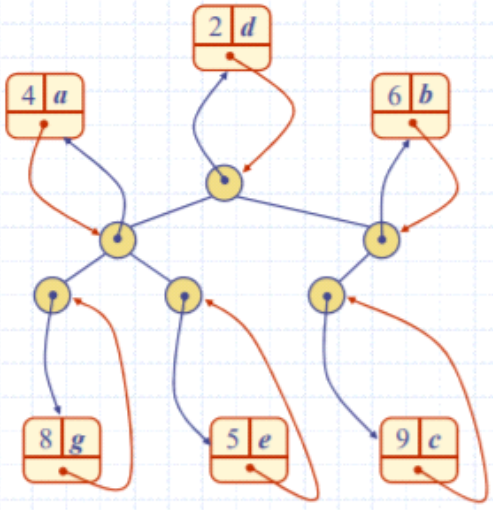
Location aware entries track the location of a key value object within a structure

Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier



Heap Implementation

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



Skip Lists

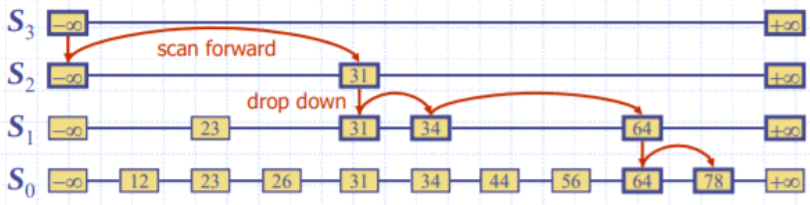
13 February 2020 17:34

- A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in nondecreasing order
  - Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$
  - List S_h contains only the two special keys
- We show how to use a skip list to implement the map ADT

In short, skip lists are a linked-list-like structure which allows for fast search. It consists of a base list holding the elements, together with a tower of lists maintaining a linked hierarchy of subsequences, each skipping over fewer elements.

Search

- We search for a key x in a a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare x with $y \leftarrow key(next(p))$
    - $x = y$: we return $element(next(p))$
    - $x > y$: we "scan forward"
    - $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return *null*
- Example: search for 78



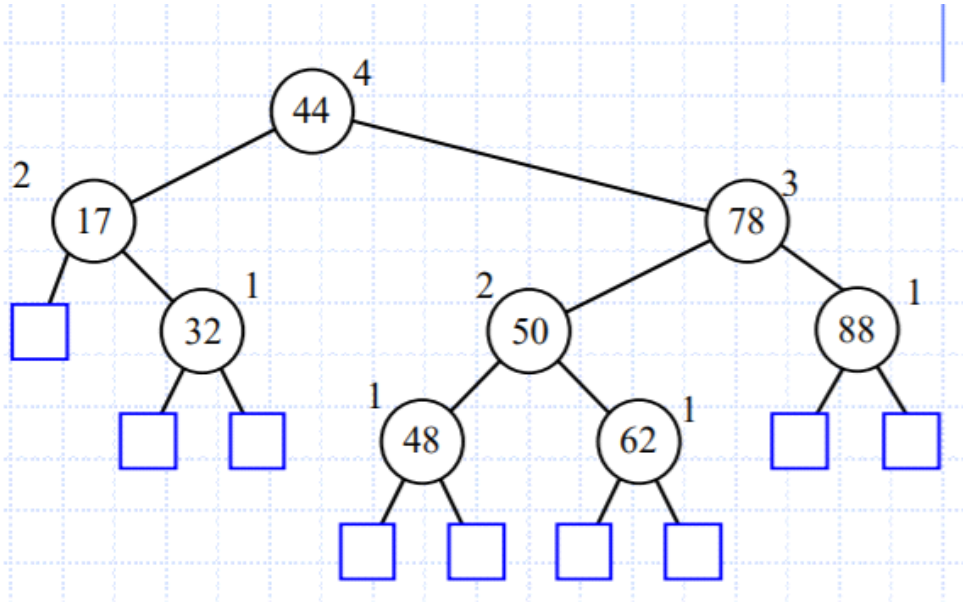
 Skip Lists 3

AVL Trees

18 February 2020 15:08

AVL trees are balanced
An AVL tree is a binary search tree such that for every internal node v or T the heights of the children of V can differ by at most 1.

The number of children that each node has (depth) can differ by at most 1



Height of an AVL Tree

The height of an AVL tree storing n keys is O(log n)

When n = 1, the number of internal nodes is 1
When n = 2, the number of internal nodes is 2
When n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1, and another of height h-2
That is, n(h) = 1 + n(h-1) + n(h-2)

Read-only operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications have to observe and restore the height balance of the sub-trees.

Searching

Searching is the same as in an unbalanced binary tree and as such the limit of the number of comparisons is h for a successful search, and near h for an unsuccessful search, thus giving the searching time a complexity of O(log n)

Traversal

Once a node has been found in a tree, the next or previous node can be found in amortized constant time
Visiting each node in the tree would cause each node to be visited twice – once in the downward traversal and once on the way back up. Since there are n – 1 links in a tree, the amortized cost of traversal is 2x(n-1)/n which is roughly 2. Thus the complexity of traversing an AVL tree is O(1)

Insertion

Insertion is as in a binary search tree – more explicitly if the tree is searched and found to be empty, the node is inserted as root. If the tree is not empty, the search returns a node and a direction left or right indicating the direction in which the returned node doesn't have a child. Then the node to be inserted is made a child of the returned node in the returned direction.

Insertion is always done by expanding an external node.

When we insert a node we have to check each of the nodes ancestors for consistency with the invariants of AVL trees. This process is called retracing.

This is achieved by considering the balance factor of each node. If the balance factor of a tree is outside of (-1, 0, 1) then the subtree rooted at this node is AVL unbalanced and rotation is needed.

The correct single rotation will always balance the tree.

Delete

The preliminary steps for deletion are the same as a binary tree.
We need to check the AVL invariant has not been broken. This is again called retracing.

Unlike insertion where a single rotation will balance the tree, after the appropriate single or double rotation the height of the rebalanced subtree decreases by one meaning that the tree has to be rebalanced again on the next higher level.

Trinode Restructuring

If after a modifying operation, e.g. an insertion or a deletion, the invariant of the AVL is broken (a node has a balance factor != (1, 0, -1) then balance must be restored.

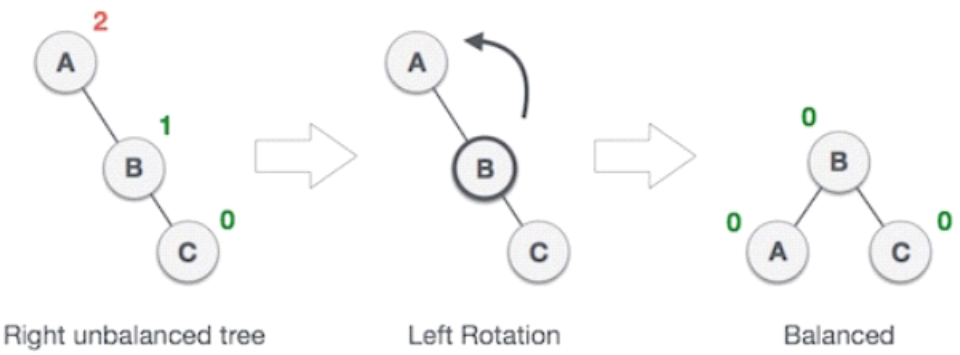
The tools for doing so are called 'rotations', because they move keys vertically, ensuring that the horizontal ordering is preserved (vital for a binary tree)

Let X be a node whose AVL invariant is broken – a key has been added to its left or right subtree.
Let Z be the higher child. Note that (by induction) Z is in AVL form.

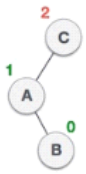
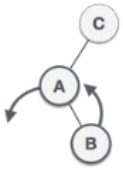
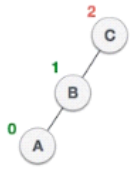
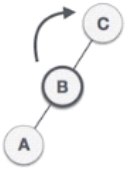
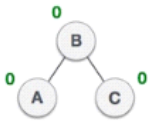
There are four types of rotation

- Left rotation – simple
- Right rotation – simple
- Left-Right rotation – double
- Right-left rotation – double

Example of a left rotation



Example of a left-right rotation

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Graphs

21 February 2020 22:46

A graph is a pair (V, E) where:

- V is a set of nodes called vertices
- E is a collection of pairs of vertices called edges
- Vertices and edges are positions and store elements

Edge Types

Directed Edges

- ordered pair of vertices (u, v)
- First vertex u is the origin
- Second vertex v is the destination
- e.g. a flight from A to B

Undirected edge

- Unordered pair of vertices (u,v)
- e.g. a flight route

Applications

- Electronic circuits – printed circuit board, integrated circuit
- Transportation networks – highway network, flight network
- Computer networks – LAN, internet
- Databases – entity relationship diagrams

Terminology

- End vertices (aka endpoints) of an edge are the vertices at the end of an edge
- Edges incident on a vertex are edges connected to a vertex
- Vertices are adjacent if they are connected
- Degree of a vertex is the number of relationships that a vertex is in.
- Parallel edges both connect from A to B
- Path – a sequence of alternating vertices and edges, beginning and ending with a vertex
- Simple path – a path such that all its vertices and edges are distinct
- Cycle – a circular path that begins and ends at the same vertex.
- Simple Cycle – a cycle such that all its vertices and edges are distinct

Properties of Graphs

N – number of vertices
M – number of edges
Deg(v) - degree of vertex v

1. Sum of deg(v) = 2m
2. In an undirected graph with no self-loops and no multiple edges - M leq n(n – 1)/2

Episode 9: Tag der Arbeit


Auf
Deutsch
gesagt!
Der Deutschpodcast
für Fortgeschrittene
B2+

Depth First Search

25 February 2020 15:38

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G

Trees and Forests

A tree is an undirected graph T such that T is connected and T has no cycles

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest

Depth First Search

DFS is a general technique for traversing a graph
A DFS traversal of G

- Visits all the vertices and edges of G
- Determines whether G is connected
- Computers the connected components of G
- Computes a spanning forest of G

DFS on a graph with n vertices and m edges takes O(n + m) times
DFS can be further extended to solve other graph problems

- Find and reported a path between two given vertices
- Find a cycle in the graph

Depth first search is to graph what Euler tour is to binary trees

Algorithm DFS(Graph G, vertex u)

Mark vertex u as visited
For each of u's outgoing edges do
    If vertex has not been visited then
        Record edge e as the discovery edge for vertex v
        Recursively call DFS(G, v)

Breadth First Search

05 March 2020 17:10

Thea algorithm searches the tree by width first, traversing a node and its children, then moving horizontally to the next node and searching those children, until all of the nodes on the same row have been traversed.

The algorithm uses a mechanism for setting and getting labels of vertices and edges

DFS vs BFS

Spanning forest, connected components, paths, cycles – DFS and BFS
Shortest Paths – BFS
Biconnected components – DFS

Back Edge (v, w)
W is an ancestor of v in the tree of discovery edges

Cross Edges (v, w)
w is in the same level as v or in the next level

Directed Graphs

05 March 2020 17:23

A digraph is a graph whose edges are all directed

Applications

One way streets
Flights
Task scheduling

Digraph Properties

A graph such that each edge goes in one direction
Edge goes from a to b but not from b to a
If we keep in edges and out edges in sperate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

Directed DFS

WE can specialise the traversal algorithms DFS and BGS to digraphs by traversing edges only along their direction

In the directed DFS algorithm we have four types of edges
Discovery edges
Back edges
Forward edges
Cross edges
A directed DFS starting at a vertex determines the vertices reachable from s

Strong Connectivity

Each vertex can reach all other vertices

The algorithm to check for strong connectivity
Pick a vertex and perform a DFS
Reverse the graph and perform a DFS again

Transitive Closure

If a graph G has a directed path from u to v, then G* has a directed edge from u to v.

Computing the transitive closure
We can perform DFS starting at each vertex in O(n(n+m)) or alternatively use the dynamic programming: The Floyd-Warshall Algorithm

Vertices have labels 1 to n

DAGs and Topological Ordering

A directed acyclic graph is a digraph that has no directed cycles
A topological ordering of a digraph is a numbering v1 – vn such that for every edge vi, vj we have I < j
Example
In a task scheduling digraph, a topological ordering is a sequence of tasks that satisfies the precedence constrains
A digraph admits a topological ordering if and only if it is a digraph

Algorithm for topological sorting

If one has a acyclic graph, there is at least one vertex which does not have an outgoing edge
You can prove this by contradiction

H = temp copy of graph
N = num vertices
While H is not empty
    Let v be a vertex with no outgoing edges
    Label v <- n
    N <- n – 1
    Remove v from H