C=is

Hash tables

• A key k hashes to a table location ℓ if h(k) = ℓ.

• A collision occurs if two keys k1 and k2 (used in the set or map) hash to the same location, that is, h(k1) = h(k2). Note: Collisions are unavoidable if the size of the set or map is greater than the table size. Different kinds of hash tables deal with collisions in different ways.

related to its **size**.

= the logarithm of its ***Size***

(c) You were asked to consider the **deletion** of a value from a red-black tree T when this value

is in the tree, recalling that some node y (with at most one non-NIL child) is removed and

that a child x of y gets promoted to replace it. You were asked to describe

• the **colour** given to x when it is promoted (and how this depends on the colour of y)

• the problem that this might cause, and

• (somewhat briefly and informally) how this problem is corrected.

Colour Given to x if y was red: In this case, the colour of x is not changed.

Colour Given to x if y was black: If x was originally red then its colour is changed to

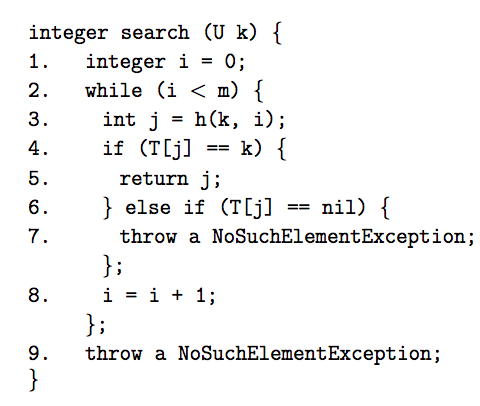
red-black. Otherwise the colour of x was originally black and its colour is changed to

double-black.

**Problems This Might Cause:** This (temporarily) introduces two new “colours” for nodes in

the tree. All red-black and double-black nodes must be eliminated before the resulting

tree is, once again, a red-lack tree.



How This is Fixed: A red-black node, or a double-black node that is at the root, can

simply be replaced by a black node in order to restore the red-black properties.

A series of **recolourings** and **rotations** can be used to move double-black nodes

**closer to the root** until either a red-black vertex results from the operation, or one of the

simpler cases mentioned above results.

**3 (a)** You were first asked to consider **hashing with chaining**, using table size m = 10, and

using the following hash function to store integers:

h(x) = x (mod 10).

You were asked to draw the hash table that would be produced by inserting the values

**Insertion in Binary Search Trees:**

**-** If the tree is empty then create a new node that stores the input value, and make this the root of the tree. Otherwise compare the input value to the value stored at the root of this tree.

• If the input value is less than the value at the root, recursively insert the input value into the left subtree of the root.

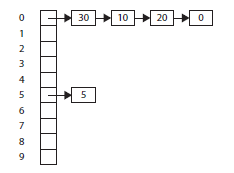
• If the input value is greater than the value at the root, recursively insert the input value into the right subtree of the root.

• Finally, if the input value is equal to the value at the root, throw an exception and do not change the tree at all.

0, 20, 10, 30, 5

(in this order) into an initially empty table.

**Solution:** Using the algorithm for **insertion** described during the first lecture on hash

tables, the following hash table is obtained:

**(b)** You were then asked to repeat the above question, using **hashing with open addressing**.

In this case you were asked to use a hash function such that

h(x, 0) = x (mod 10)

and such that

h(x, i) = x + i2 mod 10

for every integer i such that 1 < i < 9 — so that **quadratic probing** is being used.

**Solution:** Using the algorithm for **insertion** described during the second lecture on hash

tables, the following hash table is obtained:

(c) You were next asked to draw the hash table that you would get by starting with the hash

table you produced when answering part (b) and **deleting** 20.

**Solution:** Using the algorithm for **deletion** from the second lecture on hash tables, the

following hash table would be obtained. The special value DEL is being used here, instead

of DELETED, just to make the table easier to draw.

d) Finally, you were asked to explain why it is generally not a good idea to use hash tables

**Deletion in Red-Black Trees:**

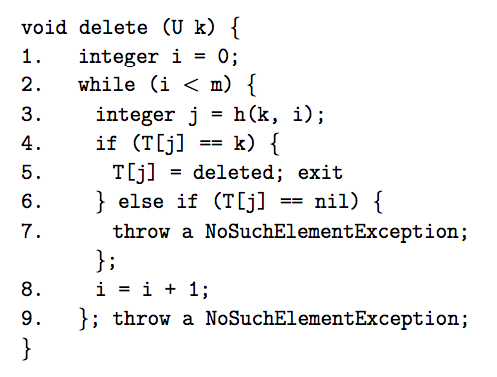
**Colour Given to x if y was red**: In this case, the colour of x is not changed.

**Colour: Given to x if y was black**: If x was originally red then its colour is changed to red-black. Otherwise the colour of x was originally black and its colour is changed to double-black.

**Problems This Might Cause**: This (temporarily) introduces two new “colours” for nodes in the tree. All red-black and double-black nodes must be eliminated before the resulting tree is, once again, a red-lack tree.

**How This is Fixed**: A red-black node, or a double-black node that is at the root, can simply be replaced by a black node in order to restore the red-black properties.

A series of recolourings and rotations can be used to move double-black nodes closer to the root until either a red-black vertex results from the operation, or one of the simpler cases mentioned above results



with open addressing if **deletions** are common.

**Solution:** As the above solution indicated, locations in the has table can be filled up using

the special value DELETED (or “DEL”) — and this can extend the cost of all operations,

because these locations must be examined along the locations storing elements of the set

being represented during searches.

**4** (a) You were first asked to describe, IN YOUR OWN WORDS, the ***binary search*** algorithm.

Recall that this algorithm

• accesses as global data, but does not change, an array A that has positive length and

stores elements of an ordered type T in nondecreasing order, as well as a key k of

the type of values stored in the array;

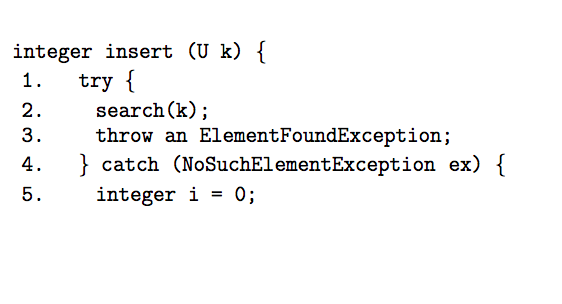
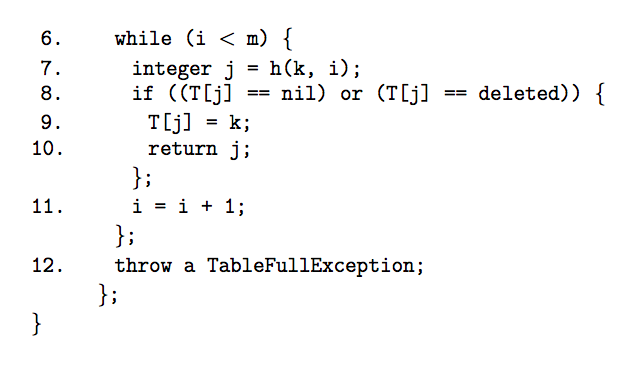
• receives integers low and high such that

0 ≤ low ≤ A.length and − 1 ≤ high ≤ A.length − 1

as input.

If low ≤ high and there exists an integer i such that low ≤ i ≤ high and A[i] = k, then

an integer i with these properties is returned as output. A NoSuchElementException is

thrown, otherwise.

***Solution:*** If high < low then a NoSuchElementException is thrown. Otherwise the

value

mid = ⌊(low + high)/2⌋

is computed. This is always an integer such that low ≤ mid ≤ high. k is then compared

to A[mid].

• If k < A[mid] then the algorithm is called recursively; low is not changed, but high

is replaced by mid − 1. The output generated (or exception thrown) by this recursive

call is returned.

• If k = A[mid] then mid is returned as output.

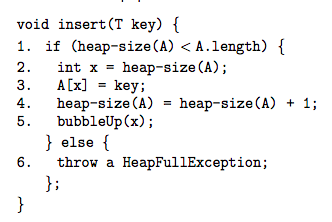
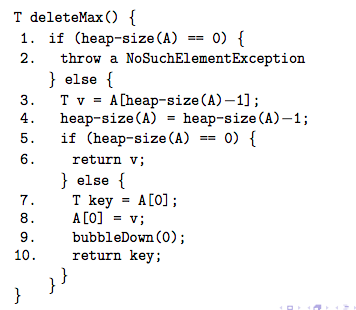
• If k > A[mid] then the algorithm is called recursively: low is replaced by mid+1 but

high is not changed. The output generated or exception thrown is returned by this recursive call.

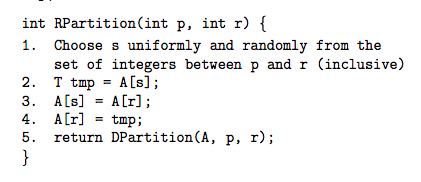
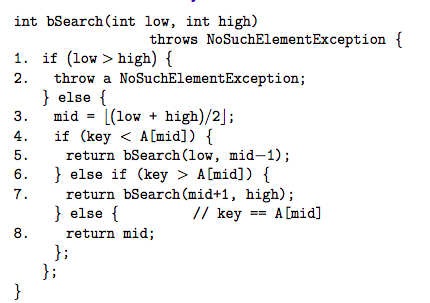
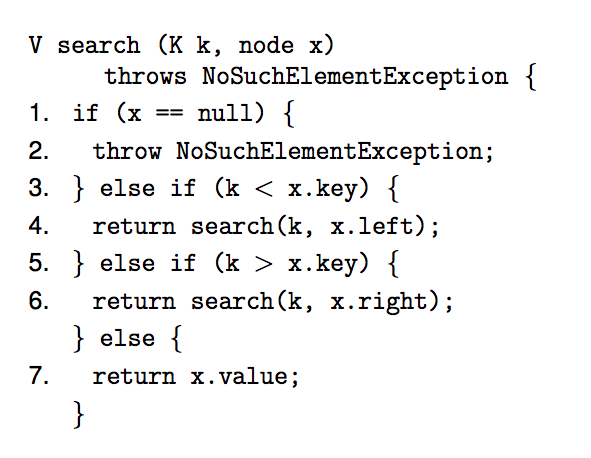
(b) You were next asked to consier the following sorted array.

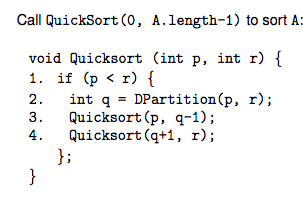
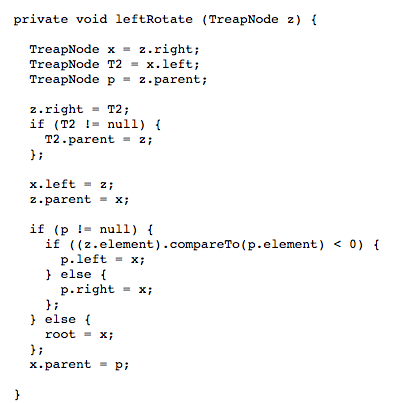


(

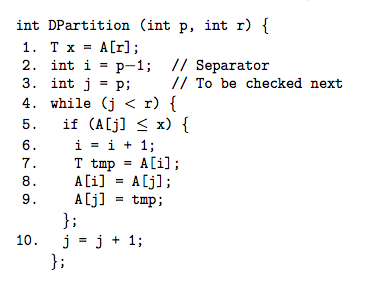
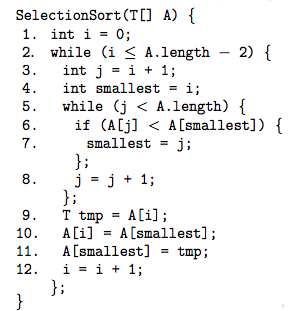
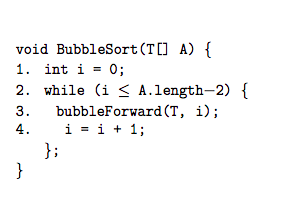
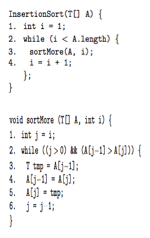
Heaps:

A screenshot of a cell phone

Description automatically generated********



f(high, low) = high − low + 1

****

**Dijkstra’s**: builds the shortest path tree from some source. Connects all nodes in a graph back to source with length being minimized. Worst case number of steps is O((|V| + |E|) log |V|) (same for Prims). Doesn’t necessarily work for negative edge weights. Does directed and undirected.

**Prims**: Constructs a minimum cost spanning tree, the tree with the least total cost of all the nodes. Length isn’t minimized. Find the least weight from current node, goes there, updates values, then goes again. Works only on undirected graphs. Can deal with negative edges.

**Full Trees:**

Definition: A free tree is a connected acyclic graph. Frequently we just call a free tree a “tree.” If we • identify one vertex as the root, and use this (with the existing edges) to identify the parent and children of each vertex, and • assign an ordering to the children of each vertex, then the result is the kind of “rooted tree” we have seen before

**Cyclic and Acyclic:**

Definition: A cycle (in an undirected graph

G = (V, E) is a path with length greater than zero from some vertex to itself: A graph G = (V, E) is acyclic if it does not have any cycles

**Spanning Trees:**

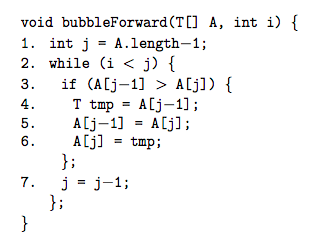
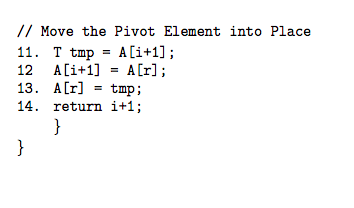
If G = (V, E) is a connected undirected graph, then a spanning tree of G is a graph

G^ = (V^, E^) such that

- V^ = V

- E^ is a subset of E

- G^ is a tree

**Macintosh HD:Users:egorushakov:Desktop:Screen Shot 2019-12-11 at 9.31.38 PM.png**

**Induced Subgraphs:**

G^ = (V^, E^) is a subgraph of G if G^ is a graph such that

- G^ is a subgraph of G and, furthermore, G^ includes all of the edges from G that it could possibly have.

**Adjacency-List Representation**

- The adjacency-list representation of G = (V, E) consists of an array AdjG of |V| lists — one for each vertex in V.

- For each u ∈ V, the adjacency list AdjG (u) contains (references to) all the vertices v ∈ V such that (u, v) ∈ E.

This representation...

• is space-efficient if G is sparse; • is not really space-efficient if G is (extremely) dense; • does not support simple operations particularly well: • Checking whether a pair of vertices are neighbours requires more than constant time — the number of steps used for this is linear in the degree of one of the inputs, in the worst case. • Adding or deleting an edge also requires this cost (if error checking is to be included, as it generally should), • Iterating over the set of neighbours of a vertex is efficient — the number of steps used is linear in the degree of the input vertex. This is a useful representation if G is large and sparse; but not if small or dense.

**Merge Sort:** Recursive. Does not sort in place, huge storage requirements. Breaks it in half over and over until each element is an individual array. Then merges those arrays together and orders then while doing it.

**Heap Sort:** Parent is always larger than its children. (i-1)/2 is the first parent. Full heap size is 2^(h+1)-1. If T is a heap with height >= 0, then size of T is 1) >= 2^h and 2) <= 2^(h+1)-1. Every heap with size n>= 1 has height Omega(log2n). Worst case running times are linear in the height of an input heap. Maxheap property: each node is greater than or equal to values of children. Minheap is opposite. Parent = (i-1)/2. Left child = 2i +1, right = 2i+2. His insert puts it at the end and then bubbles it up. His delete gets rid of it and then bubbles down.

**Quick sort:** Pivot. Move pivot to the end. From left find first element that’s larger, from right find first element that’s smaller. Swap those two. Repeat this until item from left and item from right cross. Swap the pivot with the item from left. Pivot is in the right position, everything to the left is smaller, everything to the right is larger. Recurse over these two partitions. Picking the pivot is important. Fastest known algorithm on most inputs BUT worst case is significant higher than worst case of Merge or Heap sort. If you need to guarantee worst-case performance, don’t use Quicksort.

Bound Function for This Loop:

• Depth of z in T — i.e., distance from root to z — if T is not (yet) a red-black tree

• 0, if T is now a red-black tree.

Claim 1: For each node x, the size of the set represented by the subtree with root x is at least 2 bh(x) − 1.

Theorem:

If T is a red-black tree storing an ordered set or dictionary with size n then the depth of T is at most 2 × log2 (n + 1)

Every red-black tree satisfies the following Red-Black Properties.

1. Every node is either red or black. 2. The root is black.

3. Every leaf (NIL node) is black.

4. If a node is red, then both its children are black.

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

The depth of every red black tree with n internal nodes is always in