CITS 2200
SEMESTER 1 2017

Algorithms

- Steps you take to manipulate data
- ***Data structures:*** *the way data is arranged in memory*
- Main data structure operations
    - Inserting
    - Deleting
    - Searching

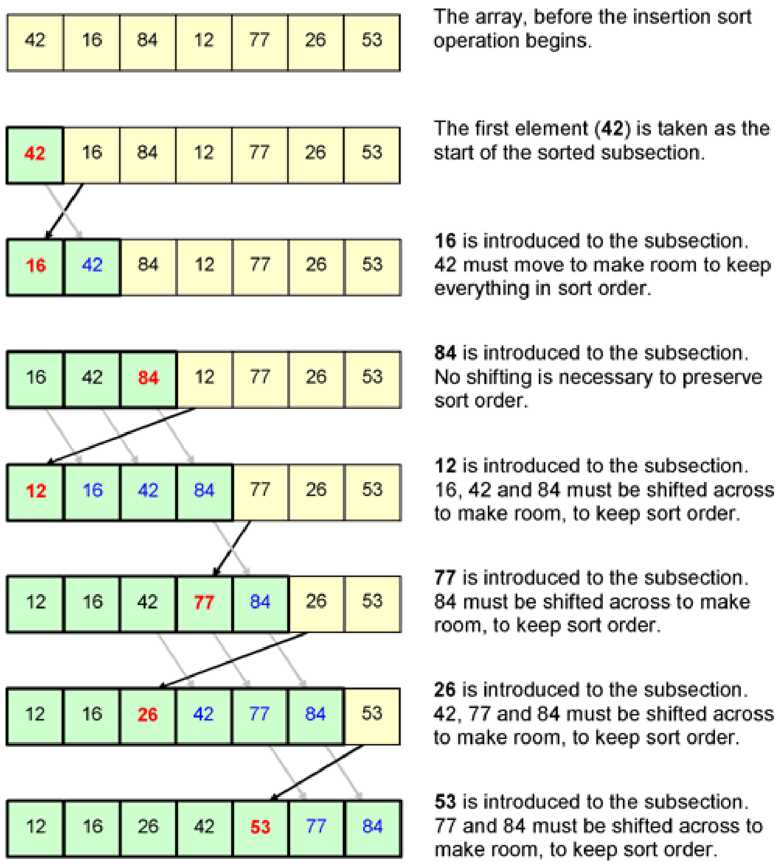
Sort Algorithms

Insertion sort

Searches through an array, finds a minimum and puts it precisely into place (ie. straight to index 0 for ascending order). Left side must always be already sorted, and the right side unsorted.

```
public void insertionSort(long[] a){
            for(int j = 1; j < a.length; j++){
              long key = a[j];
              int i = j - 1;

              while(i>=0 && a[i]>key){
                  a[i+1] = a[i];
                  i=i-1;
                  count++;
              }
              a[i+1]=key;
              count++;
              }
    }
```



42 16 84 12 77 26 53	The array, before the insertion sort operation begins.
42 16 84 12 77 26 53	The first element (**42**) is taken as the start of the sorted subsection.
16 42 84 12 77 26 53	**16** is introduced to the subsection. 42 must move to make room to keep everything in sort order.
16 42 84 12 77 26 53	**84** is introduced to the subsection. No shifting is necessary to preserve sort order.
12 16 42 84 77 26 53	**12** is introduced to the subsection. 16, 42 and 84 must be shifted across to make room, to keep sort order.
12 16 42 77 84 26 53	**77** is introduced to the subsection. 84 must be shifted across to make room, to keep sort order.
12 16 26 42 77 84 53	**26** is introduced to the subsection. 42, 77 and 84 must be shifted across to make room, to keep sort order.
12 16 26 42 53 77 84	**53** is introduced to the subsection. 77 and 84 must be shifted across to make room, to keep sort order.

Efficiency

Insertion sort can be quite efficient for use in small sample sizes. It is preferable to quicksort and mergesort when dealing with small arrays:

$O(n^2)$: *worst case*

$O(n^2)$: *average case*

$O(1)$: *best case*

Quick Sort

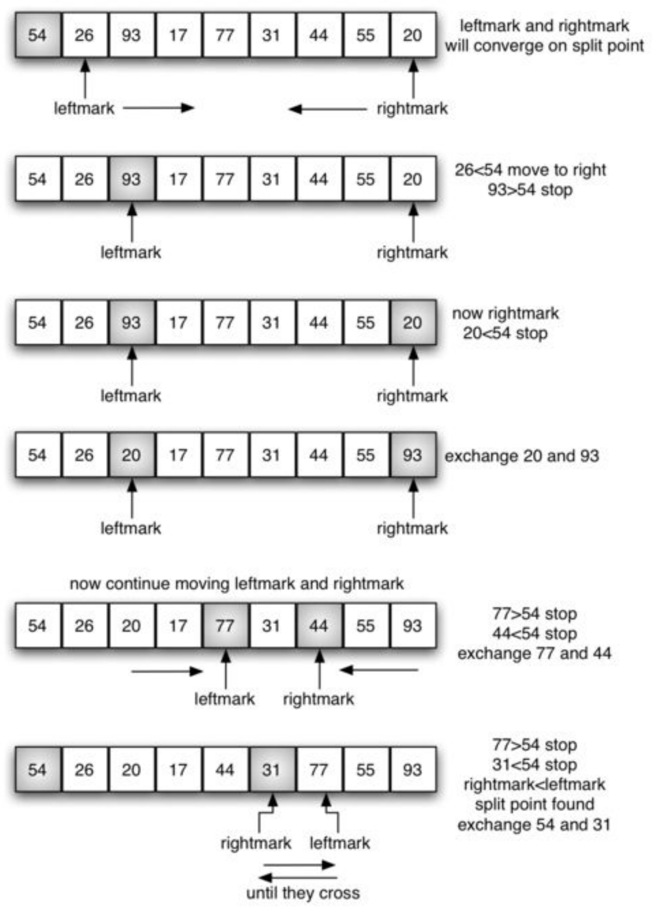
In most case quicksort is the fastest sorting algorithm. Works by partitioning the arrays so that smaller numbers are on the left and larger are on the right.

```
 public void quickSort(long[] a){
        quickSort(a, 0, a.length-1);
    }

private int partition(long[] a, int p, int r){
        long x=a[r];
        int i = p-1;
        for(int j = p; j < r; j++){
            if(a[j]<=x){
                i++;
                exchange(a,i,j);
                count++;
            }
        }
        exchange(a,i+1,r);
        count++;
        return i+1;
    }

private void exchange(long[] a,int x, int y){
        long temp = a[x];
        a[x]=a[y];
        a[y]=temp;
    }

private void quickSort(long[] a, int p, int r){
        if ( p < r ){
            int q = partition(a,p,r);
            quickSort(a,p,q-1);
            quickSort(a,q+1,r);
        }
    }
```



Efficiency

Quicksort *can* be the most efficient algorithm for sorting, but especially for large arrays.

$O(n \log n)$: *average*

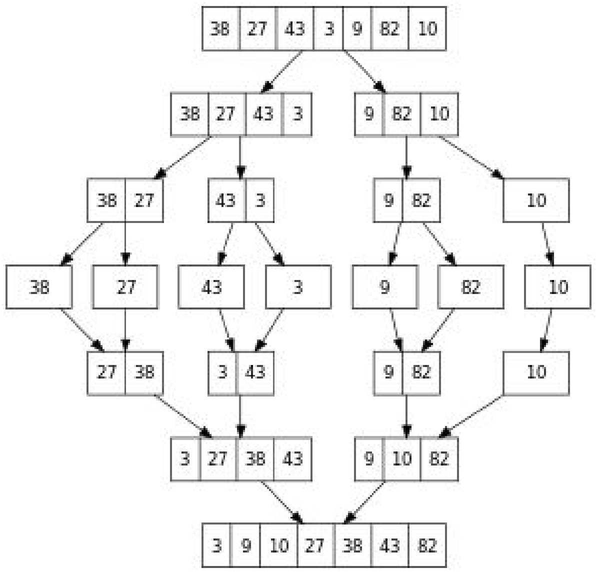
$O(n^2)$: *worst case*

Merge sort

Merge sort is an efficient sorting algorithm. It simply splits the array into smaller and smaller sub arrays. It then sorts the different arrays and merges them into larger arrays by sorting the small arrays together.

```
public void mergeSort(long[] a){
        mergeSort(a, 0, a.length-1);
        }
```

```
private void merge(long[] a, int p, int q, int r)
{
int n = q-p+1;
int m = r-q;
long[] an = new long[n];
long[] am = new long[m];
for(int i = 0; i<n; i++) {
an[i] = a[p+i];
count++;
}
for(int i = 0; i<m; i++){
am[i] = a[q+i+1];
count++;
}
int i = 0;
int j = 0;
for(int k = p; k<=r; k++){
if(i==n) a[k] = am[j++];
else if(j==m || an[i]<am[j]) a[k] = an[i++];
else a[k] = am[j++];
count++;
}
}
private void mergeSort(long[] a, int p, int r)
{
if(p<r){
int i = (p+r)/2;
mergeSort(a,p,i);
mergeSort(a,i+1,r);
merge(a, p,i,r);
}
}
```



Efficiency

Mergesort always gives the same efficiency no matter the situation. It carries out all the same operations even if the array is already sorted. Due to this it can be efficient for large arrays, but not as efficient as insertion sort for smaller algorithms. Uses more space than the other algorithms.

$O(n \log n)$ *time* : *average*

$O(n)$ *space*

Data structures

Abstraction

Recursive data structures:

- can be arbitrarily large
- support recursive programs
- are a fundamental part of computer science

Stack

Allows access to only the most recent item of the stack. Similar to a stack of paper only showing the top sheet (ie. the most recent one). Stacks are Last in First out (LIFO).

Push

Puts a new item on top of the stack:

```
public void push(Object o)throws Overflow{
       if(!isFull()){
           topStack++;

           stack[topStack] = o;
        }
        else{
            throw new Overflow("This stack is full!");
        }
    }
```

Pop

Removes item from the top of a stack and returns the object that was deleted:

```
public Object pop() throws Underflow{
   if(!isEmpty()){
           Object retObject = stack[topStack];
           stack[topStack] = null;
           topStack--;
           return retObject;
        }
       else{
           throw new Underflow("The stack is empty!");
        }
    }
```

Peek

Shows the item from the top of the stack:

```
public Object examine() throws Underflow{
       if(!isEmpty()){
           return stack[topStack];
        }
       else{
           throw new Underflow("The stack is empty!");
        }
```

Queues

Allows you to access the first item added to the que. Different to a stack in the sense that stacks are Last In First Out, queues are First In First Out (FIFO). Can be compared to a line at a movie (ie. first in line goes in first).

Enqueue

- Adds an element to the back of the queue

Dequeue

- Removes and returns element from front of the queue

Peek

- Views the element at the front of the line without removing it
- Is useful because if the element is removed you cannot replace it without putting it to the back of the queue.

Lists

A series of elements which link (reference) to each other. These are distinguished from an array because the array has an index (`a[3]` will take you to the third element), whereas to go through a linked list you must go through each element (will take linear time). Insertions and deletions can be quite simple.

In a singly linked list a node keeps its own value, and the value of the next item (ie. references the Next). In a doubly linked list the node keeps its own value, and the value of both the previous and next node.

Insert

1. Creates a new node with its value.
2. New node takes its “next” value to be the value of the next node.
3. New node takes its “prev” value to be the value of the previous node.
4. Changes the previous node's “next” reference to the new nodes value
5. Changes the next node's “prev” reference to the new nodes value

Delete

1. Make the deleted nodes next node’s “prev” reference to the deleted nodes “prev” reference
2. Make the deleted node's “prev” reference to the deleted node's “next” reference

Search

Linked lists really are not meant to be searched. This is because the efficiency will take linear time ($O(n)$).

Deque

Double Ended Queue. Ends can be defined as left and right and the same operations from queue can be done on a deque but from both ends.

Insert Left/Insert right

- Adds a new element to either the leftmost or rightmost position

Delete left/Delete right

- Deletes either the leftmost or rightmost element.

Complexity

Empirical measurement: *We can find the complexity of things based on experimental data*

Analytical measurement: *construct mathematical or theoretical models to estimate performance.*

Time units can be used to find the theoretical measurement of complexity - Assigning a method call 2 units, variable assignments 1, and other operations 1 we can calculate the number of time units

Growth rates

The rate of growth is far more important than the exact numbers. We can abstract this from:

- Ignoring specific values of input and just consider the number of items or the size of input
- Ignoring precise duration of operations and consider the number of specific operations as an abstract measure of time
- Ignore actual storage space occupied by data elements and consider the number of items stored as an abstract measure of space

Asymptotic analysis

Worst case analysis - *we can chose data which will have the largest time/space requirements. THis can be relatively simple and give a guarantee on the upper bound of how bad the algorithm can be, however could be unrepresentative of the regular operation of the system.*

Expected case analysis - *The average or expected case. Is more realistic of what to normally expect, however can be difficult to calculate, and does not give us any clues on what to expect normally.*

Asymptopia gives a simple description of behavior for use in comparison.

Big O notation

Example.

Prove that n^2 is $O(n^3)$.

For some constant $c > 0$ and $n_0 \geq 1$,

$n^2 \leq cn^3$

Which infers that $1 \leq cn$

For all $n \geq n_0$

Choosing $0 = n_0 = 1$ satisfies this inequality

Amortized analysis

Amortized analysis is a variety of worst case analysis, but rather looks at the cost of doing the operation once. That is that we must complete the operation n times, and divide the cost by n.

$$\frac{T(n)}{n}$$

Amortized is different than asymptotic in that some processes may not change their cost per item added.

Iterators

Used to traverse a collection. Iterators, compared to recursion simply loops through, rather than creating a new method call each time. Interfaces have the following methods:

hasNext(): returns true if the iterator has more items

next(): if there is a next item, return that item and advance to the next position, otherwise throw an exception

remove(): remove the underlying collection from the last item returned by the iterator. Can only occur after next() is called.

Fail fast iterators - *If a backing collection changes during use of an iterator the we can disallow further use of the iterator (throw exception) when an unexpected change to the backing collection has occurred.*

Trees

Binary tree - *an indexed tree T of n nodes*.

Leaf - *an internal node whose left and right subtrees are both empty (external nodes).*

Skinny tree - *Every node has at most one child*

Complete - *external nodes appear on at most two adjacent levels.*

Forest - *sets of trees*

Orchard - *a queue or list of trees.*

THere is a formalised relationship between the height and size of graphs:

- A binary tree of height h has size at least h.
- A binary tree of height h has size at most $2^h - 1$
- A binary tree of size n has height at most n.
- A binary tree of size n has height at least $[log(n + 1)$

Binary trees either have no items or consist of a root node u and two binary trees u(1) and u(2). The function u is called the index.

Graphs

Just like with linked lists and trees, graphs are just a set of objects with links. The internet can be considered a graph - where each site is a node, and each link is an edge.

Basic Objects - *Vertices, nodes (Labelled by integers)*

Relationships between them - *edges, arcs and links ()*

Undirected graphs - *The edges are both ways, have no direction*

Directed graph - *For every edge there is an end point, depends on direction.*

Weighted graphs - *Labelled edge that describes the distance, or cost of taking an edge.*

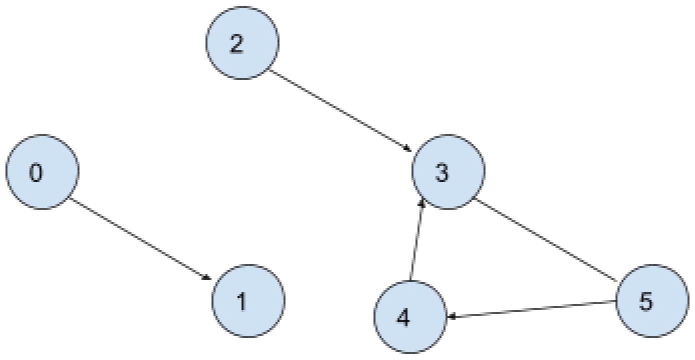
Path - *sequence of vertices and edges that depicts hopping along a graph.*

Connected - *A graph G is connected if there is a path between any two vertices. If the graph is not connected then its connected components are the maximal induced subgraph that are connected.*

Forest - *A forest is a graph with no cycles*

Trees - *A tree is a forest with only one connected component. It is easy to see a tree with n vertices must have exactly n-1 edges.*

An edge matrix (adjacency matrix) creates a table full of the links and their weights for showing links between nodes on the graph. For example 1 shows a link between two nodes, 0 shows no link or is linking to itself. The following graph and edge matrix are shown.



0	**1**	0	0	0	0
0	0	0	0	0	0
0	0	0	**1**	0	0
0	0	0	0	0	0
0	0	0	**1**	0	0
0	0	0	**1**	**1**	0

An edge list (adjacency list) is an alternative to the adjacency matrix. The above graph can be represented in the following list:

0→{1}

1→null

2→{3}

3→null

4→{3}

5→{3,4}

Graph traversals

Traversing a graph can be used for searching for a particular item, testing equality, copying, creating and displaying. The two most simple types of traversing are as follows:

***Depth-first* -** *Follow branches from root to leaves*

***Breadth-first(level-order)* -** *Visit nodes level by level*

Both of these traversals take $O(n)$ time for a binary tree.

Depth First search

Preorder traversal: *Common garden "Left to right", "backtracking", depth first search!*

Space required for this algorithm is proportional to the height of the tree. For a skinny tree of size n this would require n amounts of space.

Breadth first search

Breadth first search uses a spanning tree for G, called the breadth-first tree. It uses a queue as its main data structure.

1. Push v to the tail of Q
2. While q is not empty
3. Pop vertex w from the head of q
4. For each vertex x adjacent to w
    a. If the color is white then
        i.

Priority queues

A priority queue is an extension of the queue ADT. Each item is assigned an integer priority to indicate the relative importance of the item. Instead of storing items in chronological order, a priority queue archives items with the highest priority before all others.

enqueue (): operation is performed by iterating through the queue from the front to the back until the correct location is found to insert the new element is found. In the worst case, the entire queue will be examined $O(n)$ where n is the size

Examine(): This operation simply returns the element at the front of the queue $O(1)$.

Dequeue(): This operation returns the element at the front of the queue and updates the value at the front $O(1)$

Minimum spanning tree

A minimum spanning tree will be a subgraph of a main graph that is a:

- A spanning subgraph
- A tree
- Has a lower weight than any other spanning tree

Greedy algorithms - *A greedy algorithm is an algorithm that locally chooses the optimal choice.*

Kruskals algorithm

This algorithm is based on building a forest of lowest possible weight and continuing to add edges until it becomes a spanning tree. F is a forest with G vertices but none of the edges:

1. Pick an edge e of minimum possible weight
2. If F union e is a forsest then:
    a. F becomes F union e
3. Repeat until F contains n-1 edges

$O(E \log V)$

Prims algorithm

Prim's algorithm is another greedy algorithm for finding a minimum spanning tree. The idea behind prim's algorithm is to grow a minimum spanning tree edge-by-edge by always adding the shortest edge that touches a vertex in the current tree.

$O(V \log V + E \operatorname{Log} E) = O(E \log V)$

Shortest path algorithm

Dijkstra's algorithm

A priority-first search that can solve the shortest path problem in $O(E \lg V)$ provided all graph edges have nonnegative edge weights

Bellman-ford

Can solve all shortest path problems in $O(EV)$.

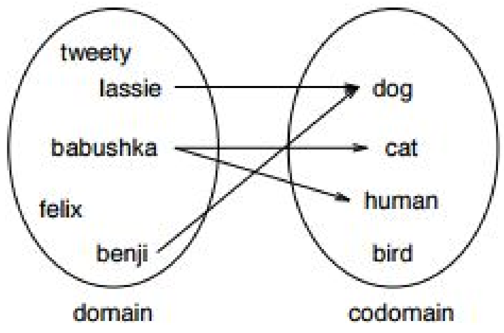
Maps

Relations- *sets of tuples*

Binary relation- *sets of pairs (2-tuples). There is often no way of sorting a binary relation.*

Domain - *set of values which can be taken on by the first item of a binary relation. Essentially a key.*

Codomain - *a set of values which can be taken on by the second item of a binary relation*



A map or a function is a binary relation in which each element in the domain is mapped to at most one element in the codomain (many-to-one).

Partial map - *not every element in the domain has an image under the map (ie. the image is undefined for some elements). For example, in the above image felix has no animal type.*

Affiliation = { <Turing , Manchester >
< Von Neumann , Princeton >
< Knuth , Stanford >
< Minsky , MIT >
< Dijkstra , Texas >
< McCarthy , Stanford >}

Implementation

Constructors

1. Map(): creates a new map that is undefined for all domain elements

Checkers

2. isEmpty(): return true if the map is empty(undefined for all domain elements)
3. isDefined(d): return true if the image d is defined, false otherwise.

Manipulators

4. assign(d,c): assign c as the image of d.
5. image(d): return the image of d if is defined, otherwise throw an exception.
6. deassign(d): if the image of d is defined return the image and make it undefined, otherwise throw an exception.

List based representation

A map can be considered to be a list of pairs. Providing this list is finite, it can be implemented using of of the techniques used to implement the list ADT. A map can be built using the list ADT.

```
public class MapLinked {
      private ListLinked list;
      public MapLinked () {
          list = new ListLinked();
     }
}

public class Pair {

     public Object item1; // the first item (or domain item)
     public Object item2; // the second item (or codomain item)
     public Pair (Object i1, Object i2) {
          item1 = i1;
          item2 = i2; }

// determine whether this pair is the same as the object passed //
assumes appropriate ‘‘equals’’ methods for the components
public boolean equals(Object o) {
     if (o == null) return false;
     else if (!(o instanceof Pair)) return false;
     else return item1.equals( ((Pair)o).item1) && item2.equals(
     ((Pair)o).item2); }

     // generate a string representation of the pair
     public String toString() {
     return "< "+item1.toString()+" , "+item2.toString()+" >";
     }
}

public Object image (Object d) throws ItemNotFound {
     WindowLinked w = new WindowLinked();
     list.beforeFirst(w); list.next(w);
     while (!list.isAfterLast(w) &&
          !((Pair)list.examine(w)).item1.equals(d) ) list.next(w);
     if (!list.isAfterLast(w)) return ((Pair)list.examine(w)).item2;
     else throw new ItemNotFound("no image for object passed"); }
```

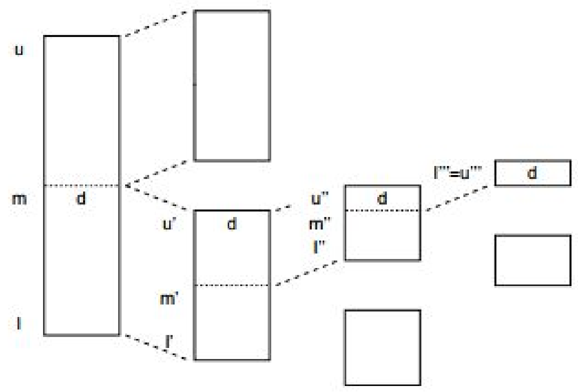
Performance

Map and isEmpty make trivial calls to a constant-time list ADT commands. The other four operations all require a sequential search within the list which is linear in the size of the defined domain (O(n))

Operation	Complexity
Map	1
isEmpty	1
isDefined`	n
assign	n
image	n
deassign	n

Sorted-block representation

Some of the above operations take linear time because they need to search for a domain element. The above program does a linear search. An algorithm for binary search is as follows:



Binary searches are essentially splitting the array in two, finding if what we are searching for is greater than or less than our half, the splitting the half we are looking in and repeating the process.

```
private Pair[] block;

protected int bSearch (Object d, int l, int u) {
      if (l == u) {
            if (d.toString().compareTo(block[l].item1.toString()) ==
0)
                  return l;
      else return -1;
}

else {
      int m = (l + u) / 2;
      if (d.toString().compareTo(block[m].item1.toString()) <= 0)
            return bSearch(d,l,m);
      else return bSearch(d,m+1,u);
      }
}
```

Performance

One way of looking at the complexity is to consider the biggest list of pairs we can find a solution for with m calls to bSearch.

Calls to bSearch	Size of list
1	1
2	1+1
3	2+1+1
4	4+2+1+1
m	2^{m-1}

Which means that the complexity is $O(log\ n)$. If the map can be ordered the complexities overall become far better. isDefined and image simply require a binary search, therefore become $O(log\ n)$.

Operation	Complexity
Map	1
isEmpty	1
isDefined	Log n
assign	n
image	Log n
deassign	n

The sorted block may be the best choice if:

- The map has a fixed maximum size
- The domain is totally ordered
- The map is fairy static (mostly isDefined, and image. Rather than writing assign, deassign)

Sets, tables and dictionaries

Sets:

- Used when set-theoretic operations are required
- Elements may or may not be ordered
- Includes “membership” operations: isEmpty, insert, delete, isMember
- Includes “set-theoretic operations: union, intersection, difference, size, complement
- *“I have one set of students who do CITS2200 and one set of students who do CITS2210. What is the set of students who do both?”*

Table

- Simpler version of set without the set-theoretic operations
- Elements assumed to be unordered

- *"I begin with the set of students originally enrolled in CITS2200. These two students joined. This one withdrew. Is a particular student currently enrolled?"*

Dictionary

- Like able but assumes elements are totally ordered
- Includes "order related" operations: isPredecessor, isSuccessor, predecessor, successor, range.
- *"Here is the set of students enrolled in CITS2200 ordered by (exact) age. Which are the students between the ages of 18 and 20?"*

Element - *may be a single item or "records" with unique keys (such as those in a database).*

Set implementation

Set(): create an empty set.

isEmpty(): returns true if the set is empty, false otherwise

isMember(e): returns true if e is a member of the set, false otherwise.

size(): returns the cardinality of (number of elements in) the set.

complement(): returns the complement of the set (only defined for finite universes)

insert(e): forms the union of the set with the singleton {e}

delete(e): removes e from the set

union(t): returns the union of the set with t

intersection(t): returns the set obtained by removing any items that appear in t.

enumerate(): returns the "next" element in the set. Successive calls to enumerate should return successive elements until the set is exhausted.

Block Implementation

Assume A is a set from some universe U. A set can be viewed as a boolean function. The characteristic function maps this sequence to a sequence of 1s and 0s, thus the set can be represented as a block of 1s and 0s, or e bit vector:

e1	e2	e3	e4	e5	e6	...
1	1	0	0	1	0	...

$1 \lor 1 = 1$ - logic or

$1 \lor 0 = 1$

$0 \lor 1 = 1$

$0 \lor 0 = 0$

$1 \land 1 = 1$ - logic and

$1 \land 0 = 0$

$0 \land 1 = 0$

$0 \land 0 = 0$

Advantages

Insert, delete and isMember - All of these things can be done in constant time providing the index can be calculated in constant time.

Complement, union, intersection, difference - O(m); linear in size of universe

Enumerate - O(m) for n calls, where n is the size of the set, this means that O(m/n) amortized over n calls.

Disadvantages

If the universe is large compared to the size of the sets then:

- The latter operations are expensive

- Large amounts of space wasted
- Requires the universe to be bounded, totally ordered, and known in advance

List representaion

An alternative is to represent the set as a list using one of the list representations.

Performance

Assume we have a set of size p.

Insert, delete, isMember - take O(p) time; the best that can be achieved in an unordered list

Union - for each item is the first set, check if it is a member of the second, and if not, add it (to the results)

Ordered list representation

If the universe is totally ordered, we can obtain more efficient implementations

Performance

Each list is traversed once in O(p+q) time. This is much better than O(pq).

Table

Table(): create an empty table

isEmpty(): returns true if the table is empty, false otherwise

isMember(e): returns true if e is in the table, false otherwise

insert(e): forms the union of the table with the singleton {e}

delete(e): removes e from the table

Table representations

Since the table operations are a subset of those of set, the unordered list representation can be used. The more efficient list representation and the characteristic function representation are not available since the elements are assumed to be unordered.

Dictionary

Dictionary(): creates an empty dictionary

isEmpty(): returns true if the dictionary is empty, false othewise

isMember(e): returns true if e is a member of the dictionary, false otherwise

isPredecessor(e): returns true if there is an element in the dictionary that precedes e in the total order, false otherwise.

isSuccessor(e): returns true if there is an element in the dictionary that succeeds e in the total order false otherwise.

insert(e): adds e (if not already present)

predecessor(e): returns the largest element in the dictionary that is smaller than e if one exists

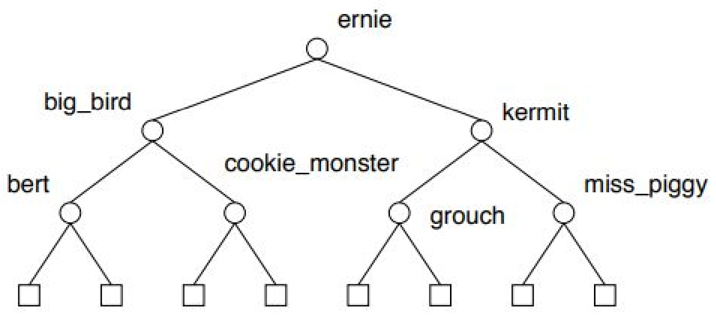
successor(e): returns the smallest element in the dictionary that is larger than e if one exists

range(p,s): returns the dictionary of all elements that lie between p and s (including p and s if present)

delete(e): removes item e from the dictionary

Binary search tree implementation

We can use binary search tree. A binary tree is a tree whose internal nodes are labelled with elements (or their keys) such that they satisfy the binary search tree condition: For every internal node u, all nodes in u’s left subtree precedes u in the order of all nodes in u’s right subtree succeed u in ordering.

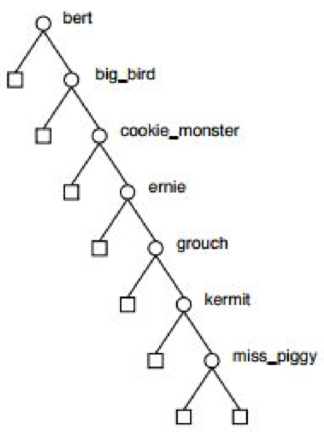


In the above example the characters with letters greater than e (for ernie) are in the left subtree, and all over e (for ernie) are in the right subtree. This is just like a normal dictionary, all words come in order.

Array lists are efficient in a normal case setting, but binary search trees are efficient in the worst case sense. A tree can guarantee the optimal efficiency in all cases.

Performance

The performance of a binary tree really depends on the shape of it.



The above is the worst case scenario for a binary search tree. While it is perfectly a binary search tree it will give worst complexity. THe best case is a perfect binary tree, the worst case is called a skinny binary tree.

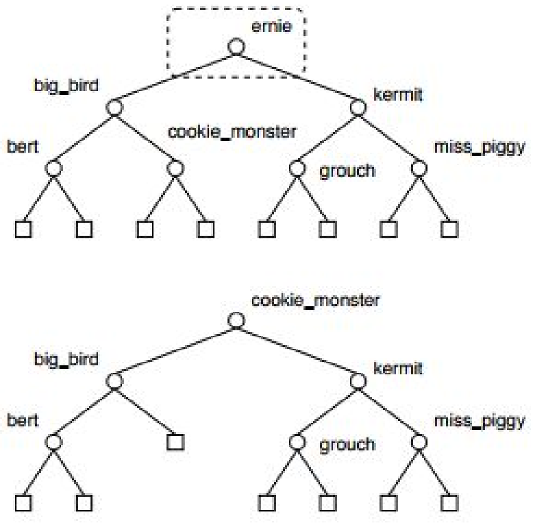
Insert is a fairly straightforward method

- Performs a search for the element
- If the element is found end
- If not found insert a new node containing the element

Delete is just as straightforward if the element is found on a node with atleast one external child - this is just the same as the standard Bintree delete operation

Otherwise:

- Replace the deleted element with its predecessor - note that the predecessor will always have an empty right child
- Delete the predecessor

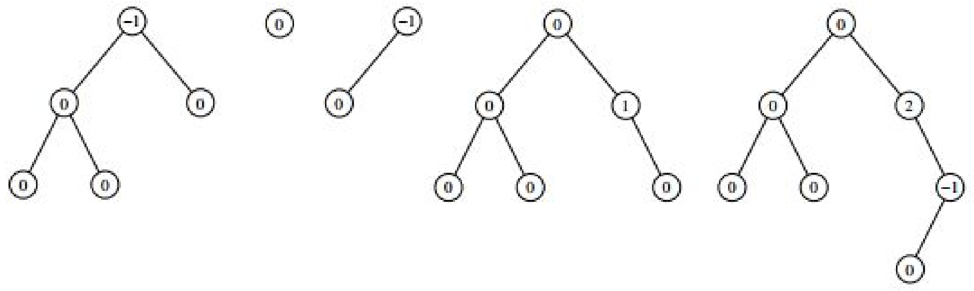


Balancing trees

The delete procedure has a tendency over time to skew the tree to the right - as we have seen this will make it less efficient. We can alternate between replacing with predecessor and successor - This is generally not a desirable thing to do. It is normally desirable to keep the tree balanced or complete. There are a number of data structures that are designed to keep trees balanced - B-trees, AVL-trees and Red-Black trees.

AVL Trees

An AVL tree is a binary search tree where, for every node, the height of the left and right subtrees differ by at most one. This means the depth of any external node is no more than twice the depth of any other internal node.



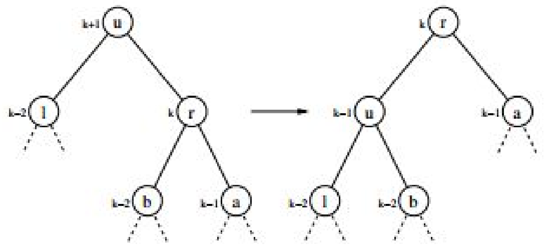
AVL trees have worst case O(log n) time performance for searching inserting and deleting. Searching an AVL tree is the same as for a normal binary tree, however the insertion and deletion operation must be modified to maintain the balance of the tree.

For an insertion to occur we must first find the appropriate place (a leaf node) to add the new element. If the insertion makes the tree unbalanced, then we locally reorganize the tree via a rotation.

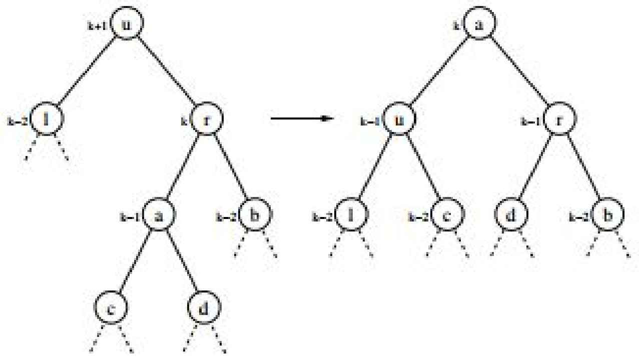
1. Insert the element into an external node
2. Start with its parent, check each ancestor of the node to ensure the left and right subtree heights differ by less than two
3. If we find a node such that one child has height k-2 and the other has height k, we must perform a rotation to restore the balance.

Rotations can occur in two cases:

1. Suppose u is a node where its left child l has a height of two less than the right child r, and the right child r has height one more than the left child of r.
   We arrange the tree as follows.



2. Suppose u is a node where its left child l has height two less than its right child r, and the left child of r has height one more than the right child of r.
   We arrange the tree as follows

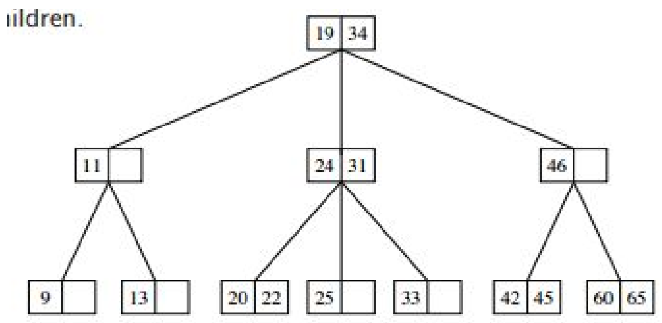


These rotations are all done to maintain the fundamental part of a binary tree that: What is in the left of the parent node **MUST** be less than the parent node, and the right child **MUST** be more than the parent node.

Rotations are constant time operations. Insertions and deletions require a search O(h) and then checking every ancestor of that element O(h) (where h is the height of the tree). The height of an AVL tree is less than 2log n where n is the number of elements in the tree.

B-Trees

A B-tree is a tree data structure that allows searching, insertions and deletion in amortized logarithmic time. Each node of a B-tree can contain multiple items and multiple children (these numbers can be varied to tweak performance). We will consider 2-3 B-trees, where each node can contain up to two items and up to three children.



If there is just one node, then the B-tree is organised as a biary tree: all itmens in the left sub-tree must be less than the item in the node, and all items in the right sub-tree must be greater. If there are two elements in the node then:

- All elements in the left sub-tree must be less than the smallest item in the ndoe
- All items in the middle sub-tree must be between the two items in the node
- All elements in the right sub-tree must be greater than the largest item in the node
- Every non-leaf node must have atleast two successors and all leaf nodes must be at the same level.

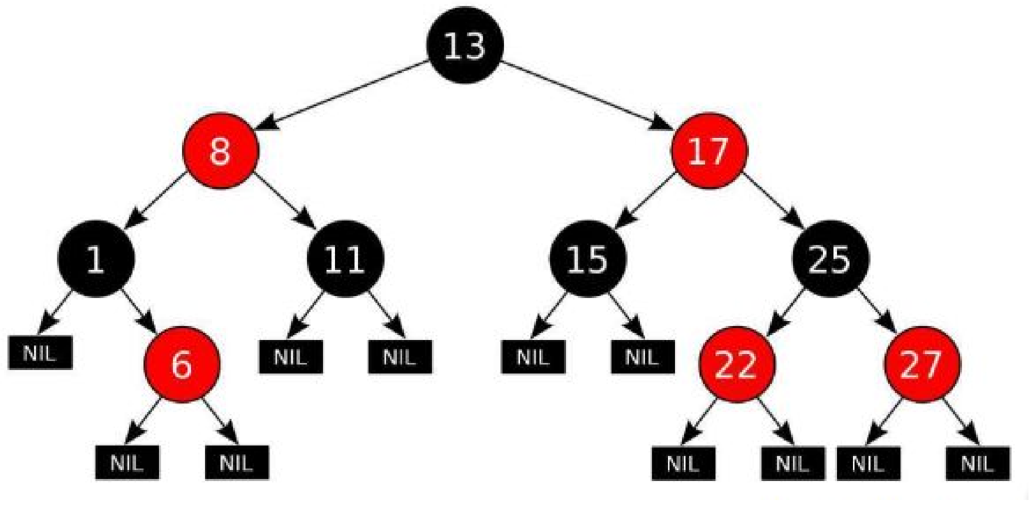
Red-Black trees

A red-black tree is another variation of a binary search tree that is slightly more efficient than B-trees and AVL trees.

A red-black tree is a binary tree where each node is either coloured either red, or black such that the coloring satisfies:

- The root property - root is black
- The external property - every external node is black
- The internal property - the children of a red node are black

- The depth property - every external node has the same number of black ancestors


13
8
17
1
11
15
25
NIL
6
NIL
NIL
NIL
NIL
22
27
NIL
NIL
NIL
NIL
NIL
NIL