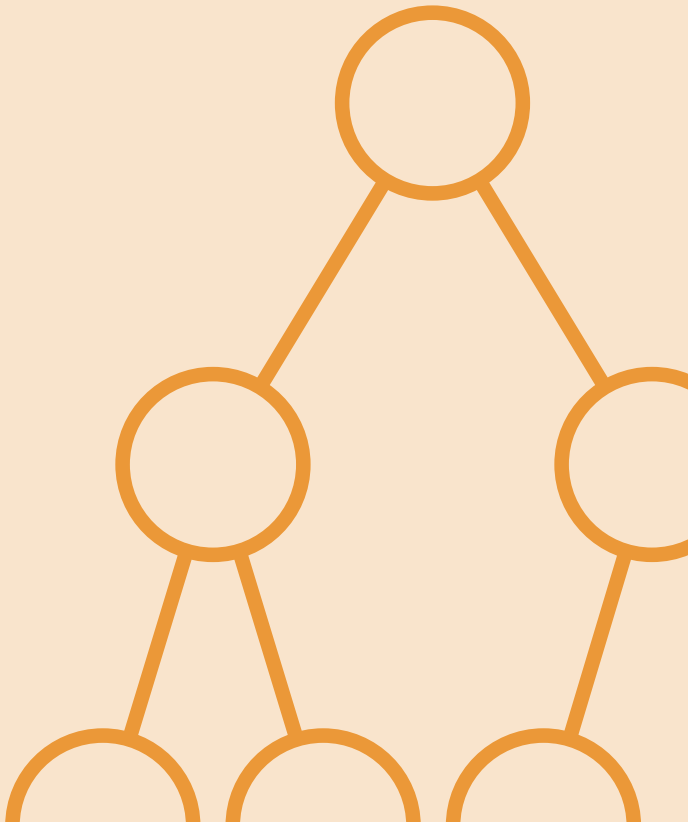
Data Structures and Analysis



Contents

Tutorial

Data Structures

Lecture 1 Abstract Data Types

1.1 Abstract Data Type

Definition 1.1.1 — Abstract Data Type. An abstract data type (ADT) is a set of mathematical objects and a set of operations on those objects. An ADT describes how information can be used in a program, which is important for specification and provides modularity and reuseability.

■ **Example 1.1 — ADT for Integers.**

- Objects: $\mathbb{Z}$
- Operations: $\text{ADD}(x,y)$ $\text{SUBSTRACT}(x,y)$, $\text{MULTIPLY}(x,y)$, $\text{QUOTIENT}(x,y)$, and $\text{REMAINDER}(x,y)$.

■

■ **Example 1.2 — Stack ADT.**

- Objects: sequences
- Operations: $\text{PUSH}(S)$ $\text{POP}(S)$, $\text{EMPTY}(S)$.

■

1.2 Data Structures

Definition 1.2.1 — Data Structure. A data structure is an implementation of an abstract data type.

■ **Example 1.3 — Data Structures for Stack.** A data structure for stacks is an array with a counter. Alternatively, a stack can be implemented as a singly linked list with the top of the stack at the beginning of the list. ■

An ADT specifies what kind of data you can have and what you can do with the data. A data structure specifies how the data is implemented; in other words, it specifies how the data is stored and how you actually do the operations on the data.

1.3 Algorithm Compleixty

The complexity of an algorithm tells us the amount of resources used by the algorithm, expressed by a function of the size of the input. Such resources can be time, space, number of messages, numbers

of bits of communication, etc. We are interested in analyzing the complexity of algorithms because it allows us to:

- compare different algorithms
- give bound to resources needed for a given input
- determine the largest size of input for which the algorithm is still efficient

The definition of input size depends on the problem that we are interested in. Below are examples of some common definitions of input size for the type of data that we are dealing with.

- integer: number of bits
- list: number of elements
- array: dimension of the array, or number of bits
- graph: number of vertices, or number of edges, or both

1.4 Dictionary ADT and Implementations

1.4.1 Dictionary ADT

In this seciton, we will take a look at an example of ADT and some implementation of it. For the dictionary ADT, the objects are defined to be the set of elements each of which has a key drawn from a totally ordered set. And the dictionary ADT should support the following operations:

- SEARCH(S,k): search the set S for an element with the key k and return a pointer to one such element. If no such element exists, return NIL.
- INSERT(S,x): insert element pointed by the pointer x into the set S.
- DELTE(S,x): delete the element pointed to by the pointer x from the set S.

Each element x will contain a property k that stores the key.

Some examples of totally ordered sets are: $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, colors, English words. The set of complex numbers $\mathbb{C}$ is not totally ordered. A more formal definition of a total order can be found in the notes on Theory of Computation (CSC 240/CSC 236).

1.4.2 Data Structures for Dictionary

There are many ways to implement a dictionary. The simplest and most common way is to use a hash table, but there are also equally valid implementations.

Hashing

SEARCH: average complexity $O(1)$, worst-case complexity $O(n)$
INSERT: average complexity $O(1)$, worst-case complexity $O(n)$

DELETE: average complexity $O(1)$, worst-case complexity $O(n)$

Array

We can use two arrays, one with keys, the other with values in the correesponding position of the keys. In the case of unsorted arrays, the time complexities of the operations are:

SEARCH: worst-case complexity $O(n)$; since unsorted, we need to perform linear search to find the element
INSERT: worst-case complexity $O(1)$
DELETE: worst-case complexity $O(1)$

Another assumption that we need to make is that the elements and keys are placed consecutively in the array. However, empty slots might be created upon deleting elements. To solve this, we simply replace the deleted element with the last element in the array. By doing so, we ensure that the number of elements in S is at most the size of the array.

Binary Search Tree

SEARCH: $\Theta(h)$ INSERT: $\Theta(h)$
DELETE: $\Theta(h)$
where h is the height of the tree.

Sorted Array with Counter

SEARCH: $O(\log n)$ using binary search
INSERT: $\Theta(n)$
DELETE: $\Theta(n)$

Unsorted Singly Linked List

SEARCH: $\Theta(n)$
INSERT: $\Theta(1)$
DELETE: $\Theta(n)$; this is because for a singly linked list, it takes $\Theta(n)$ time to find the pointer to the previous element in order to reconnect the links after deleting

Unsorted Doubly Linked List

SEARCH: $\Theta(n)$
INSERT: $\Theta(1)$
DELETE: $\Theta(1)$

Sorted Doubly Linked List

SEARCH: $\Theta(n)$; we cannot perform binary search because we don't know the length of the array
INSERT: $\Theta(n)$ to insert into the correct position to keep the list sorted
DELETE: $\Theta(1)$

Direct Access Table

If our set of keys S is a subset of some finite universe $U = \{0, 1, \ldots, m-1\}$ where $|U| = m$, then we can use a direct access table to implement a dictionary ADT. To represent the dictionary, we use an array A with m slots indexed from 0 to m, each of which corresponds to a key in S. The value of the slot $A[i]$ is the pointer to the element in the set S with the key i. The limitations of such implementation include:

- keys have to be unique
- size of the list can get arbitrarily large
- size of the list is limited

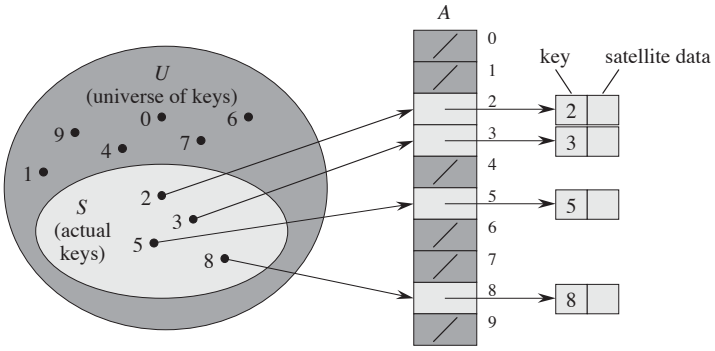


Figure 1.1: Direct access table

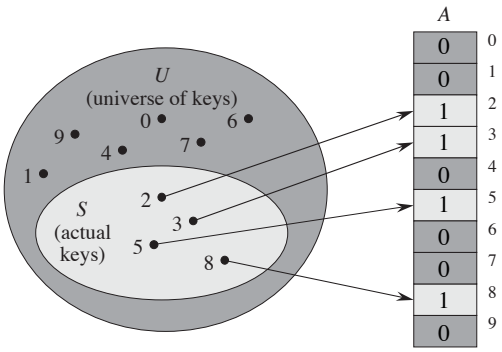


Figure 1.2: Bit vector direct access table

SEARCH(A, k)
1 **return** $A[k]$

INSERT(A,x)
1 $A[x.key] = x$

DELETE(A,x)
1 $A[x.key] = \text{NIL}$

Each of the three operations above takes constant time $O(1)$.

Alternatively, we can use value 1 to indicate the presence of element at a given slot, and value 0 to denote the absence of element at a given slot. The resulting data structure called a bit vector direct access table.

Lecture 2 Binary Search Trees

2.1 Binary Search Tree

Definition 2.1.1 — Binary Search Property. If y is in the left subtree of x, then $y.key \leq x.key$. If y is in the right subtree of x, hen $y.key \geq x.key$.

Definition 2.1.2 — Binary Search Tree. A binary tree is a binary search tree if all nodes x, y of the tree satisfies the binary search property.

If we perform an in-order traversal of a binary search tree, we gets the nodes in increasing order by key.

2.2 Searching in BST

```
SEARCH(T, k)
1  if T = NIL
2      return NIL
3  if T.key == k
4      return T
5  if T.key > k
6      return SEARCH(T.left, k)
7  if T.key < k
8      return SEARCH(T.right, k)
```

2.3 Insertion

```
INSERT(T, x)
1  X = SEARCH(T, x)
2  if X == NIL
3      X = x
4  else
5      // there is already an element with key x.k stored in BST, we can do the following
6      // 1. return without inserting x
7      // 2. continues to search until NIL is found
8      // 3. store all elements with the same key in an auxillary data structure (e.g. linked list)
9      // 4. replace old element with x
```

2.4 Deletion

Deletion is the trickiest operation for BST.

```
DELETE(T, x)
1  y = x.left
2  while y.right ≠ NIL
3        y = y.right
4  x = y
5  y = NIL
```

Lecture 3 Balanced Search Trees

3.1 Balanced Trees

There are two different ways of defining balancedness of a binary tree: weight balance and height balance. In this chapter, we will mainly focus on height balanced trees. As it turns out, weight balance is a more strict requirement than height balance, and weight balance implies height balance. Since the runtime complexity of binaryy tree operations are height-dependent, both definitions should give us $O(\lg n)$ time on most operations.

Definition 3.1.1 — Height Balancedness. A binary tree is height balanced if for every node in the tree, the height of its left and right subtrees differ by at most one.

Definition 3.1.2 — Weight Balancedness. A binary tree is weight balanced if for every node in the tree, the number of nodes of its left and right subtrees differ by at most one.

Corollary 3.1.1 Weight-balanced binary trees are height-balanced.

In this section we will look at a few height balanced search tree including red-black trees, AVL (Adelson-Velskii and Landis) trees, 2-3 trees, and B-trees which is a more general form of 2-3 trees. The first two are binary trees while 2-3 tree and B-tree are not necessarily binary.

3.2 Red-Black Tree

3.2.1 Definition and Properties

Definition 3.2.1 — Red-Black Tree. A red-black tree is a binary search tree in which every node is either red or black and satisfies the following properties:

1. The root is black
2. Every leaf node (NIL node) is black
3. If a node is red, then both its children are black
4. For each node, all paths from the node to descendant leaves (NIL nodes) contain the same number of black nodes

Alternatively, the properties can be stated without using the NIL node.

1. The root is black
2. A red node has no red children
3. Every path from the root to a node with at most one child contains the same number of black nodes

Figure 3.1 illustrates the properties of red-black trees.

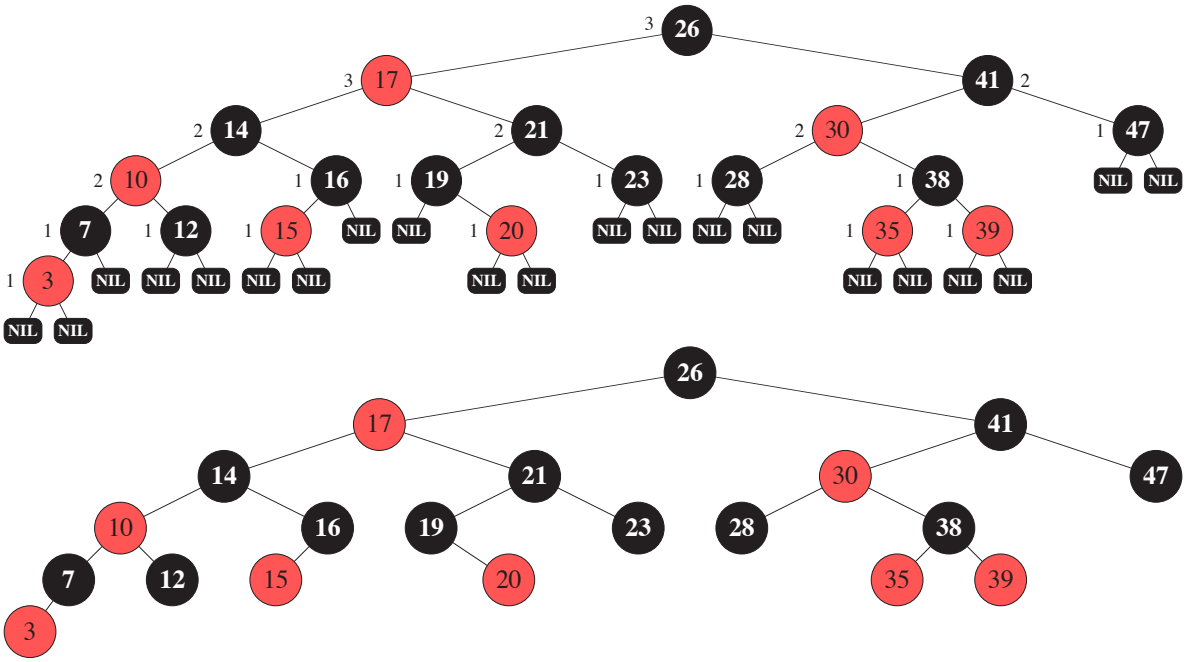


Figure 3.1: Red-black trees. The first tree is represented with the NIL sentinel node. The second tree is without the NIL node. The two trees are equivalent and both satisfies the red-black tree properties.

Lemma 3.2.1 The number of nodes in a red-black tree of height h is at least $2^{\lceil (h+1)/2 \rceil} - 1$.

Proof. A red-black tree of height h has a path of length h from the root to a leaf node. This path contains $h+1$ nodes, the first of which is black. Since it does not contain two consecutive red nodes, the path contains at least $b = \lceil (h+1)/2 \rceil$ black nodes (i.e. at least half of the nodes on that path are black). Hence, every path from the root to a node with at most one child contains at least b black nodes. It suffices to prove that there are at least $2^b - 1$ black nodes in a red-black tree of height h such that the number of black nodes in the path from the root to a leaf node is b.

BASE CASE: If the height of the tree is 0, then the number of nodes is 1.

INDUCTIVE STEP: Let $h \in \mathbb{N}$ be arbitrary. Assume that for all tree with height $h' < h$, there are $2^{b'} - 1$ black nodes where b' is the number of black nodes in the path from the root of that tree to a leaf node.

Consider an arbitrary tree T with height h with two subtrees. It follows that there are $b = \lceil (h+1)/2 \rceil$ black nodes from the root of the tree to a leaf node. If the subtree has a black root, then the path going from the root of the subtree to a leaf contains $b-1$ black nodes. If the root is red, then the path contains b black nodes. Therefore, the number of black nodes in the path from the root of the tree to a leaf

node is $b-1$ or b. Then, by induction hypothesis, the number black nodes in each subtree is at least $2^{b-1}-1$.

Since the root of a red-black tree is black, the number of black nodes in T is the number of nodes in the left subtree plus the number of nodes in the right subtree plus the root node.

$$\left(2^{b-1}-1\right)+\left(2^{b-1}-1\right)+1=2^b-1$$

By induction, the number of black nodes in a red-black tree of height h is at least $2^b-1=2^{\lceil (h+1)/2\rceil}-1$.

■

Corollary 3.2.2 A red-black tree with n nodes has height $h \leq 2\log_2(n+1)-1$.

It follows immediately from this corollary that the SEARCH operation will run in $O(\lg n)$ time on a red-black tree.

3.3 Insertion and Deletion in Red-Black Tree

3.3.1 Rotation Operations

It is obvious that INSERT and DELETE will also run in $O(\lg n)$ time, but the resulting tree may not satisfy the red-black tree properties, meaning that the tree after insertion and deletion of nodes may not be a red-black tree. We can fix this using a technique known as rotation.

3.3.2 Insertion

INSERT(T,z)

We want to first deal with the cases where simple recoloring can fix the problem. We need to determine what color should the newly inserted node be.

If the tree is empty, z should be black.

If the tree is not empty and the newly inserted node has a black parent, we can make the new node red. This will fix the violation.

If the tree is not empty and the newly inserted node has a red parent, we cannot easily fix the violation by recoloring. If we let that newly inserted node be black, it may lead to a violation of property 3, and if we let the new node be red, it will violate property 2 because it has a red parent.

Suppose that the parent p of the new node is red. Then, p will satisfy the following properties.

- p has a black parent g because of property 2;
- the parent has no other child other than the newly inserted node. This is because because of property 2, p's children must be black, but then by property 3, the black child will violate property 3.
- if g has another child, it would be red

z Is a Leaf

Case 0: g has only 1 child. In this case, we perform a right rotation around p and recolor.

Case 1: g has 2 children, recolor according to Figure 3.2. But this might create a new violation at g, so we need to fix that. If g is the root, we can simply change it to black. Otherwise, we need to continue the fix-up procedure at g. By doing so, we move the violation up the tree.

z Is an Internal Node

Now suppose that z is an internal node.

Case 1: z's two children are black, and P has another black child. G is red. In this case, recolor according to Figure 3.2.

Case 2: G is black. Recoloring won't work. We need to do rotation.

z is a left child of a right child or right child of a left child (i.e. a zigzag path from $G \to P \to z$). In this case, do a left rotation at P according to Figure 3.3. After this operation, the tree falls into the next case.

z is a left child of a left child or right child of a right child (i.e. a straight path from $G \to P \to z$). In this case, do a right rotation at G according to Figure 3.4.

Case 1

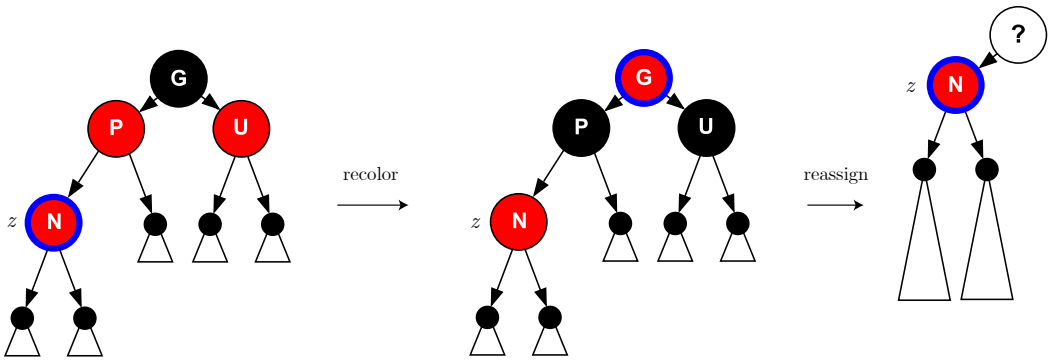


Figure 3.2: <caption>

Case 2

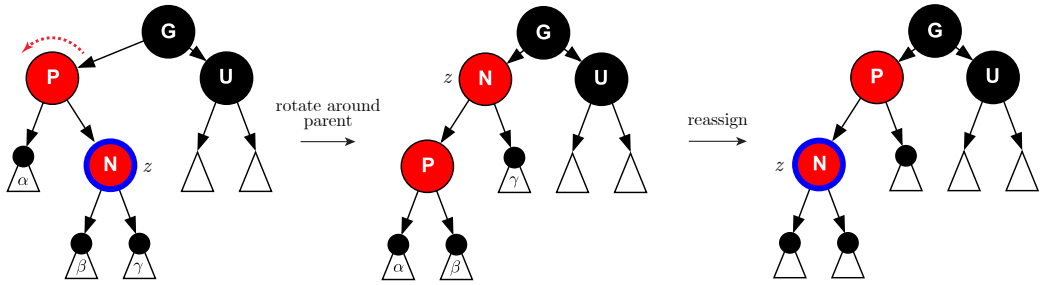


Figure 3.3: <caption>

Case 3

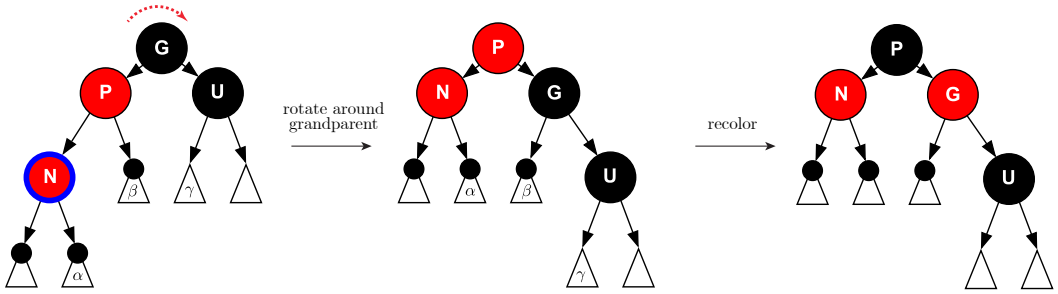


Figure 3.4: <caption>

3.3.3 Deletion

Use the BST implementation of DELETE(T,x).

When x has two children, we replace x by its successor, which gets the same color as x. But then we also need to delete the successor first.

When x has only one child, delete x, promote w and make it black.

When x is a red leaf, we can just delete it without recoloring.

When x is a black leaf, deleting it will cause a violation of property 3, so we need to fix the violation. Let P be the parent of x, and let W be the sibling of x.

Case 1: p is red.

Case 2: p is black

Case 2(a): w has exactly one child Case 2(b): w is black and has two children Case 2(c): w is red and has two children Case 2(d): w has no children

In Case 2(d), w has to be black. Once x is deleted, we have only P and W, which will gives us a black-height of 1 on both sides. This is not fixable, so we make p double-black. And then our goal is to remove the double-black.

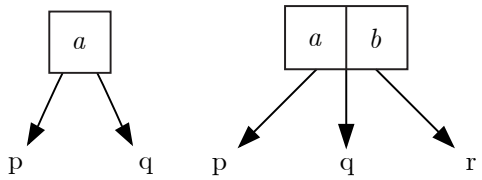
3.4 AVL Tree

3.5 B-Tree

3.5.1 2-3 Tree

Definition 3.5.1 A **search** tree is a 2-3 tree if every internal node has either two children with one element, or three children with two elements. Leaf nodes of a 2-3 tree have no children and one or two elements. More formally, a tree T is a 2-3 tree if and only if one of the following is true:

- T is empty
- T has one element a and two children p, q (p being the left child and q being the right child). p and q are 2-3 trees of the same height, and a is greater than every element in p, and a is less than every element in q
- T has two elements $a < b$ and three children p, q, r (left, middle, right children, respectively). p and q are 2-3 trees of the same height; and a is greater than every element in p and less than every element in q; and b is greater than every element in q and less than every element in r.



Lecture 4 Augmenting Data Structures

4.1 Augmenting Data Structures

For many problems, it is not enough to use only the elementary data structures such as linked list, hash table, or binary tree. But for most of those problems, we don't need to reinvent the wheel. Instead, we can augment the data structures we already have along with some additional information.

An example of augmenting data structure is a dictionary ADT with a size variable. This has the benefit of allowing for computing the size of the dictionary in $O(1)$ time.

4.2 Order Statistics With Red-Black Trees

Dictionary implemented by a red-black tree with a MINIMUM(S) operation that returns the pointer to the element with smallest key in S.

Normally, this MINIMUM operation would take $O(\lg n)$ by following the left-most path. But we can make it better by augmenting the red-black tree:

- Maintain a pointer MINPTR to the minimum element.
  When performing INSERT, compare key of newly inserted element and if it is less than the key of the element pointed to by MINPTR, update MINPTR.
  For DELETE, if the element to be deleted is not the minimum element, then do nothing. If the current minimum is deleted, we need to recompute MINPTR using $O(\lg n)$ time.
- Add a variable MINVAL storing the node with minimum value.
- Maintain a doubly linked list of the elements in the list ordered by their key.
  If you insert a node v as a left child of a node p, then v is the predecessor of p. If v is inserted as the right child, then v is the successor of p.
- At each node, store the minimum element in the subtree rooted at that node.
  After insertion and deletion, update the minimum field of the ancestor of the inserted or deleted node. Also updaate the minimum field during red-black tree rotation.

4.2.1 The RANK Operation

The function RANK(S,k) returns the number of elements in S with key $\leq k$. Let's take a look at some elementary data structures that enables the RANK operation.

- Unordered array: linear search, $\Theta(n)$
- Ordered array: binary search, $\Theta(\lg n)$
- Red-black tree: compare with every element in T, or perform an in-order traversal until reached a key $> k$; count the number of nodes visited while doing the traversal. In both cases, $\Theta(n)$.

Without augmentation, the best we can achieve is doing binary search on a sorted array, with $\Theta(n)$ time for the RANK operation.

We can augment a red-black tree in the following way.

With each node, store the number of nodes in its subtree. By doing so, $v.size$ is equal to the number of nodes in the subtree rooted at v.

For RANK(S,k), search for k. Whenever you go right, add the one plus the number of nodes in the left subtree.

If multiple elements in S has the same key, find the last node (in an inorder traversal) with value $\leq k$.

4.2.2 Maintaining the Size Property at Each Node

The problem then arises with how to maintain the size field under insertion and deletion.

When inserting, add 1 to the size field of proper ancestors of newly inserted node. Similarly, when deleting, subtract 1 from the size field of all proper ancestors of the node being deleted v (v is the physically deleted node with ≤ 1 children).

We also need to maintain the size field when doing rotations. Size of a node can be recomputed from the sizes of its children.

$$v.size = v.left.size + v.right.size + 1$$

2. As a small improvement from the previous implementation, we can store $v.left.size$ instead of $v.size$.

3. Store the rank of each node within the node. But then, INSERT and DELETE are now $\Theta(n)$ in the worst case because we need to go back to recalculate the rank.

4.2.3 The SELECT Operation

SELECT(S,i) returns the element of rank i in the set S.

Suppose S is represented by a RB-tree T. If T is not augmented, we need to do an in-order traversal until i nodes have been visited, which costs $\Theta(n)$ time.

If $T.rootsizeleft \geq i$, then search in the left subtree. If $T.rootsizeleft = i-1$, then return the root. If $T.rootsizeleft < i$, then search in the right subtree for element of rank $i - T.rootsizeleft - 1$.

```
SELECT(v, i)
1  r = v.sizeleft + 1
2  if r > i
3        return SELECT(v.left, i)
4  elseif r == i
5        return r
6  return SELECT(v.right, i − r)
```

4.3 Steps To Create Augmented Data Structures

To create augmented data strucutres, we typically follow these four steps:

1. choose an underlying data structure
2. determine additional information to be maintained
3. verify that the additional information can be maintained by update operations (or basic steps of update operations, e.g. rotations)
4. develop new operations

Theorem 4.3.1 — Augmenting Red-Black Tree. Let f be a field augmenting each node of a red-black tree and suppose that $x.f$ can be computed using information in node x, $x.left$, and $x.right$, possibly including $x.left.f$ and $x.right.f$. Then, the f field can be maintained in all nodes during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance of these operations.

Proof Idea. A change to $x.f$ only propagates to $y.f$ for the ancestors y of x. Since the height of a red-black tree is $O(\lg n)$, at most $O(\lg n)$ nodes have their f fields changed an each change takes $O(1)$ time.

4.4 Intervals ADT

- Objects: a set of closed intervals $[t, t']$ where $t \leq t'$, or equivalently, the set $\{x \in \mathbb{R} \mid t \leq x \leq t'\}$.
- Operations: INTERVAL-INSERT(S, $[t,t']$), INTERVAL-DELETE(S, $[t,t']$), INTERVAL-SEARCH(S, $[t,t']$) that returns a pointer to the interval in S that overlaps with $[t,t']$ (non-empty intersection).

Naive implementations:

- Unsorted linked list: $\Theta(n)$
- Sorted linked list: $\Theta(n)$
- Sorted array: $\Theta(n)$
- Red-black tree: store $i.low$ as the key, $O(\lg n)$ for insertion and deletion. For search, skip intervals $i \leq T$ with $t' \leq i.low$.

Augment each node x with

$$\text{MAX-HIGH}(x) = \max\{y.high \mid \ y \text{ is an interval stored in the subtree rooted at } x\ \}$$

This field can be calculated using

$$x.\textit{max-high} = \max\{x.\textit{high}, x.\textit{right}.\textit{max-high}, x.\textit{right}.\textit{max-high}\}$$

INTERVAL-SEARCH$(T, [t,t'])$

```
1  x = T.root
2  while x ≠ NIL and [t,t'] does not intersect [x.low,x.high]
3      if x.left ≠ NIL and x.left.max-high ≥ t
4          x = x.left
5      else
6          x = x.right
7  return x
```

The time complexity of INTERVAL-SEARCH is $O(\lg n)$.

Theorem 4.4.1 Any execution of INTERVAL-SEARCH$(T, [t,t'])$ either returns a pointer to a node whose interval intersects $[t,t']$, or returns NIL if no such interval exists.

Lemma 4.4.2 Loop invariant: if T contains a node whose interval intersects $[t,t']$, then there is such an interval in the subtree rooted at x.

Lecture 5 Priority Queue and Heap

5.1 Priority Queue

5.1.1 Priority Queue ADT

The priority queue ADT is a data type that stores a collection of items with priorities (keys) that supports the following opeations:

- INSERT(Q,x) inserts the element x into the priority queue Q.
- MAXIMUM(Q) returns the element of Q with the largest key.
- EXTRACT-MAX(Q) removes and returns the element of q with the largest key.
- INCREASE-KEY(S,x,k) increases the value of element x's key into the new value k, assuming that $k \geq x$.

Priority queue allows us to access the element with largest (if max-priority queue) or smallest (if min-priority queue) more efficiently. It has many applications in computer science, such as: job scheduling in operation systems, bandwith management, or finding minimum spanning tree of a graph, etc.

5.1.2 Primitive Implementation Using Linked Lists

We can have a naive implementation of a priority queue simply using a sorted linked list, which has the following time complexity:

- INSERT(Q,x): $\Theta(n)$ in the worst case. We have to linearly search the correct location of insertion.
- MAXIMUM(Q): $\Theta(1)$ by returning the head of the list.
- EXTRACT-MAX(Q): $\Theta(1)$ by removing and returning the head of the list.
- INCREASE-KEY(S,x,k): $\Theta(n)$ in the worst case. Need to move element to new location after increase.

However, we want to have something that is more efficient than $\Theta(n)$. As it turns out, by putting the elements in a specific way, we can achieve worst-case time complexity of $\Theta(\log n)$ for INSERT, EXTRACT-MAX, and INCREASE-KEY. Even better, we can show that the amortized complexity of INSERT is $\Theta(1)$ and EXTRACT-MAX is $\Theta(\log n)$.

5.2 Heap

5.2.1 Types of Binary Trees

Before starting to formally define heaps, let's review some definitions about binary trees.

Definition 5.2.1 — Full Binary Tree. A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.

Definition 5.2.2 — Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Importantly, a complete binary tree with n nodes has $\lfloor n/2 \rfloor$ internal nodes (nodes that are not leaves).

Definition 5.2.3 — Perfect Binary Tree. A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.

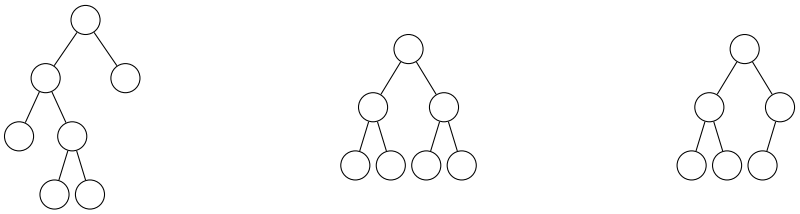


Figure 5.1: From left to right: full binary tree, complete binary tree, perfect binary tree.

Conveniently, a complete binary tree can be represented as an array.

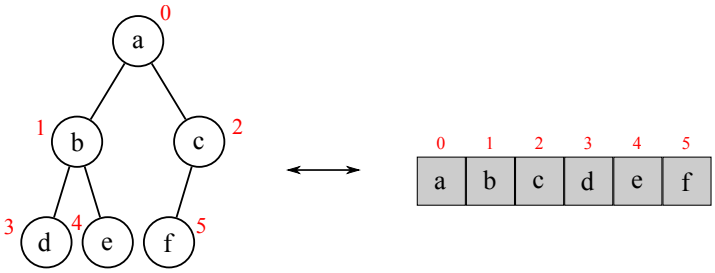


Figure 5.2: A complete binary tree and its corresponding array representation

Assuming that we index from 0, we can compute the indices of each node's parent, left, and right child.

PARENT(i) = $\lfloor (i-1)/2 \rfloor$
LEFT(i) = $(2i)+1$
RIGHT(i) = $(2i)+2$

If we index from 1, the indices are calculated as follows:

PARENT(i) = $\lfloor i/2 \rfloor$

LEFT(i) = $2i$
RIGHT(i) = $2i+1$

5.2.2 Heap Property

Then, we can define a max-heap as a complete binary tree with the max-heap property.

> **Definition 5.2.4 — Max-heap Property.** In a max-heap represented by the array A , the max-heap property is that for every node i other than the root,
>
> $$A[\text{PARENT}(i)] \geq A[i].$$
>
> that is, the value of a node is at most the value of its parent.

The max-heap property guarantees that the largest element in a heap is always stored at its root.

For our heap implementation, we will include the following operations: INSERT, MAXIMUM, EXTRACT-MAX, INCREASE-KEY, MAX-HEAPIFY, BUILD-MAX-HEAP. The first few operations allow us to use heap to implement the priority queue ADT, and in addition to those, BUILD-MAX-HEAP allows us to produce a max-heap from an unordered array.

5.3 Maintaining the Heap Property

Given an array A and index i, MAX-HEAPIFY will correct a single violation of the max-heap property in the subtree with i as its root. To implement MAX-HEAPIFY, we use a technique called "trickle down".

First, assume that the trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps. If element $A[i]$ violates the max-heap property, we correct thie violation by "trickling" element $A[i]$ down the tree until it reaches the correct position. By doing so, we can make the subtree rooted at index i a max-heap. In every trickle-down step, swap $A[i]$ with its largest child.

MAX-HEAPIFY(A, i)
1 $l = \text{LEFT}(i)$
2 $r = \text{RIGHT}(i)$
3 **if** $l \leq A.heapsize$ and $A[l] > A[i]$
4 $\quad largest = l$
5 **else** $largest = i$
6 **if** $r \leq A.heapsize$ and $A[r] > A[largest]$
7 $\quad largest = r$
8 **if** $largest \neq i$
9 $\quad$ exchange $A[i]$ with $A[largest]$
10 $\quad$ MAX-HEAPIFY$(A, largest)$

5.3.1 Correctness of MAX-HEAPIFY

5.3.2 Running Time of MAX-HEAPIFY

5.4 Inserting Into Max-Heap

To insert into a max-heap while maintaining the heap property, we use a similar technique. We first append the new element to the end of the heap. The new element will become the right-most leaf at the last level. Then, we check if the element is already at the right position. If not, we "bubble" the element up the tree, until it reaches the correct position.

5.5 Build Heap From Unsorted Array

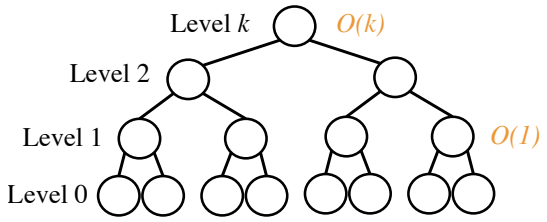
BUILD-MAX-HEAP(A)
1 $A.heapsize = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

The reason we start calling MAX-HEAPIFY at $\lfloor A.length/2 \rfloor$ is because elements beyond that $A[n/2+1, \cdots, n]$ are all leaves of the tree. Recall that for a complete binary tree with n nodes, there are only $\lfloor n/2 \rfloor$ internal nodes.

5.5.1 Running Time of BUILD-MAX-HEAP

Each call of MAX-HEAPIFY takes $O(\log n)$ time, and BUILD-MAX-HEAP calls MAX-HEAPIFY $O(n)$ times. Thus, the running time of BUILD-MAX-HEAP is $O(n \log n)$. However, this upper bound is not tight. We will prove a tighter upper bound of $O(n)$.

Note that MAX-HEAPIFY takes $O(1)$ time for nodes tthat are one level above the leaves, and in general, it takes $O(k)$ for nodes that are k levels above the leaves. We have $n/4$ nodes at level 1, $n/8$ at level 2, etc. At the root level, which is $\log_2 n$ levels above the leaves, we have only 1 node.



More generally, a heap with n nodes has a height of $\lfloor \log_2 n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes at any height h. Hence, the total cost of BUILD-MAX-HEAP can be written as

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\log_2 n} \frac{h}{2^h} \right).$$

The last summation is bounded by a constant, namely

$$\sum_{h=0}^{\log_2 n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = 2 = O(1).$$

Thus,

$$O\left(n \sum_{h=0}^{\log_2 n} \frac{h}{2^h}\right) = O(n)O(1) = O(n).$$

5.6 Heapsort

HEAPSORT(A)

```
1  for i = A.length downto 2
2      exchange A[1] with A[i]
3      A.heapsize = A.heapsize − 1
4      MAX-HEAPIFY(A, 1)
```

Lecture 6 Mergeable Heaps

II Algorithm Analysis

Lecture 7 Average Case Complexity and Randomized Algorithms

7.1 Basic Probability Theory

7.1.1 Sample Space and Events

Definition 7.1.1 — Probability Space and Sample Space. A ***probability space*** $(\Omega, \Pr)$ consists of a finite or countable set Ω called the sample space, and the probability function $\Pr : \Omega \to \mathbb{R}$ such that for all $\omega \in \Omega$, $\Pr(\omega) \geq 0$ and $\sum_{\omega \in \Omega} \Pr(\omega) = 1$. We cann an element $\omega \in \Omega$ a sample point, or ***outcome***, or ***simple event***.

A sample space models some random experiment, where Ω contains all possible outcomes, and $\Pr(\omega)$ is the probability of the outcome ω. We always talk about probabilities in relation to a sample space.

Definition 7.1.2 — Event. An event A is a set of outcomes, $A \subseteq \Omega$. We define the probability of an event A to be the sum of the probabilities of its elements

$$\Pr(A) = \sum_{\omega \in A} \Pr(\omega)$$

If we have $\Pr(\omega) = \Pr(\omega')$ for all distinct $\omega, \omega' \in \Omega$, we say that the probability is uniform over Ω.

7.1.2 Properties of Probability Functions

Definition 7.1.3 — Complement. The complement of an event A is

$$\overline{A} = \Omega \setminus A$$

The complement of A is also denoted by not A.

Theorem 7.1.1

$$\Pr(\overline{A}) = 1 - \Pr(A)$$
$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$$
$$\Pr(A \cup B) \leq \Pr(A) + \Pr(B)$$

It can then be proved by induction that

$$\Pr(A_1 \cup A_2 \cup \cdots \cup A_k) \leq \Pr(A_1) + \Pr(A_2) + \cdots + \Pr(A_k)$$

for any events $A_1, \cdots A_k$.

We say that two events A and B are disjoint or mutually exclusive if $A \cap B = \emptyset$. If A and B are disjoint, $\Pr(A \cup B) = \Pr(A) + \Pr(B)$.

7.2 Conditional Probability and Independence

Conditional probability allows us to compute the probability of some event given that we already know that some other events have occurred.

Definition 7.2.1 — Conditional Probability. The probability of an event A conditional on an event B is

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)}$$

given that $\Pr(B) > 0$.

7.3 Average Case Analysis

Let A be an algorithm, and let

$$t(x) = \text{number of steps taken by } A \text{ on input } x.$$

We know that the worst case time complexity of A is

$$T(n) = \max\{t(x) \mid x \text{ has size } n\}$$

We can define the average case time complexity of A as

Definition 7.3.1 The average case time complexity of A is

$$T'(n) = \mathbb{E}[t(x) \mid x \text{ has size } n]$$

If all inputs of size n are equally likely, then

$$T'(n) = \frac{\sum\{t(x) \mid size(x) = n\}}{|x \mid size(x) = n|}$$

In general, the average case time complexity is less than or equal to the worst case time complexity, that is $T'(n) \leq T(n)$ for all n.

The average case time complexity is dependent on the probability distribution of the inputs, which in turn can depends on the application where the algorithm is to be used. However, this is usually unknown, in which case, assuming uniform distribution usually makes the analysis easier. For some applications, an algorithm with good average case behavior but bad worst case behavior is sufficient.

We follow the following steps before we can analyze the average case time complexity of an algorithm:

- define the sample space of the inputs
- define the probability distribution function
- define any necessary random variables

7.4 Average Case Analysis of Linear Search

Consider the following algorithm for linear search in an unsorted array.

```
LINEAR-SEARCH(L, k)
1  j = 1
2  while j ≤ n
3      if L[k] == k
4          return j
5      j = j + 1
6  return 0
```

It is obvious that the worst case complexity is $O(n)$ because there will be at most n comparisons with the searching key k. Following the procedure introduced above, we can perform an average case complexity analysis.

1. **Define sample space**: This will be the set of possible inputs that we are considering in our analysis. We define our sample space to be all pairs (L,k) where L is an array of size n and k is a key. Some observations on our choice of sample space:
   - If we fix $L = [l_1, \cdots, l_n]$, then the sample space for k is $[\text{NIL}, l_1, \cdots, l_n]$ where NIL is used to indicate a value that is not in L. Hence, for a fixed L of size n, the sample space for k contains $n+1$ sample points.
   - The number of comparisons performed by LINEAR-SEARCH is the same for (L,k) and (L',k') if k occurs in L at the same position as k' in L'. Duplicates don't matter because the algorithm returns as soon as it finds the first occurrence.
   - If k and k' are not in L, then LINEAR-SEARCH(L,k) will also perform the same number of comparisons.
   - The relative order between elements of L does not matter since LINEAR-SEARCH only performs equality tests, so choosing $L = [1, \cdots, n]$ is just as good as $L = [n, \cdots, 1]$.

   Hence, we can formally define our sample space as

   $$S_n = \{(L,i) \mid i \in \{\text{NIL}, 1, \cdots, n\}\}$$

   where $L = [1, \cdots, n]$.

2. **Probability distribution**: We choose uniform distribution, which means that every point in the sample space has the same probability $\frac{1}{n+1}$.

$$\Pr(k=j)=\frac{1}{n}$$

Alternatively, we can say that NIL has probability of $1/2$ and everything else has probability $1/2n$.

$$\Pr(k=j)=\begin{cases}\frac{1}{2} & \text{if } j=\text{NIL}\\ \frac{1}{2n} & \text{otherwise}\end{cases}$$

3. **Define random variable**: Let $t_n: S_n \to \mathbb{N}$ be such that $t_n(i)$ is the number of comparisons on elements L performed when the input is (L,i) where

$$t_n(i)=\begin{cases}i & \text{for } i=1,\cdots,n\\ n & \text{for } i=\text{NIL}\end{cases}$$

4. **Analysis**:
For the uniform distribution,

$$\begin{aligned}T'(n)=\mathbb{E}[t_n]&=\frac{\sum\{t_n(L,i)\mid i=\text{NIL},\cdots,n\}}{n+1}\\ &=\frac{n+\sum_{i=1}^{n} i}{n+1}\\ &=\frac{n+\frac{n(n+1)}{2}}{n+1}\\ &=\frac{n}{2}+\frac{n}{n+1}\\ &<\frac{n}{2}+1\end{aligned}$$

Hence, $T'(n)\in O(\frac{n}{2})$. This says the average case is twice better than the worst case.

7.5 Average Case Analysis of Quick Sort

Quicksort is ued to sort a multi-set of elements S from a totally ordered domain. The pseudocode for quicksort is shown below.

```
QUICKSORT(S)
1  if |S| ≤ 1
2        return S
3  pivot = select a pivot
   // partition S into L, E, and G such that L contains all elements less than pivot
   // E contains all elements equal to pivot, and G contains all elements greater than pivot
4  L,E,G = PARTITION(S, pivot)
5  return QUICKSORT(L) + E + QUICKSORT(G)
```

In practice, we often choose the first element of S as the pivot. The worst case time complexity of quicksort is $O(n^2)$ because it has to partition the input S into three parts, L, E, and G, which gives us this recurrence.

$$T(n) = T(n-1) + T(0) + \Omega(n) \in O(n^2)$$

Following the procedure introduced above, we can perform an average case complexity analysis. In QUICKSORT, or most comparison-based sorting algorithms, only the relative order of the elements matter, not their actual values. So, it's reasonable to assume that S_n is the set of all permutations of S_n.

1. Define sample space: S_n = all permutations of $\{1, \cdots, n\}$
2. Probability distribution: We know that $|S_n| = n!$. Assuming uniform distribution, we have

$$\Pr[\pi] = \frac{1}{n!} \qquad \text{for all } \pi \in S_n$$

3. Random variables: Let $t_n : S_n \to \mathbb{N}$ be the random variable such that

$$t_n(i) = \text{number of element comparison performed by QUICKSORT}(S)$$

4. Analysis: We first consider the complexity measure and the worst-case analysis. We choose the number of comparisons as the complexity measure. The comparisons occur when PARTITION is called. In the worst case, the number of comparisons performed by QUICKSORT is $\leq \binom{n}{2} \in O(n^2)$. The matching lower bound on the worst-case compexity is achieved when QUICKSORT is performed on a sorted array.
   Let $T'(n) = \mathbb{E}[t_n]$. Let X_{ij} $S_n \to \{0,1\}$ be an indicator random variable such that $X_{ij}(\pi) = 1$ if and only if elements i and j are compared during QUICKSORT(π).
   No pair of elements is compared more than once, so

$$t_n(\pi) = \sum_{1 \leq i < j \leq n} X_{ij}(\pi),$$

   and

$$\begin{aligned} T'(n) = \mathbb{E}[t_n] &= \sum_{1 \leq i < j \leq n} \mathbb{E}[X_{ij}] && \text{by linearity of expectation} \\ &= \sum_{1 \leq i < j \leq n} \Pr[X_{ij} = 1] && \text{since } X_{ij} \text{ is an indicator variable} \end{aligned}$$

   In QUICKSORT(π), as long as no element in $\{i, \cdots, j\}$ is chosen as a pivot, these elements will stay together, either all going to L, or all going to G in recursive calls.
   Eventually, one of the elements in $\{i, \cdots, j\}$ is chosen as pivot. Suppose that p is the first of the elements in $\{i, \cdots, j\}$ to be chosen as pivot. If $i < p < j$, then i and j are not compared during QUICKSORT. If $p = i$ or $p = j$, then i and j are compared, making $X_{ij} = 1$.
   There are $j - i + 1$ possibilities for p to be chosen as pivot. Among those possible choices, $X_{ij} = 1$ for two possibilities, and $X_{ij} = 0$ for the rest. Since all permutations of the inputs are equally likely, each of these possibilities for p is equally likely and has probability of $\frac{1}{j-i+1}$.
   Hence,

$$\Pr[X_{ij} = 1] = \frac{2}{j-i+1}$$

Using this, we can rewrite $T'(n)$ as with $k = j - i + 1$

$$\begin{aligned}
T'(n) &= \mathbb{E}[t_n] \\
&= \sum_{1 \le i < j \le n} \Pr[X_{ij} = 1] \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n} \sum_{k=1}^{n-1} \frac{2}{k+1} && \text{substituting } k \\
&< \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} \\
&= n \sum_{k=1}^{n} \frac{2}{k} \\
&\in O(n \lg n) && \text{harmonic series}
\end{aligned}$$

Therefore, the average case time complexity of quicksort is $O(n \lg n)$.

From this analysis, we notice that the average case time complexity for quicksort is much smaller than the worst case time complexity. If the actual distribution of inputs is close to uniform distribution, then the average case time complexity will serve as a more realistic estimate of running time. But problem arises when the actual distribution is not uniform. To solve this issue, we can modify the quicksort algorithm to use a randomized pivot selection.

7.6 Randomized Quicksort

In randomized quicksort, we randomly pick the pivot instead of using the first element of S. Pick each element of S to be the pivot with probability of $\frac{1}{|S|}$ using a random number generator.

```
RANDOMIZED-QUICKSORT(S)
1  if |S| ≤ 1
2      return S
3  r = RANDOM(1, |S|)
4  pivot = S[r]
   // partition S into L, E, and G such that L contains all elements less than pivot
   // E contains all elements equal to pivot, and G contains all elements greater than pivot
5  L, E, G = PARTITION(S, pivot)
6  return RANDOMIZED-QUICKSORT(L) + E + RANDOMIZED-QUICKSORT(G)
```

RANDOM(i, j) will return a number in $\{i, i+1, \cdots, j-1, j\}$ each equally likely, assuming $i \le j$.

The behavior of a randomized algorithm A may depend on its input I and the sequence of choices made by the algorithm.

Let $t(I,p)$ denote the running time of algorithm A on input I with the sequence of random choices p. Then, the expected running time of algorithm A on input I is

$$\mathop{\mathbb{E}}_{p}[t(T,p)] = \sum_{e} \Pr[p]t(I,p)$$

The worst case expected time complexity of algorithm A is

$$T''(n) = \max_{I \in S_n} \mathop{\mathbb{E}}_{p}[t(I,p)]$$

where S_n is the set of all inputs of size n.

Note that the worst case complexity does not depend on any assumption about the input distribution.

R Be careful that the worst-case expected time complexity is the max of expectation (rather than the other way around).

Since all elements in S are equally likely to be chosen as the pivot in each call to RANDOMIZED-QUICKSORT, each of the $j-i+1$ possibilities for p is equally likely. The same average-case analysis for the non-randomized quicksort applies to the worst-case expected analysis of RANDOMIZED-QUICKSORT.

Therefore, the worst-case expected time complexity of RANDOMIZED-QUICKSORT is $O(n \log n)$.

7.7 Randomized Selection

This section is covered in CLRS 9.2. Given a multi-set S and an integer k such that $1 \leq k \leq |S|$, we want to find the kth smallest element in S.

If $S = \{a_1 \leq a_2 \leq \cdots \leq a_n\}$, then return a_k. A naive implementation involves sorting S using RANDOMIZED-QUICKSORT and returning the kth element of the sorted list.

```
RANDOMIZED-SELECT(S, k)
 1  if |S| == 1
 2      return S
 3  r = RANDOM(1, |S|)
 4  pivot = S[r]
    // partition S into L, E, and G such that L contains all elements less than pivot
    // E contains all elements equal to pivot, and G contains all elements greater than pivot
 5  L, E, G = PARTITION(S, pivot)
 6  if |L| ≥ k
 7      return RANDOMIZED-SELECT(L, k)
 8  if |L| + |E| ≥ k
 9      return pivot
10  return RANDOMIZED-SELECT(G, k − |L| − |E|)
```

In RANDOMIZED-SELECT we perform one fewer recursive call than RANDOMIZED-QUICKSORT, so we know that the worst-case time complexity is less than or equal to the compexity of RANDOMIZED-QUICKSORT.

Let S_n be the set of all permutations of $\{1, \cdots, n\}$, which is our sample space. There is no probability distribution because we are already considering a randomized algorithm. Similar to randomized quicksort, we consider the worst-case expected time complexity $T''(n)$.

$$T''(n) \leq \frac{1}{n} \sum_{i=1}^{n} \max\{T''(i-1), T''(n-i)\} + n - 1$$

Lecture 8 Hashing

8.1 Hashing and Hash Function

Let U be the universe of possible keys, and let m be the size of the hash table. Then, we say that

$$h: U \to \{0, \cdots, m-1\}$$

is a hash function.

If two keys are mapped to the same location/bucket/slot by the hash function, we say that they collide.

Furthermore, if $|U| > m$, then by the pigeonhole principle, there are at least two keys that collide. And because in virtually all cases, $|U| > m$, collision is unavoidable. A well-chosen hash function will minimize the number of collisions, but we still need some means to resolve collisions.

R A fun fact about the etymology of the word "hash": it is said that the word "hash" originated from the french word "hache", which refers to the action of chopping something into pieces. Hashing, as we will see, involves the same notion of randomly chopping and mixing.

8.2 Resolving Collision

Chaining put all elements in $S \subseteq U$ that hash to the same slot in a linked list.

We define the load factor α to be $\alpha = n/m$ where $n = |S|$ and m is the number of buckets (slots) in the hash table. α is the average number of elements of S stored in a slot of the hash table.

Let n_i be the number of elements of S in slot i. Then,

$$\sum_{i=0}^{m-1} n_i = n$$

In the worst case, hashing with chaining takes $\Theta(\max_{0 \le i \le m-1} n_i)$ in addition to the time to compute $h(x)$ (the hashing step).

From the analysis above, we know that $\max\{n_i\} = n$ if all elements of S map to the same bucket.

If $|U| > m(n-1)$, then by the pigeonhole principle, there exists $S \subseteq U$ with $|S| = n$ such that all element of S hash to the same bucket.

Overall, hashing with chaining without randomization has the following time complexity:

- INSERT(x): $O(1)$ assuming all linked lists are unsorted and $x \notin S$
- DELETE(p): $O(1)$ if list is doubly linked; if singly linked, then $O(n_i)$ where n_i is the size of the bucket where p is in.

8.3 Universal Hashing

We can achieve better time complexity by randomly choosing the hash function. There is a family of hash functions that we call the universal hash functions, which we will define as follows.

Definition 8.3.1 — Universal Hashing. Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U into the range $\{0, \cdots, m-1\}$. Such a collection is said to be universal if for each pair of distinct keys, $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys is not more than the chance $1/m$ of a collision if $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, \cdots, m-1\}$.

That is, a finite set $\mathcal{H}$ of hash functions from $U \to [0, \cdots, m-1]$ is universal if for all $x \neq y \in U$

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$$

or equivalently,

$$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{m}$$

Here are some examples of universal hashing families.

1. $\mathcal{H}$ is the set of all functions from $U \to [0, \cdots, m-1]$ assuming U is finite. Let $u = |U|$. Then, $|\mathcal{H}| = m^u$. For any distinct $x, y \in U$

   $$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| = m^{u-1} = \frac{|\mathcal{H}|}{m}$$

   since there are m choices for which bucket each element of $U - \{y\}$ gets mapped to. So,

   $$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}$$

   Therefore, $\mathcal{H}$ is universal.
2. Let p be prime and $U = \{0, \cdots, p-1\} = \mathbb{Z}_p$. And let

   $$h_{a,b}(x) = [(ax + b) \bmod p] \bmod m$$

   and

   $$\mathcal{H}_{p,m} = \{h_{a,b} : U \to [0, \cdots, m-1] \mid a, b \in U, a \neq 0\}$$

So $|\mathcal{H}_{p,m}| = p(p-1)$. We can prove that $\mathcal{H}_{p,m}$ is universal.

3. Let $U = \{0, \cdots, 2^k - 1\}$ and let

$$h_a(x) = \left\lfloor \frac{ax \bmod 2^k}{2^{k-m'}} \right\rfloor$$

This selects the $(k-m'+1)$th through kth least significant bits. In other words, we throw away everything in ax except the k least significant bits, and then remove $k-m$ least significant bits. The size of the hash table is $m = 2^{m'}$.

Let

$$\mathcal{H}' = \{h_a \mid 0 < a < 2^k, a \text{ is odd}\}$$

We have $|\mathcal{H}'| = u/2$. $\mathcal{H}'$ is universal. The proof is more complicated.

For each of the universal hash families, let's also take a look at their sizes

1. $|\mathcal{H}| = m^u$. To specify a hash function in $\mathcal{H}$, we need $u \log_2 m$ bits. This is too big.
2. $|\mathcal{H}_{p,m}| < u^2$. To specify a hash function $\mathcal{H}_{p,m}$, we need less than $2\log_2 u$ bits.
3. $|\mathcal{H}'| = u/2$. To specify a hash function in $\mathcal{H}'$, wee need $(\log_2 u) - 1$ bits.

8.4 Analysis of Hashing with Chaining

Fix $S \subseteq U$ where $|S| = n$. Pick $h \in \mathcal{H}$ randomly where $\mathcal{H}$ is a universal family of hash functions from $U \to \{0, \cdots, m-1\}$.

Let $x \in U$. Let $C_x : \mathcal{H} \to \mathbb{N}$ be such that

$$C_x(h) = \text{the number of keys in } S \text{ that hash to } h(x)$$

C_x is a random variable that depends on the choice of h.

For each $y \in S$, let $C_{x,y} : \mathcal{H} \to \{0,1\}$ be the indicator random variable that is 1 if and only if $h(x) = h(y)$.

$$C_x(h) = \sum_{y \in S} C_{x,y}(h)$$

Theorem 8.4.1 — Expected Number of Collisions For Each Key in Universal Hashing.

$$\mathop{\mathbb{E}}_{h \in \mathcal{H}}[C_x] \leq 1 + \frac{n}{m} = 1 + \alpha$$

Proof. If $y = x$, then $C_{x,y}(h) = 1$ for all $h \in \mathcal{H}$ so $\mathbb{E}_{h \in \mathcal{H}}[C_x] = 1$.

If $y \neq x$, then

$$\begin{aligned} \mathop{\mathbb{E}}_{h \in \mathcal{H}}[C_{x,y}] &= \Pr_{h \in \mathcal{H}}[C_{x,y}(h) = 1] && \text{since } C_{x,y} \text{ is indicator variable} \\ &= \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq 1/m && \text{since } \mathcal{H} \text{ is universal} \end{aligned}$$

Hence,

$$\begin{aligned}\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C_x] &= \sum_{y\in S} \mathop{\mathbb{E}}_{h\in\mathcal{H}}[C_{x,y}] && \text{by linearity of expectation}\\ &\leq 1 + \sum_{y\in S-\{x\}} \frac{1}{m} \leq 1 + \frac{n}{m} = 1+\alpha\end{aligned}$$

■

Corollary 8.4.2 The worst-case expected search time for x is $O(1+\alpha)$.

Corollary 8.4.3 Starting with an initially empty hash table of size m, the worst-case expected time to handle any sequence of s INSERT, DELETE, SEARCH operations containing $n = O(m)$ INSERT operations is $O(s)$. Hence, we can perform each operation in $O(1)$ time.

8.5 Perfect Hashing

h is perfect for S if for each element of S hashes to a different slot (i.e. no collisions). For example,

$$h(x) = x \bmod 5$$

is perfect for $\{1,14,20\}$, but not for $\{1,9,14\}$.

For a static dictionary (where S does not change with no INSERT and DELETE),then it may be worthwhile to find a perfect hash function for S. For example, it is useful to construct a perfect hash function if we are designing a compiler for a programming language, and our S is the set of reserved keywords in the language.

8.5.1 Constructing a Perfect Hash Functions

Let S be a set of n keys. Let $\mathcal{H}$ be a universal family of hash functions. Let $C : \mathcal{H} \to \mathbb{N}$ be the random variable so that

$$C(h) = \text{the number of collisions when } h \text{ is used to hash } S$$

More formally,

$$C(h) = |C_{x,y} \in S \mid x \neq y, h(x) = h(y)|$$

where $C_{x,y}$ is defined similarly as the indicator variable that is 1 if and only if $h(x) = h(y)$.

Lemma 8.5.1 — Expected Number of Collisions in Universal Hashing.

$$\mathbb{E}[C] \leq \frac{\binom{n}{2}}{m}$$

Proof.

$$C(h) = \sum \{C_{x,y}(h) \mid x < y, x, y \in S\}$$

By linearity of expectation,

$$\begin{aligned}
\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C] &= \sum \{\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C_{x,y}] \mid x, y \in S, x < y\} \\
&= \sum \{\Pr_{h\in\mathcal{H}}[h(x) = h(y)] \mid x, y \in S, x < y\} \\
&\leq \sum \{1/m \mid x, y \in S, x < y\} && \text{since } \mathcal{H} \text{ is universal} \\
&= \frac{\binom{n}{2}}{m} && \text{since there are } \binom{n}{2} \text{ pairs of keys that may collide}
\end{aligned}$$

■

If $m > \binom{n}{2}$, then $\mathbb{E}_{h\in\mathcal{H}}[C] < 1$. Since $C(h) \in \mathbb{N}$ for all $h \in \mathcal{H}$, this implies that there exists some $h \in \mathcal{H}$ such that $C(h) = 0$. This tells if we are willing to use $\Theta(n^2)$ space for the hash table, there exists a hash function $h \in \mathcal{H}$ that is perfect for S, so that SEARCH only takes only one step.

Theorem 8.5.2 If $m > 2\binom{n}{2} = n(n-1)$, then

$$\mathop{\mathbb{E}}_{h\in\mathcal{H}}[C] < \frac{1}{2}$$

This means more than half of the functions in $\mathcal{H}$ is perfect for S.

The randomized algorithm for constructing a perfect hash function, we

CONSTRUCT-PERFECT-HASH$(\mathcal{H}, S)$

```
1  Pick h ∈ H uniformly at random.
2  for s in S
3      if there is a collision
4          start over and try again
5  if no collision
6      return h
```

If $m > 2\binom{n}{2}$, the expected number of tries is less than 2, and the algorithm takes $O(2n)$ time to find a perfect hash function for S.

8.5.2 FKS Hashing

However, sometimes when the size of S is large, the $O(n^2)$ space that our current perfect hashing table is not longer ideal. In this case, we want to use $m \in O(n)$ space for the hash table. Then,

$$\mathbb{E}[C] = \frac{\binom{n}{2}}{m} \in O(n)$$

Some buckets will have size > 1, but all buckets are likely to have size $O(\sqrt{n})$. Essentially, if a bucket have size b, there are at least $\binom{b}{2}$ collisions.

The idea is to use a perfect hash function to represent each bucket. If the size of bucket i is b, we will use b^2 space for it. The resulting data structure is a 2-level hash. For $i = 0, \cdots, m-1$, let $n_i(h) = \{x \in S \mid h(x_i) = i\}$ be the number of keys in S that get mapped to bucket i by h. An example of the resulting hash table is shown in Figure 8.1.

$$\sum_{i=0}^{m-1} n_i(h) = n$$

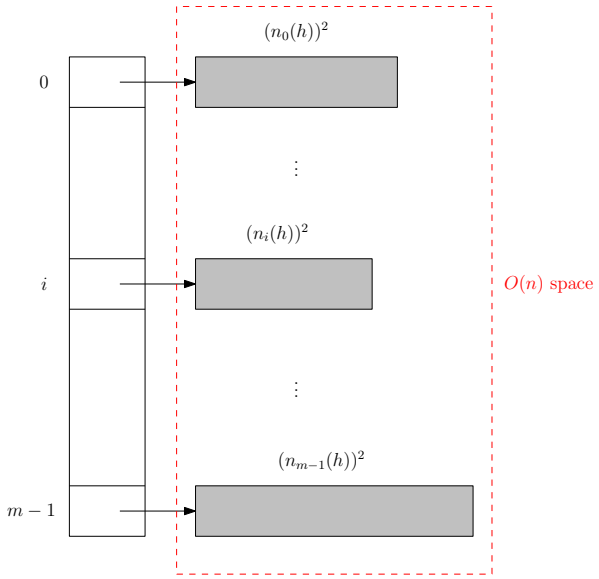


Figure 8.1: FKS 2-level hash table. As shown in the diagram and explained above, the length of each secondary hash table is $(n_i(h))^2$.

The expected amount of space used by all the secondary hash tables can be computed as follows:

$$\begin{aligned} C(h) &= \sum_{i=0}^{m-1} \binom{n_i(h)}{2} \\ &= \sum_{i=0}^{m-1} \frac{n_i(h)^2}{2} - \sum_{i=0}^{m-1} \frac{n_i(h)}{2} \\ &= \frac{1}{2} \sum_{i=0}^{m-1} n_i(h)^2 - \frac{n}{2} \end{aligned}$$

By linearity of expectation,

$$\mathbb{E}\left[\sum_{i=0}^{m-1} n_i(h)^2\right] = 2\,\mathbb{E}[C] + n$$
$$= 2\frac{\binom{n}{2}}{m} + n \in \Theta(n) \quad \text{if } m \in \Theta(n)$$

h itself is not a perfect hash function. For each non-empty bucket, find a hash function $h_i : U \to \{0, \cdots, m_i - 1\}$, where $m_i = (n_i(h))^2$, that is perfect for the seconday hash table at bucket i (i.e. perfect for the set $\{x \in S \mid h(x) = i\}$).

8.5.3 Representing FKS Hash Table

In practice, we can represent the two-level FKS hash table for $S \subseteq U$ as a single array. Suppose that $U = \{0, \cdots, p-1\}$. We use $\mathcal{H}_{p,m}$ for different values of m.

The first 3 locations of the array contain m, a, and b used to specify the first-level hash function h. In particular, m is the size of the first-level hash table; a and b are the parameters for the first-level table.

The next m locations contain pointers to array corresponding to each second-level hash table.

Each second-level table is stored preceeded by the specifications of its hash function: $m_i = n_i(h_{a,b})$ and the parameter $a_i + b_i$ that specifies the perfect hash function h_{a_i,b_i} for the ith secondary hahs table.

■ **Example 8.1** Let $p = 31$. $U = \{0, \cdots, 30\}$, $n = 6$, and $S = \{2, 4, 5, 15, 18, 30\}$, so $n = 6$. Let the size of the first-level hash table be m.

We randomly choose a hash function from $\mathcal{H}_{31,6}$ for the first-level table. Suppose that we end up getting

$$h_{2,0}(x) = [(2x + 0) \bmod 31] \bmod 6$$

as the hash function.

In this case,

- Bucket 0 contains: 15; so $n_0(h) = 1$ and $n_0(h)^2 = 1$,
- Bucket 2 contains: 4; so $n_2(h) = 1$ and $n_2(h)^2 = 1$,
- Bucket 4 contains: 2, 5; so $n_4(h) = 2$ and $n_4(h)^2 = 4$,
- Bucket 5 contains: 18, 30; so $n_5(h) = 2$ and $n_5(h)^2 = 4$, and
- All other buckets are empty with $n(h) = 0$.

The two-level hash table would looks like Figure 8.2

If we were to represent this two-level using a 1-D array, it should look like Figure 8.3 ■

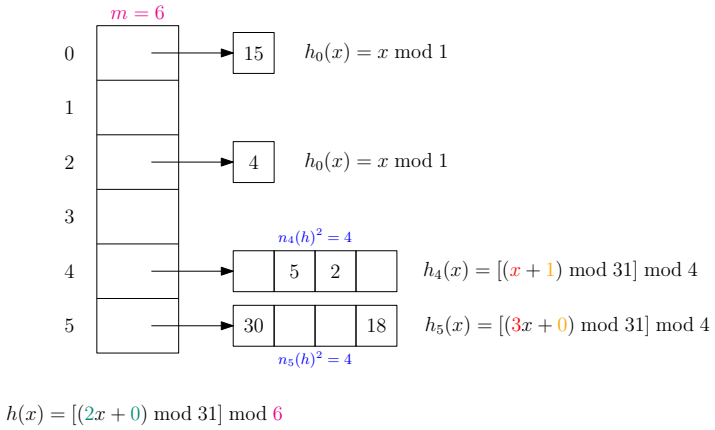


Figure 8.2: Example of a two-level hash table

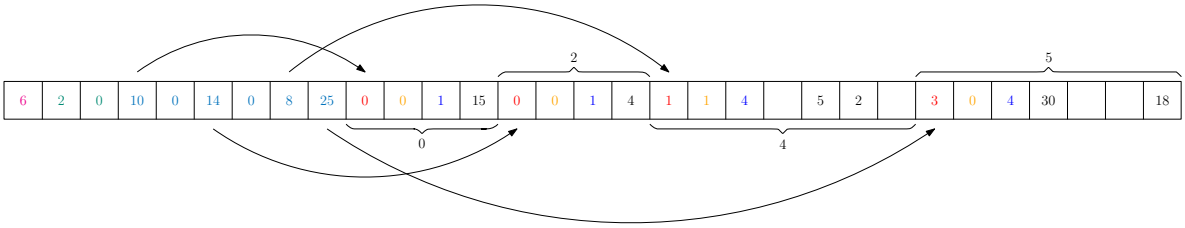


Figure 8.3: The previous two-level hash table represented by an array. The colored represents the corresponding parameters in the two-level hash table representation.

R This two-level hash table is often called the FKS hash table, which owes its name to its inventors M. Fredman, J. Komlós, and E. Szemerédi. They published the article titled *Storing a Sparse Table with O(1) Worst Case Access Time* in 1984, in which they first introduced this kind of hash table.

8.6 Open Addressing

The idea of open addressing is to store keys directly into the hash table. If the hash table is full, we cannot insert new elements, unless we create a new hash table and double the size. In open addressing, rather than mapping each key to a single slot and use chaning to resolve collision, we instead compute a sequence of slots. With such approach, the load factor α can never exceed 1.

Let $h: U \times \{0, \cdots, m-1\} \to \{0, \cdots, m-1\}$. For each key k_1, its probe sequence

$$(h(k,0), h(k,1), \cdots, h(k,m-1))$$

is a permutation of $(0, 1, \cdots, m-1)$

For INSERT(H,k), we put k into the first empty (N_i) slot in its probe sequence.

8.7 Application of Hashing

Given an array A of n numbers, determine if all of them are different.

- Sort the array and then check if adjacent numbers are duplicates. The time complexity would be $O(n \log n + n)$, where the $n \log n$ is from sorting the array and n is for iterating over the array.
- Insert every element into the hash table on at a time. Each time, check for duplicates with a search. This take $\Theta(n)$ worst-case expected time.

Given a list of n possible integers $x_1, x_2, \cdots, x_n$, determine in $O(n)$ worst-case expected time if there exists and i and j such that $x_i = x_j + 1$.

Lecture 9 Amortized Analysis

9.1 Amortized Cost

Definition 9.1.1 — Worst-Case Sequence Complexity. The worst-case sequence complexity $C(m)$ is defined as

$$C(m) = \max\{T(\sigma) \mid \sigma \text{ is a sequence of } m \text{ operations}\}$$

C is a function of the length of the sequence.

If there are many possible initial states, we use the notation $T(\sigma, S_0)$ and $C(m, S_0)$ where S_0 is an initial state.

To compute $C(m)$:

- To get an upper bound on $C(m)$, prove that for every sequence σ of m operations, the time to process σ from the initial initial state is $\leq f(m)$. Thus, $C(m) \leq f(m)$.
- To get a lower bound on $C(m)$, prove that for some sequence of m operations from the initial state, the time to process σ is $\geq g(m)$. Thus, $C(m) \geq g(m)$.

Definition 9.1.2 — Amortized Cost. The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n. Formally,

$$A(m) = \frac{C(m)}{m}$$

for a sequence of m operations.

As an example, consider if you borrowed \$100k mortgage to buy a house. You need to pay it back over 10 years. Assume interest rate is 10% per year.

There are two ways to pay the mortgage.

In general, the worst-case sequence complexity is less than or equal to m times the worst-case complexity. This happens if every operation in that sequence is the worst case. It follows that the amortized complexity is always less than or equal to the worst-case complexity.

Example 1: sorted linked list. Sequence of m insert and search operations starting from an empty list. For complexity measure, we count element comparison.

ith operation takes less than or equal to $i-1$ steps since before the ith operation, the list has length $\leq i-1$. So worst-case time for one operation in a sequence of m operations is $m-1$.

For the amortized complexity, consider the sequence $\textsc{Insert}(1), \textsc{Insert}(2), \cdots \textsc{Insert}(m)$. In this sequence, the ith operation performs exactly $i-1$ comparisons and

$$T(\sigma) = \sum_{i=1}^{m}(i-1) = \frac{m(m-1)}{2}$$

Thus the amortized complexity of insertions and searchings in a sorted linked list is at least $T(\sigma)/m = (m-1)/2$.

In this case, the worst-case complexity is a good upper bound for amortized complexity.

Example 2: increasing a binary counter starting from 0.

Consider an array $X\ [0,\cdots,k-1]$ of bits representing an integer such that

$$x = \sum_{i=0}^{k-1} X[i] \cdot 2^i.$$

Initially, $x = 0$.

We have an operation $\textsc{Increment}(x)$ which adds 1 to x modulo 2^k.

The worst-case number of bit flips in an $\textsc{Increment}$ operation is k (e.g. occurs when all bits of X are 1).

There is only one sequence of m $\textsc{Increment}$ operations. In this sequence, $X[0]$ is flipped m times; $X[1]$ is flipped $\lfloor m/2 \rfloor$; $X[2]$ is flipped $\lfloor m/4 \rfloor$; and $X[i]$ is flipped $\lfloor m/2^i \rfloor$ times.

The worst-case sequence complexity is

$$C(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{m}{2^i} \right\rfloor < m \sum_{i=0}^{\infty} 2^{-i} = 2m$$

Thus,

$$A(m) = \frac{C(m)}{m} \leq 2$$

while the worst-case complexity of an $\textsc{Increment}$ is k.

Generally, there are three methods for performing amortized analysis. Although all methods should give us the same answer, depending on the circumstances, some methods will be easier than others.

- Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n)/n$.
- The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. We use the analogy of

a bank account. Prior operations that may impact future operations can store some credits in the bank for uses by future operations. Usually, many short-running operations accumulate credits in small increments, while rare long-running operations use the credits.
- The potential method is like the accounting method, but the balance of the imaginary bank account at each state is given by the potential function.

9.2 Aggregate Method

Get an upper and lower bound on $C(m)$ worst-case sequence complexity and then divide it by m.

9.3 Accounting Method

Each kind of operation is assigned a fixed charge $a(op)$ called the allocated charge or amortized cost of that operation.

When the allocated cost $a(op)$ is greater than the actual cost of op, the excess is associated with specific objects in the data structure as credit.

Credit can be used later to help pay for operations where allocated costs are less than their actual costs. The can be viewed as storing money in a bank account for later uses.

If the total credit in the data structure is always non-negative (i.e. the bank account never runs out of money), then the total actual cost of the sequence of operations is less than or equal to the sum of allocated charges of the operations in the sequence.

To prove that the total credit in the data structure is always non-negative, we will use a credit invariant (similar to loop invariant used to prove correctness). The credit invariant is a rule that says that parts of the data structure with certain properties will have a certain number of credits associated with them. Credit invariants are proved by induction.

Sometimes, credits are needed to establish the credit invariant for the initial state. The total actual cost of the sequence is less than or equal to the cost to establish the credit invariant plus the sum of allocated charge to the operations in the sequence.

It is important not to explicitly implement the credit system in the code. It is for analysis purpose only.

The initial value of the account is 0. We define that \$1 is the cost of one bit flip.

The allocated charge to INCREMENT is \$2.

The credit invariant is: Each bit with value value 1 has \$1 credit.

When we flip a bit from 0 to 1, we use \$1 to actually flip that bit, and store the additional \$1 as the credit on that bit (in order to maintain the credit invariant).

When we flip a bit from 1 to 0, the associated \$1 credit at that bit can be used to pay for performing the bit fip.

Proof. Initially, all bits are 0, so the credit invariant is vacuously true.

Assume that the credit invariant is true before an INCREMENT operation. Then, all bits of 1 in X contain a \$1 credit.

Case 1: the i least significant bits $X[0], \cdots, X[i-1]$ are 1, and $X[i] = 0$ for some i such that $0 \leq i \leq k$. The actual number of bit flips in this case is $i+1$. Use the i credits associated with the i least significant bits to pay for flipping these bits from 1 to 0. Use \$1 of allocated charge to pay for flipping $X[i]$ from 0 to 1. Use the remaining \$1 of allocated charge to associate as credit with $X[i]$. All other 1 bits still have \$1 credit. Thus the credit invariant is true after this INCREMENT operation.

Case 2: all of the bits are 1. All k bits are flippted, use the k credits on these bits to pay for the flips. This maintains the credit invariant. The allocated charge can be donated to charity (LOL).

Then by induction, the credit invariant is true after every INCREMENT operation. The cost to establish the credit invariant is 0.

The total actual cost of m operations $C(m) \leq 0 + 2m$. Hence, the amortized cost $A(m)$ of INCREMENTE is at most 2 bit flips. ■

9.4 Dynamic Array

In this section, we will consider the data structure known as dynamic array. It is similar to a regular array, but its length changes according to the "fullness" of the array.

The array will be initialized with a fixed length. Upon calling INSERT, it will insert an element into the array, and whenver the current array becomes full, we create a longer array and copy everything into the new array. Similarly, after calling DELETE, we will shrink the array whenever the array becomes too empty. If we only consider the worst case, the runtime complexities are bad: $O(n)$ for both operations. However, if we look at the amortized cost, the runtime is actually smaller.

9.4.1 Accounting Method Analysis for Dynamic Array

9.5 Potential Method

Prepaid wors in represented as potential energy associated with the whole data structure. Potential energy in a data structure can be released to pay for later operations.

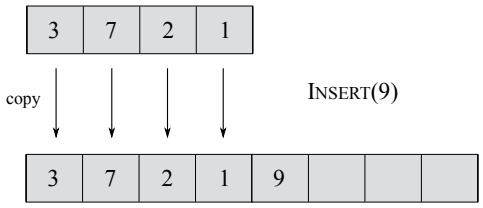


Figure 9.1 The dynamic array after the operation INSERT(9). The length of the new array is doubled, and old elements are copied to the new array.

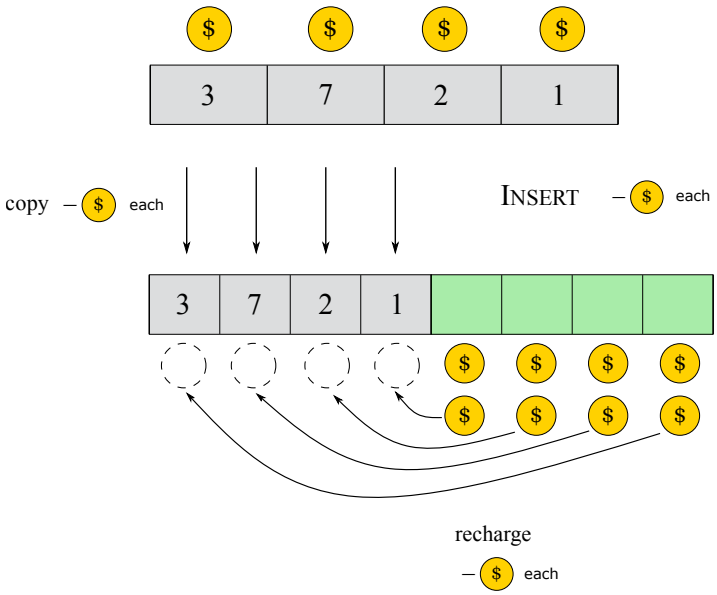


Figure 9.2: <caption>

Definition 9.5.1 — Potential Function. Let D be some data structure with initial state D_0. For each $i = 1, 2, \cdots, n$, let c_i be the actual cost of the i-th operation and D_i be the resulting data structure after applying the i-th operation to data structure D_{i-1}.

The potential function Φ maps each data structure at state i, denoted D_i, to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i.

The amortized cost $\hat{c}_i$ of the i-th operation with respect to the potential function Φ is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

And the total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

9.5.1 Analysis of Binary Counter Using Potential Method

Let C_i be the configuration after ith INCREMENT.

Let $\Phi(c)$ be the number of 1 bits in the stored representation of the counter in counfiguration C.

$\Phi(C) \geq 0$ for all configurations C. $\Phi(C) = 0$ since all bits are 0 initially.

III Advanced Data Structures

Lecture 10 Dynamic Arrays

Lecture 11 Fibonacci Heaps

Lecture 12 Disjoint Sets

IV Lower Bounds

Lecture 13 Decision Trees

13.1 Lower Bounds

So far, we have been talking almost exclusively about how we can use different algorithms and data structures to solve certain problems as fast as possible. In this part, we will focus on proving certain problems cannot be solved as quickly as we might want. In other words, there is a limit to how good we can do. For example, in a comparison model, sorting can only be achieved at best in $\Omega(n \log n)$ time in the worst case.

Let $T_A(X)$ be the running time of algorithm A given input X. Then, the worst-case running time is

$$T_A(n) = \max_{|X|=n} T_A(X)$$

The worst-case complexity of a **problem** Π is the worst-case running time of the fastest algorithm for solving it.

$$T_\Pi(n) = \min_{A \text{ solves } \Pi} T_A(n) = \min_{A \text{ solves } \Pi} \max_{|X|=n} T_A(X)$$

We can prove the upper-bound of the complexity of a problem by giving a specific algorithm A that solves Π, and faster algorithms give us smaller (tighter and better) upper bounds.

However, to prove that a problem has a certain lower bound, we have to show that every algorithm that solves Π has a worst-case running time $\Omega(f(n))$, or equivalently, that there is no algorithm that solves Π that runs in $o(f(n))$ time. To be more specific, we need to specify what kinds of algorithms we want to consider, which is formally known as model of computation. For example, the comparison model is one model that is used to solve sorting and searching problems.

13.2 Comparison Model

In the comparison model, we consider all input items as black boxes, or more precisely, ADTs. The only operations allowed on the items are comparisons: $<, \leq, >, \geq, =$. Most searching and sorting algorithms we have been looking at so far use the comparison model: heap sort, merge sort, binary search and binary search tree, etc. In the comparison model, we count the number of comparisons and define it as the time cost of the algorithm.

13.3 Decision Tree

13.3.1 Intuition

Any comparison algorithm can be viewed as a tree of all possible comparisons, the outcomes of the comparisons, and the resulting answer. This tree is called a decision tree.

For any particular n,

- *internal node* corresponds to binary decision in the algorithm (in this case, binary comparisons)
- *leaf* corresponds to a possible answer of the problem
- *root-to-leaf path* corresponds to an execution of the algorithm
- *length of the root-to-leaf path* corresponds to the time cost of the execution associated with that path
- *height of the tree* (or depth of the deepest leaf) corresponds to the worst-case running time.

13.3.2 Decision Tree for Searching

In this subsection, we will look at the decision tree for binary search and use it to prove that the lower bound of searching under the comparison model is $\Omega(\log n)$.

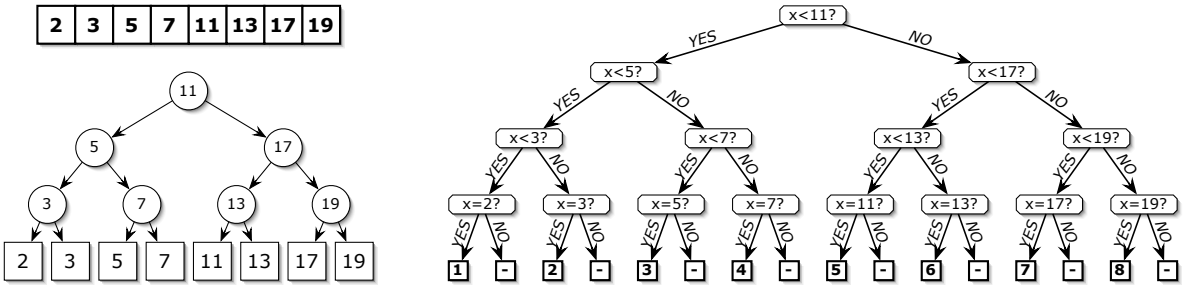


Figure 13.1: <caption>

13.3.3 Decision Tree for Sorting

13.4 Sorting in Linear Time

Lecture 14 Information Theory

Lecture 15 Adversary Arguments

Lecture 16 Reduction

V

Graphs

Lecture 17 Breadth-First Search

17.1 Definition

Definition 17.1.1 — Graphs. A graph is defined as a tuple $G = (V, E)$ where $V$is an arbitrary non-empty finite set, whose elements are called vertices or nodes; and E is a set of pairs of elements of V, which we call edges. For an undirected graph, the edges are unordered pairs u, v. In a directed graph, the edges are ordered pairs (u, v).

Definition 17.1.2 — Neighbors and Degrees. For any edge uv in an undirected graph, we call u neighbor of v and vice versa, and we say that u and v are adjacent. The degree of a node is its number of neighbors.

In directed graphs, for every edge $u \to v$, we call u a predecessor of v, and we call v a successor of u. The in-degree of a vertex is its number of predecessors; the out-degree is its number of sucessors.

Definition 17.1.3 — Subgraphs. A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A proper subgraph of G is any subgraph that is not G itself.

17.2 Representations of Graphs

17.2.1 Adjacency List

Definition 17.2.1 — Adjacency List. The adjacency-list representation of a graph $G = (V, E)$ consists of an arrray Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, Adj[u] contains all the vertices adjacent to u in G.

17.3 Breadth-first Search

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to discover every vertex that is reachable from s. Breadth-first search expands the frontier between disvoered and undiscovered vertices uniformly across the breadth of the frontier. The algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

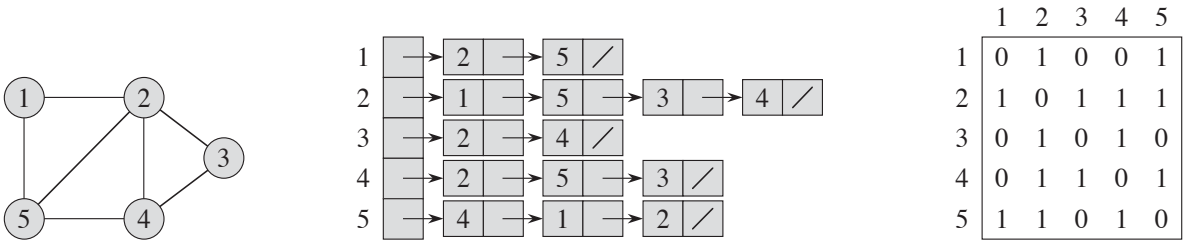


Figure 17.1: Representations of undirected graph: (a) the graph; (b) adjacency list of the graph; (c) adjacency matrix of the graph.

Lecture 18 Depth-First Search

Lecture 19 Minimum Spanning Trees

Lecture 20 Bellman-Ford's Algorithm

Lecture 21 Dijkstra's Algorithm

Appendix

Commonly Used Axioms & Theorems

Rules of Inference

Axiom 1 — Modus Ponens. $(P \wedge (P \text{ IMPLIES } Q)) \text{ IMPLIES } Q$

Axiom 2 — Modus Tollens. $(\neg Q \wedge (P \text{ IMPLIES } Q)) \text{ IMPLIES } \neg P$

Axiom 3 — Hypothetical Syllogism (transitivity).

$((P \text{ IMPLIES } Q) \wedge (Q \text{ IMPLIES } R)) \text{ IMPLIES } (P \text{ IMPLIES } R)$

Axiom 4 — Disjunctive Syllogism. $((P \vee Q) \wedge \neg P) \text{ IMPLIES } Q$

Axiom 5 — Addition. $P \text{ IMPLIES } (P \vee Q)$

Axiom 6 — Simplification. $(P \wedge Q) \text{ IMPLIES } P$

Axiom 7 — Conjunction. $((P) \wedge (Q)) \text{ IMPLIES } (P \wedge Q)$

Axiom 8 — Resolution. $((P \vee Q) \wedge (\neg P \vee R))$ IMPLIES $(Q \vee R)$

Laws of Logic

Axiom 9 — Implication Law. $(P$ IMPLIES $Q) \equiv (\neg P \vee Q)$

Axiom 10 — Distributive Law.

$$(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$$
$$(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$$

Axiom 11 — De Morgan's Law.

$$\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$$
$$\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$$

Axiom 12 — Absorption Law.

$$(P \vee (P \wedge Q)) \equiv P$$
$$(P \wedge (P \vee Q)) \equiv P$$

Axiom 13 — Commutativity of AND. $A \wedge B \equiv B \wedge A$

Axiom 14 — Associativity of AND. $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$

Axiom 15 — Identity of AND. $\mathbf{T} \wedge A \equiv A$

Axiom 16 — Zero of AND. $\mathbf{F} \wedge A \equiv \mathbf{F}$

Axiom 17 — Idempotence for AND. $A \wedge A \equiv A$

Axiom 18 — Contradiction for AND. $A \wedge \neg A \equiv \mathbf{F}$

Axiom 19 — Double Negation. $\neg(\neg A) \equiv A$

Axiom 20 — Validity for OR. $A \vee \neg A \equiv \mathbf{T}$

Induction

Axiom 21 — Well Ordering Principle. Every nonempty set of nonnegative integers has a smallest element. i.e., For any $A \subset \mathbb{N}$ such that $A \neq \emptyset$, there is some $a \in A$ such that $\forall a' \in A . a \leq a'$.

Basic Prerequisite Mathematics

Basic Prerequisite Mathematics

SET THEORY

Common Sets

- $\mathbb{N} = \{0, 1, 2, \dots\}$: the natural numbers, or non-negative integers. The convention in computer science is to include 0 in the natural numbers.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$: the integers
- $\mathbb{Z}^+ = \{1, 2, 3, \dots\}$: the positive integers
- $\mathbb{Z}^- = \{-1, -2, -3, \dots\}$: the negative integers
- $\mathbb{Q}$ the rational numbers, $\mathbb{Q}^+$ the positive rationals, and $\mathbb{Q}^-$ the negative rationals.
- $\mathbb{R}$ the real numbers, $\mathbb{R}^+$ the positive reals, and $\mathbb{R}^-$ the negative reals.

Notation

For any sets A and B, we will use the following standard notation.

- $x \in A$: "x is an element of A" or "A contains x"
- $A \subseteq B$: "A is a subset of B" or "A is included in B"
- $A = B$: "A equals B" (Note that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.)
- $A \subsetneq B$: "A is a proper subset of B"
  (Note that $A \subsetneq B$ if and only if $A \subseteq B$ and $A \neq B$.)
- $A \cup B$: "A union B"
- $A \cap B$: "A intersection B"
- $A - B$: "A minus B" (*set* difference)
- $|A|$: "cardinality of A" (the number of elements of A)
- $\varnothing$ or $\{\}$: "the empty set"
- $\mathcal{P}(A)$ or 2^A: "powerset of A" (the set of all subsets of A)
  If $A = \{a, 34, \triangle\}$, then $\mathcal{P}(A) = \big\{\{\}, \{a\}, \{34\}, \{\triangle\}, \{a, 34\}, \{a, \triangle\}, \{34, \triangle\}, \{a, 34, \triangle\}\big\}$.
  $S \in \mathcal{P}(A)$ means the same as $S \subseteq A$.
- $\{x \in A \mid P(x)\}$: "the set of elements x in A for which $P(x)$ is true"
  For example, $\{x \in \mathbb{Z} \mid \cos(\pi x) > 0\}$ represents the set of integers x for which $\cos(\pi x)$ is greater than zero, *i.e.*, it is equal to $\{\dots, -4, -2, 0, 2, 4, \dots\} = \{x \in \mathbb{Z} \mid x \text{ is even}\}$.

- $A \times B$: "the cross product or Cartesian product of A and B"
  $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$.
  If $A = \{1, 2, 3\}$ and $B = \{5, 6\}$, then $A \times B = \{(1, 5), (1, 6), (2, 5), (2, 6), (3, 5), (3, 6)\}$.)
- A^n: "the cross product of n copies of A"
  This is set of all sequences of $n \geq 1$ elements, each of which is in A.
- B^A or $A \to B$: "the set of all functions from A to B."
- $f : A \to B$ or $f \in B^A$: "f is a function from A to B"
  f associates one element $f(x) \in B$ to every element $x \in A$.

NUMBER THEORY

For any two natural numbers a and b, we say that a *divides* b if there exists a natural number c such that $b = ac$. In such a case, we say that a is a *divisor* of b (*e.g.*, 3 is a divisor of 12 but 3 is not a divisor of 16). Note that any natural number is a divisor of 0 and 1 is a divisor of any natural number. A number a is *even* if 2 divides a and is *odd* if 2 does not divide a.

A natural number p is *prime* if it has exactly two positive divisors (*e.g.*, 2 is prime since its positive divisors are 1 and 2 but 1 is **not** prime since it only has one positive divisor: 1). There are an infinite number of prime numbers and any integer greater than one can be expressed in a unique way as a finite product of prime numbers (*e.g.*, $8 = 2^3$, $77 = 7 \times 11$, $3 = 3$).

Inequalities

For any integers m and n, $m < n$ if and only if $m + 1 \leq n$ and $m > n$ if and only if $m \geq n + 1$. For any real numbers w, x, y, and z, the following properties always hold (they also hold when $<$ and $\leq$ are exchanged throughout with $>$ and $\geq$, respectively).

- if $x < y$ and $w \leq z$, then $x + w < y + z$
- if $x < y$, then $\begin{cases} xz < yz & \text{if } z > 0 \\ xz = yz & \text{if } z = 0 \\ xz > yz & \text{if } z < 0 \end{cases}$
- if $x \leq y$ and $y < z$ (or if $x < y$ and $y \leq z$), then $x < z$

Functions

Here are some common number-theoretic functions together with their definitions and properties of them. (Unless noted otherwise, in this section, x and y represent arbitrary real numbers and k, m, and n represent arbitrary positive integers.)

- $\min\{x, y\}$: "minimum of x and y" (the smallest of x or y)

  Properties: $\min\{x, y\} \leq x$
  $\min\{x, y\} \leq y$

- $\max\{x, y\}$: "maximum of x and y" (the largest of x or y)

  Properties: $x \leq \max\{x, y\}$
  $y \leq \max\{x, y\}$

- $\lfloor x \rfloor$: "floor of x" (the greatest integer less than or equal to x, *e.g.*, $\lfloor 5.67 \rfloor = 5$, $\lfloor -2.01 \rfloor = -3$)

  Properties: $x - 1 < \lfloor x \rfloor \leq x$
  $\lfloor -x \rfloor = -\lceil x \rceil$
  $\lfloor x + k \rfloor = \lfloor x \rfloor + k$
  $\lfloor \lfloor k/m \rfloor / n \rfloor = \lfloor k/mn \rfloor$
  $(k - m + 1)/m \leq \lfloor k/m \rfloor$

- $\lceil x \rceil$: "ceiling of x" (the least integer greater than or equal to x, *e.g.*, $\lceil 5.67 \rceil = 6$, $\lceil -2.01 \rceil = -2$)

  Properties: $x \leq \lceil x \rceil < x + 1$
  $\lceil -x \rceil = -\lfloor x \rfloor$
  $\lceil x + k \rceil = \lceil x \rceil + k$
  $\lceil \lceil k/m \rceil / n \rceil = \lceil k/mn \rceil$
  $\lceil k/m \rceil \leq (k + m - 1)/m$

  Additional property of $\lfloor\ \rfloor$ and $\lceil\ \rceil$: $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$.

- $|x|$: "absolute value of x" ($|x| = x$ if $x \geq 0$; $-x$ if $x < 0$, *e.g.*, $|5.67| = 5.67$, $|-2.01| = 2.01$)

  BEWARE! The same notation is used to represent the cardinality $|A|$ of a set A and the absolute value $|x|$ of a number x so be sure you are aware of the context in which it is used.

- $m \operatorname{div} n$: "the quotient of m divided by n" (integer division of m by n, *e.g.*, $5 \operatorname{div} 6 = 0$, $27 \operatorname{div} 4 = 6$, $-27 \operatorname{div} 4 = -6$)

  Properties: If $m, n > 0$, then $m \operatorname{div} n = \lfloor m/n \rfloor$
  $(-m) \operatorname{div} n = -(m \operatorname{div} n) = m \operatorname{div} (-n)$

- $m \operatorname{rem} n$: "the remainder of m divided by n" (*e.g.*, $5 \operatorname{rem} 6 = 5$, $27 \operatorname{rem} 4 = 3$, $-27 \operatorname{rem} 4 = -3$)

  Properties: $m = (m \operatorname{div} n) \cdot n + m \operatorname{rem} n$
  $(-m) \operatorname{rem} n = -(m \operatorname{rem} n) = m \operatorname{rem} (-n)$

- $m \bmod n$: "m modulo n" (*e.g.*, $5 \bmod 6 = 5$, $27 \bmod 4 = 3$ $-27 \bmod 4 = 1$)

  Properties: $0 \leq m \bmod n < n$
  n divides $m - (m \bmod n)$.

- $\gcd(m, n)$: "greatest common divisor of m and n" (the largest positive integer that divides both m and n)

  For example, $\gcd(3, 4) = 1$, $\gcd(12, 20) = 4$, $\gcd(3, 6) = 3$

- $\operatorname{lcm}(m, n)$: "least common multiple of m and n" (the smallest positive integer that m and n both divide)

  For example, $\operatorname{lcm}(3, 4) = 12$, $\operatorname{lcm}(12, 20) = 60$, $\operatorname{lcm}(3, 6) = 6$

  Properties: $\gcd(m, n) \cdot \operatorname{lcm}(m, n) = m \cdot n$.

CALCULUS

Limits and Sums

An infinite sequence of real numbers $\{a_n\} = a_1, a_2, \ldots, a_n, \ldots$ *converges* to a limit $L \in \mathbb{R}$ if, for every $\varepsilon > 0$, there exists $n_0 \geq 0$ such that $|a_n - L| < \varepsilon$ for every $n \geq n_0$. In this case, we write $\lim_{n\to\infty} a_n = L$ Otherwise, we say that the sequence *diverges*.

If $\{a_n\}$ and $\{b_n\}$ are two sequences of real numbers such that $\lim_{n\to\infty} a_n = L_1$ and $\lim_{n\to\infty} b_n = L_2$, then

$$\lim_{n\to\infty}(a_n + b_n) = L_1 + L_2 \quad \text{and} \quad \lim_{n\to\infty}(a_n \cdot b_n) = L_1 \cdot L_2.$$

In particular, if c is any real number, then

$$\lim_{n\to\infty}(c \cdot a_n) = c \cdot L_1.$$

The sum $a_1 + a_2 + \cdots + a_n$ and product $a_1 \cdot a_2 \cdots a_n$ of the finite sequence $a_1, a_2, \ldots, a_n$ are denoted by

$$\sum_{i=1}^{n} a_i \text{ and } \prod_{i=1}^{n} a_i.$$

If the elements of the sequence are all different and $S = \{a_1, a_2, \ldots, a_n\}$ is the set of elements in the sequence, these can also be denoted by

$$\sum_{a\in S} a \text{ and } \prod_{a\in S} a.$$

Examples:

- For any $a \in \mathbb{R}$ such that $-1 < a < 1$, $\lim_{n\to\infty} a^n = 0$.
- For any $a \in \mathbb{R}^+$, $\lim_{n\to\infty} a^{1/n} = 1$.
- For any $a \in \mathbb{R}^+$, $\lim_{n\to\infty} (1/n)^a = 0$.
- $\lim_{n\to\infty}(1 + 1/n)^n = e = 2.71828182845904523536\ldots$

- For any $a, b \in \mathbb{R}$, the *arithmetic* sum is given by:

$$\sum_{i=0}^{n}(a + ib) = (a) + (a + b) + (a + 2b) + \cdots + (a + nb) = \frac{1}{2}(n + 1)(2a + nb).$$

- For any $a, b \in \mathbb{R}^+$, the *geometric* sum is given by:

$$\sum_{i=0}^{n}(ab^i) = a + ab + ab^2 + \cdots + ab^n = \frac{a(1 - b^{n+1})}{1 - b}.$$

EXPONENTS AND LOGARITHMS

Definition: For any $a, b, c \in \mathbb{R}^+$, $a = \log_b c$ if and only if $b^a = c$.

Notation: For any $x \in \mathbb{R}^+$, $\ln x = \log_e x$ and $\lg x = \log_2 x$.

For any $a, b, c \in \mathbb{R}^+$ and any $n \in \mathbb{Z}^+$, the following properties always hold.

- $\sqrt[n]{b} = b^{1/n}$
- $b^a b^c = b^{a+c}$
- $(b^a)^c = b^{ac}$
- $b^a / b^c = b^{a-c}$
- $b^0 = 1$
- $a^b c^b = (ac)^b$
- $b^{\log_b a} = a = \log_b b^a$
- $a^{\log_b c} = c^{\log_b a}$
- $\log_b(ac) = \log_b a + \log_b c$
- $\log_b(a^c) = c \cdot \log_b a$
- $\log_b(a/c) = \log_b a - \log_b c$
- $\log_b 1 = 0$
- $\log_b a = \log_c a / \log_c b$

BINARY NOTATION

A *binary number* is a sequence of bits $a_k \cdots a_1 a_0$ where each bit a_i is equal to 0 or 1. Every binary number represents a natural number in the following way:

$$(a_k \cdots a_1 a_0)_2 = \sum_{i=0}^{k} a_i 2^i = a_k 2^k + \cdots + a_1 2 + a_0.$$

For example, $(1001)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 1 = 9$, $(01110)_2 = 8 + 4 + 2 = 14$.

Properties:

- If $a = (a_k \cdots a_1 a_0)_2$, then $2a = (a_k \cdots a_1 a_0 0)_2$, *e.g.*, $9 = (1001)_2$ so $18 = (10010)_2$.
- If $a = (a_k \cdots a_1 a_0)_2$, then $\lfloor a/2 \rfloor = (a_k \cdots a_1)_2$, *e.g.*, $9 = (1001)_2$ so $4 = (100)_2$.
- The smallest number of bits required to represent the positive integer n in binary is called the *length* of n and is equal to $\lceil \log_2(n+1) \rceil$.

Make sure you know how to add and multiply two binary numbers. For example, $(1111)_2 + (101)_2 = (10100)_2$ and $(1111)_2 \times 101)_2 = (1001011)_2$.

Proof Templates

Proof Outlines

LINE NUMBERS: Only lines that are referred to have labels (for example, L1) in this document. For a formal proof, all lines are numbered. Line numbers appear at the beginning of a line. You can indent line numbers together with the lines they are numbering or all line numbers can be unindented, provided you are consistent.

INDENTATION: Indent when you make an assumption or define a variable. Unindent when this assumption or variable is no longer being used.

1. **Implication**: Direct proof of A IMPLIES B.

    L1. Assume A.
    ⋮
    L2. B
   A IMPLIES B; direct proof: L1, L2

2. **Implication**: Indirect proof of A IMPLIES B.

    L1. Assume $NOT(B)$.
    ⋮
    L2. $NOT(A)$
   A IMPLIES B; indirect proof: L1, L2

3. **Equivalence**: Proof of A IFF B.

    L1. Assume A.
    ⋮
    L2. B
   L3. A IMPLIES B; direct proof: L1, L2
    L4. Assume B.
    ⋮
    L5. A
   L6. B IMPLIES A; direct proof: L4, L5
   A IFF B; equivalence: L3, L6

4. **Proof by contradiction** of A.

    L1. To obtain a contradiction, assume $NOT(A)$.
    ⋮
        L2. B
    ⋮
        L3. $NOT(B)$
    L4. This is a contradiction: L2, L3
   Therefore A; proof by contradiction: L1, L4

5. **Modus Ponens**.

   ⋮
   L1. A
   ⋮
   L2. A IMPLIES B
   B; modus ponens: L1, L2

6. **Conjunction:** Proof of A AND B:

   ⋮
   L1. A
   ⋮
   L2. B
   A AND B; proof of conjunction; L1, 2

7. **Use of Conjunction**:

   ⋮
   L1. A AND B
   A; use of conjunction: L1
   B; use of conjunction: L1

8. **Implication with Conjunction**: Proof of (A_1 AND A_2) IMPLIES B.

      L1. Assume A_1 AND A_2.
      A_1; use of conjunction, L1
      A_2; use of conjunction, L1
      ⋮
      L2. B
   (A_1 AND A_2) IMPLIES B; direct proof, L1, L2

9. **Implication with Conjunction**: Proof of A IMPLIES (B_1 AND B_2).

      L1. Assume A.
      ⋮
      L2. B_1
      ⋮
      L3. B_2
      L4. B_1 AND B_2; proof of conjunction: L2, L3
   A IMPLIES (B_1 AND B_2); direct proof: L1, L4

10. **Disjunction**: Proof of A OR B and B OR A.

    ⋮
    L1. A
    A OR B; proof of disjunction: L1
    B OR A; proof of disjunction: L1

11. **Proof by cases**.

    L1. C OR $NOT(C)$ tautology
    L2. Case 1: Assume C.
    $\vdots$
    L3. A
    L4. C IMPLIES A; direct proof: L2, L3
    L5. Case 2: Assume $NOT(C)$.
    $\vdots$
    L6. A
    L7. $NOT(C)$ IMPLIES A; direct proof: L5, L6
    A proof by cases: L1, L4, L7

12. **Proof by cases** of A OR B.

    L1. C OR $NOT(C)$ tautology
    L2. Case 1: Assume C.
    $\vdots$
    L3. A
    L4. A OR B; proof of disjunction, L3
    L5. C IMPLIES (A OR B); direct proof, L2, L4
    L6. Case 2: Assume $NOT(C)$.
    $\vdots$
    L7. B
    L8. A OR B; proof of disjunction, L7
    L9. $NOT(C)$ IMPLIES (A OR B); direct proof: L6, L8
    A OR B; proof by cases: L1, L5, L9

13. **Implication with Disjunction**: Proof by cases of (A_1 OR A_2) IMPLIES B.

    L1. Case 1: Assume A_1.
    $\vdots$
    L2. B
    L3. A_1 IMPLIES B; direct proof: L1,L2
    L4. Case 2: Assume A_2.
    $\vdots$
    L5. B
    L6. A_2 IMPLIES B; direct proof: L4, L5
    (A_1 OR A_2) IMPLIES B; proof by cases: L3, L6

14. **Implication with Disjunction**: Proof by cases of A IMPLIES (B_1 OR B_2).

    L1. Assume A.
    L2. C OR NOT(C) tautology
    L3. Case 1: Assume C.
        ⋮
        L4. B_1
        L5. B_1 OR B_2; disjunction: L4
    L6. C IMPLIES (B_1 OR B_2); direct proof: L3, L5
    L7. Case 2: AssumeNOT(C).
        ⋮
        L8. B_2
        L9. B_1 OR B_2; disjunction: L8
    L10. NOT(C) IMPLIES (B_1 OR B_2); direct proof: L7, L9
    L11. B_1 OR B_2; proof by cases: L2, L6, L10
    A IMPLIES (B_1 OR B_2): direct proof. L1, L11

15. **Substitution of a Variable in a Tautology**:
    Suppose P is a propositional variable, Q is a formula, and R' is obtained from R by replacing *every* occurrence of P by (Q).

    L1. R tautology
    R'; substitution of all P by Q: L1

16. **Substitution of a Formula by a Logically Equivalent Formula**:
    Suppose S is a subformula of R and R' is obtained from R by replacing *some* occurrence of S by S'.

    L1. R
    L2. S IFF S'
    L3. R'; substitution of an occurrence of S by S': L1, L2

17. **Specialization**:

    L1. $c \in D$
    L2. $\forall x \in D.P(x)$
    $P(c)$; specialization: L1, L2

18. **Generalization**: Proof of $\forall x \in D.P(x)$.

    L1. Let x be an arbitrary element of D.
    ⋮
    L2. $P(x)$
    Since x is an arbitrary element of D,
    $\forall x \in D.P(x)$; generalization: L1, L2

19. **Universal Quantification with Implication**: Proof of $\forall x \in D.(P(x)$ IMPLIES $Q(x))$.

    L1. Let x be an arbitrary element of D.
        L2. Assume $P(x)$
        $\vdots$
        L3. $Q(x)$
    L4. $P(x)$ IMPLIES $Q(x)$; direct proof: L2, L3
    Since x is an arbitrary element of D,
    $\forall x \in D.(P(x)$ IMPLIES $Q(x))$; generalization: L1, L4

20. **Implication with Universal Quantification**: Proof of $(\forall x \in D.P(x))$ IMPLIES A.

    L1. Assume $\forall x \in D.P(x)$.
    $\vdots$
    L2. $a \in D$
    $P(a)$; specialization: L1, L2
    $\vdots$
    L3. A
    Therefore $(\forall x \in D.P(x))$ IMPLIES A; direct proof: L1, L3

21. **Implication with Universal Quantification**: Proof of A IMPLIES $(\forall x \in D.P(x))$.

    L1. Assume A.
        L2. Let x be an arbitrary element of D.
        $\vdots$
        L3. $P(x)$
    Since x is an arbitrary element of D,
    L4. $\forall x \in D.P(x)$; generalization, L2, L3
    A IMPLIES $(\forall x \in D.P(x))$; direct proof: L1, L4

22. **Instantiation**:

    L1. $\exists x \in D.P(x)$
    Let $c \in D$ be such that $P(c)$; instantiation: L1
    $\vdots$

23. **Construction**: Proof of $\exists x \in D.P(x)$.

    L1. Let $a = \cdots$
    $\vdots$
    L2. $a \in D$
    $\vdots$
    L3. $P(a)$
    $\exists x \in D.P(x)$; construction: L1, L2, L3

24. **Existential Quantification with Implication**: Proof of $\exists x \in D.(P(x)$ IMPLIES $Q(x))$.

    L1. Let $a = \cdots$

    $\vdots$

    L2. $a \in D$

        L3. Suppose $P(a)$.

        $\vdots$

        L4. $Q(a)$

    L5. $P(a)$ IMPLIES $Q(a)$; direct proof: L3, L4

    $\exists x \in D.(P(x)$ IMPLIES $Q(x))$; construction: L1, L2, L5

25. **Implication with Existential Quantification**: Proof of $(\exists x \in D.P(x))$ IMPLIES A.

    L1. Assume $\exists x \in D.P(x)$.

        Let $a \in D$ be such that $P(a)$; instantiation: L1

        $\vdots$

        L2. A

    $(\exists x \in D.P(x))$ IMPLIES A; direct proof: L1, L2

26. **Implication with Existential Quantification**: Proof of A IMPLIES $(\exists x \in D.P(x))$.

    L1. Assume A.

        L2. Let $a = \cdots$

        $\vdots$

        L3. $a \in D$

        $\vdots$

        L4. $P(a)$

    L5. $\exists x \in D.P(x)$; construction: L2, L3, L4

    A IMPLIES $(\exists x \in D.P(x))$; direct proof: L1, L5

27. **Subset**: Proof of $A \subseteq B$.

    L1. Let $x \in A$ be arbitrary.

    $\vdots$

    L2. $x \in B$

    The following line is optional:

    L3. $x \in A$ IMPLIES $x \in B$; direct proof: L1, L2

    $A \subseteq B$; definition of subset: L3 (or L1, L2, if the optional line is missing)

28. **Weak Induction**: Proof of $\forall n \in N.P(n)$

    Base Case:
    $\vdots$
    L1. $P(0)$
        L2. Let $n \in N$ be arbitrary.
            L3. Assume $P(n)$.
            $\vdots$
            L4. $P(n+1)$
        The following two lines are optional:
        L5. $P(n)$ IMPLIES ($P(n+1)$; direct proof of implication: L3, L4
    L6. $\forall n \in N.(P(n)$ IMPLIES $P(n+1))$; generalization L2, L5
    $\forall n \in N.P(n)$ induction; L1, L6 (or L1, L2, L3, L4, if the optional lines are missing)

29. **Strong Induction**: Proof of $\forall n \in N.P(n)$

        L1. Let $n \in N$ be arbitrary.
            L2. Assume $\forall j \in N.(j < n$ IMPLIES $P(j))$
            $\vdots$
            L3. $P(n)$
        The following two lines are optional:
        L4. $\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)$; direct proof of implication: L2, L3
    L5. $\forall n \in N.[\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)]$; generalization: L1, L4
    $\forall n \in N.P(n)$; strong induction: L5 (or L1, L2, L3, if the optional lines are missing)

30. **Structural Induction**: Proof of $\forall e \in S.P(e)$, where S is a recursively defined set

    Base case(s):
        L1. For each base case e in the definition of S
        L2. $P(e)$.
    Constructor case(s):
        L3. For each constructor case e of the definition of S,
            L4. assume $P(e')$ for all components e' of e.
            $\vdots$
            L5. $P(e)$
    $\forall e \in S.P(e)$; structural induction: L1, L2, L3, L4, L5

31. **Well Ordering Principle**: Proof of $\forall e \in S.P(e)$, where S is a well ordered set, i.e. every nonempty subset of S has a smallest element.

    L1. To obtain a contradiction, suppose that $\forall e \in S.P(e)$ is false.
    L2. Let $C = \{e \in S \mid P(e) \text{ is false}\}$ be the set of counterexamples to P.
    L3. $C \neq \phi$; definition: L1, L2
        L4. Let e be the smallest element of C; well ordering principle: L2, L3
            Let $e' = \cdots$
            $\vdots$
            L5. $e' \in C$
            $\vdots$
            L6. $e' < e$.
    L7. This is a contradiction: L4, L5, L6

$\forall e \in S.P(e)$; proof by contradiction: L1, L7

Index

Index

Bibliography

Courses

[2] Erik Demaine and Srini Devadas. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. Fall 2011.

[3] Erik Demaine, Srini Devadas, and Nancy Lynch. *6.046J Design and Analysis of Algorithms*. Massachusetts Institute of Technology. Spring 2015.

[4] Faith Ellen. *CSC240S1 Winter 2021*. University of Toronto. 2021.

[5] Faith Ellen. *CSC265F1 Fall 2021*. University of Toronto. 2021.

[8] Mauricio Karchmer, Anand Natarajan, and Nir Shavit. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. Spring 2021.

Books

[1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[7] Vassos Hadzilacos. *Course notes for CSC B36/236/240 Introduction to Theory of Computation*. University of Toronto. 2007.

[9] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. 8th edition. New York: McGraw-Hill Education, 2019.

Journal Articles

[6] Michael Fredman, Janos Komlos, and Endre Szemeredi. “Storing a Sparse Table with O(1) Worst Case Access Time”. In: *Journal of the Association for Computing Machinery* 31.3 (July 1984), pages 538–544.

Tutorial