CSC263

Data Structures and Analysis

Sinan Li

2023

CONTENTS

Part I

Data Structure

CHAPTER 1

PRIORITY QUEUES AND HEAPS

Data

- Collection of elements
- Each element x has a priority $x.priority$

Operations

- INSERT(Q, x)
  Add x to Q
  Note: $x.priority$ can be non-unique
- MAX(Q)
  Return the element with max priority
  Note: Q is unchanged
- EXTRACT-MAX(Q)
  Remove and return the element with the max priority

1.1 Implementation

1.1.1 Attempts

Implementation 1: Unsorted Array / Linked List

- INSERT takes $\Theta(1)$ time in the worst case
- MAX takes $\Theta(n)$ time in the worst case
- EXTRACTMAX takes $\Theta(n)$ time in the worst case

Implementation 2: Sorted Array / Linked List

- INSERT takes $\Theta(n)$ time in the worst case
- MAX takes $\Theta(1)$ time in the worst case
- EXTRACTMAX takes $\Theta(1)$ time in the worst case

1.1.2 Implementation

We want to combine the advantages of both data structures by having a "partially sorted" ADT – a (binary) *heap*.
There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a **heap property**, the specifics of which depend on the kind of heap.

- Max heap property: the key of every node x is *larger* than or equal to the keys of its children.

  The largest element in a max-heap is stored at the root.
- Min heap property: the key of every node x is *smaller* than or equal to the keys of its children

  The smallest element in a min-heap is stored at the root,

These are called *heap orders*. There is no ordering between the siblings. A max/min heap is valid if it is a nearly complete binary tree and it satisfies the max/min heap property.

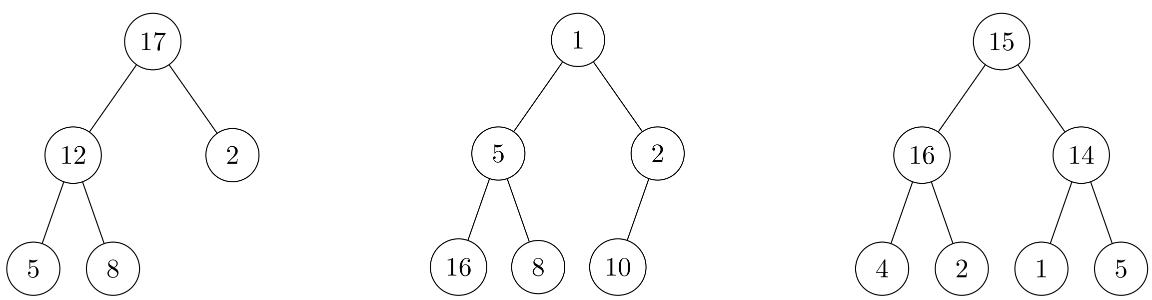


Figure 1.1: A valid max-heap (left), a valid min-heap (middle), and an invalid heap (right)

Although a heap is an **almost complete binary tree** [1], in practice, we usually use an array to store the data in memory. An array H that represents a heap is an object with two attributes: $H.length$, which (as usual) gives the number of elements in the array, and $H.heap\text{-}size$, which represents how many elements in the heap are stored within array H. The root of the tree is $H[1]$, and given the index i of a node, we can compute the indices of its parent, left child, and right child:

- PARENT

  return $\lfloor i/2 \rfloor$
- LEFT

  return $2i$
- RIGHT

  return $2i + 1$

[1] That is, the tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

1.2 Operations

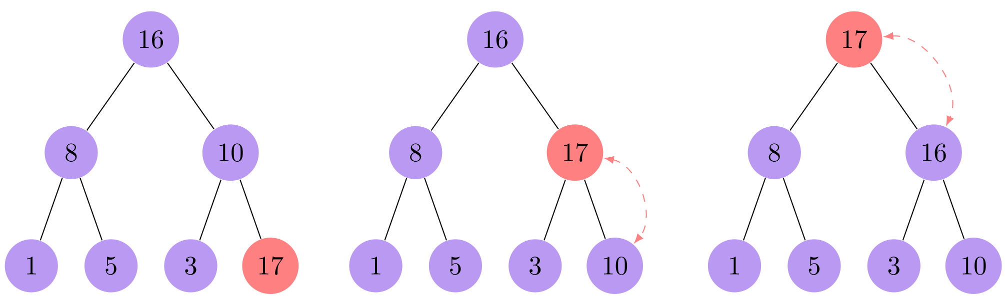
1.2.1 Insert

To insert element with key p into the heap H,

- Increment $H.heap\text{-}size$ and add a new node with key p to the next available position
- Repeatedly swap the new item with its parent until the heap property is satisfied

  This swapping process is called **bubbling up**
- Worst-case runtime: $\Theta(\lg n)$

For example, consider $\textsc{Insert}(H, 17)$ where $H = [16, 8, 10, 1, 5, 3]$



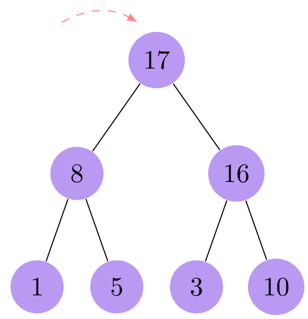
---

procedure $\textsc{Max-Heap-Insert}(H, p)$
    $i \leftarrow H.heap\text{-}size \leftarrow H.heap\text{-}size + 1$
    $H[i] = p$
    while $\textsc{Parent}(i) > 0$ **and** $H[i] > H[\textsc{Parent}(i)]$ **do**
        swap $H[i]$ with $H[\textsc{Parent}[i]]$
        $i \leftarrow \textsc{Parent}(i)$
    end while
end procedure

1.2.2 Find-Max

To find the maximum key in the heap H,

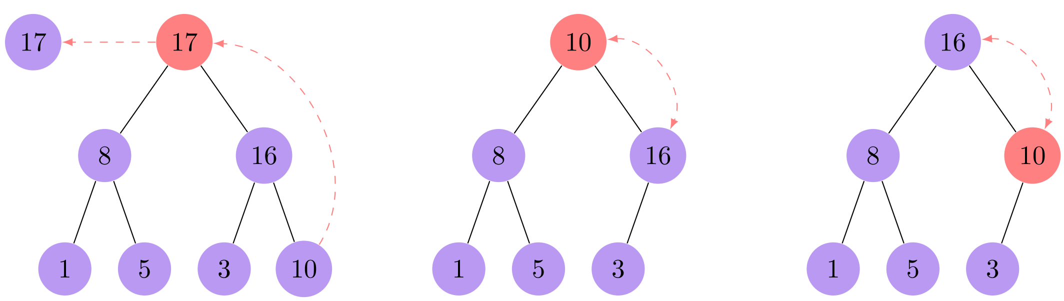
- Simply return the item in the root
- Worst-case runtime: $\Theta(1)$



procedure FIND-MAX(H)
    return $H[1]$
end procedure

1.2.3 EXTRACT-MAX

- Save the item from the root in a temporary variable
- Replace the root with the rightmost item in the lowest level of the tree and decrement $H.heap\text{-}size$
- Repeatedly swap the item we moved with its largest child until the heap property is restored. This swapping process is called **bubble down**.
- Worst-case runtime: $\Theta(\lg n)$



procedure EXTRACT-MAX(H)
    $max \leftarrow H[1]$
    $H[1] \leftarrow H[H.heap\text{-}size]$
    $H.heap\text{-}size \leftarrow H.heap\text{-}size - 1$
    MAX-HEAPIFY($H, 1$)
    return max
end procedure

```
procedure Max-Heapify(H, i)
    l ← Left(i)
    r ← Right(i)
    if l ≤ H.heap-size and H[l] > H[i] then
        largest ← l
    else
        largest ← i
    end if
    if r ≤ H.heap-size and H[r] > H[largest] then
        largest ← r
    end if
    if largest ≠ i then
        swap H[i] with H[largest]
        Max-Heapify(H, largest)
    end if
end procedure
```

1.2.4 Build-Max-Heap

- Takes an array A of length n and builds a max-heap H from it.
- Worst-case runtime: $\Theta(n)$

```
procedure Build-Max-Heap(A)
    H.heap-size ← A.length
    for i = ⌊A.length / 2⌋ downto 1 do
        Max-Heapify(H, i)
    end for
end procedure
```

1.3 Applications

1.3.1 Heap Sort

Implementation 1:

- Run Build-Max-Heap(A)
- Run Extract-Max(A) for $n-1$ times

Implementation 2:

- We modify the Extract-Max algorithm to swap the root with the last item

CHAPTER

DICTIONARY

2

Data

- A set S
- Each element x has a **unique** key $x.key$

Operations

- SEARCH(S, k)

  Return x in S with $x.key = k$ (or NIL).

- INSERT(S, x)

  Add x to S – if S contains y with $y.key = x.key$, then *replace* y with x.

- DELETE(S, x) [a]

  Remove element x from S.

[a] If we are given the key k instead of the element x, we could do DELETE$(S,$ SEARCH(S, k))

	SEARCH	INSERT	DELETE#
unsorted array	n	n	1
sorted* array	$\lg n$	n	n
unsorted linked list	n	n	1
sorted* linked list	n	n	1
direct access table	1	1	1
hash table	n	n	n
binary search tree	height of the BST	height of the BST	height of the BST
balanced search tree	$\lg n$	$\lg n$	$\lg n$

*: these require the keys to be ordered
#: here we are only doing deletion (without having to search for x)

2.1 Binary Search Tree

A binary search tree is a binary tree with the *binary-search-tree property*:
Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

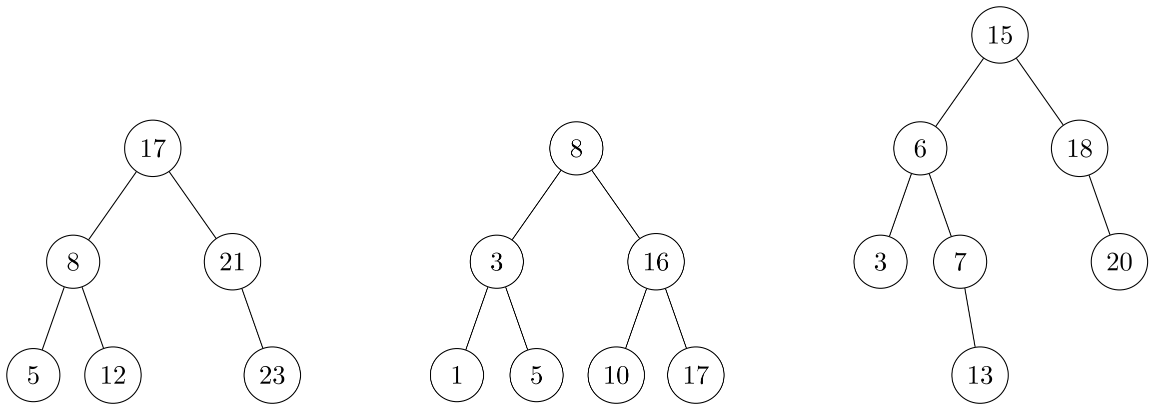


Figure 2.1: Examples of binary search trees

```
class Item:
    key: Any
    value: Any
```

```
class BST_Node:
    item: Item
    left: BST_Node
    right: BST_Node
```

```
class Dictionary:
    root: BST_Node
```

2.1.1 Insert

procedure INSERT(S, x)
    $S.root \leftarrow$ BST-INSERT($S.root, x$)
end procedure

procedure BST-INSERT($root$, x)
    # Insert x into stubree at root; return new root
    if $root =$ NIL **then**
        $root \leftarrow$ BST_NODE(x) # Add x
    else if $x.key < root.item.key$ **then**
        $root.left \leftarrow$ BST-INSERT($root.left, x$)
    else if $x.key > root.item.key$ **then**
        $root.right \leftarrow$ BST-INSERT($root.right, x$)
    else # $x.key = root.item.key$
        $root.item \leftarrow x$ # replace with x
    end if
    return $root$
end procedure

2.1.2 Search

```
procedure Search(S, k)
    node ← BST-Search(S.root, k)
    if node = NIL then
        return NIL
    end if
    return node.item
end procedure
```

```
procedure BST-Search(root, k)
    # Return node under root with key k (or NIL)
    if root = NIL then
        pass # k not in tree
    else if k < root.item.key then
        root ← BST-Search(root.left, k)
    else if k > root.item.key then
        root ← BST-Search(root.right, k)
    else  # k = root.item.key
        pass
    end if
    return root
end procedure
```

2.1.3 Delete

```
procedure Delete(S, k)
    S.root ← BST-Delete(S.root, k)
end procedure
```

```
procedure BST-Delete(root, x)
    # Delete x from stubree at root; return new root
    if root = NIL then pass # x not in tree
    else if x < root.item.key then
        root.left ← BST-Delete(root.left, x)
    else if x > root.item.key then
        root.right ← BST-Delete(root.right, x)
    else  # x.key = root.item.key
        if root.left = NIL then
            root ← root.right # could be NIL
        else if root.right = NIL then
            root ← root.left
        else  # Replace root.item with its successor
            root.item, root.right ← BST-Del-Min(root.right)
        end if
    end if
    return root
end procedure
```

```
procedure BST-Del-Min(root)
    # Remove element with smallest key under root; return item and root of resulting subtree
Require: root ≠ NIL
    if root.left = NIL then
        return root.item, root.right
    else
        item, root.left ← BST-Del-Min(root.left)
        return item, root
    end if
end procedure
```

2.2 Balanced Search tree

Despite the simplicity of the BST, it is not a very efficient data structure. The worst-case running time of the BST operations is proportional to the height of the tree, which is $\Theta(n)$, where n is the number of elements in the tree. The shape of a BST is determined by the order in which keys are inserted. If the keys are inserted in sorted order, the BST degenerates into a linked list.

We can improve the performance of the BST by making it more balanced. A *balanced BST* (also known as an *AVL tree* – Adelson-Velsky, Landis Tree) is one in which the heights of the two subtrees of any node differ by at most one. The height of a balanced BST is $\Theta(\log n)$, where n is the number of elements in the tree.

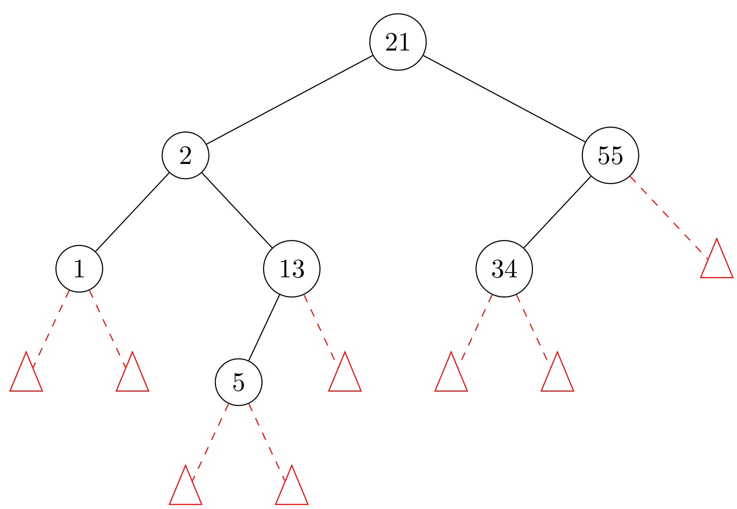


Figure 2.2: AVL balanced binary search tree

To implement an AVL tree, we need a mechanism to detect imbalance in the tree, and a way to restore balance. We will use the following definition of *balance factor* of a node x in a BST:

Definition 2.2.1 **Balance Factor**

An AVL balanced node x has a balance factor of -1, 0, or 1. If the height of its left subtree is h_L, and the height of its right subtree is h_R, then x has a balance factor of $h_L - h_R$.

- If $h_R - h_L = 0$, then x is balanced.
- If $h_R - h_L = 1$, then x is right-heavy.
- If $h_R - h_L = -1$, then x is left-heavy.

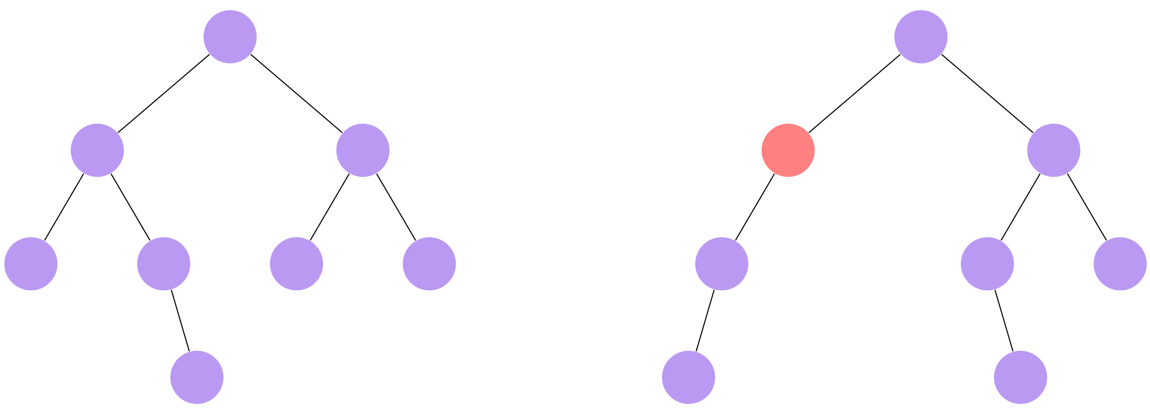


Figure 2.3: AVL balanced tree (left) and unbalanced tree (right)

Rotations

To restore balance, we need to perform a *rotation* on the tree. There are four types of rotations, depending on the balance factor of the node and its children. The following figure shows the four types of rotations.

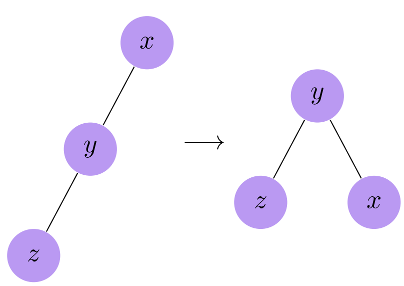


Figure 2.4: Single Left Rotation

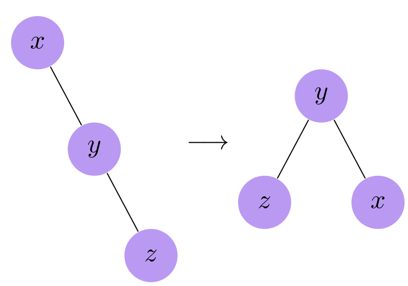


Figure 2.5: Single Left Rotation

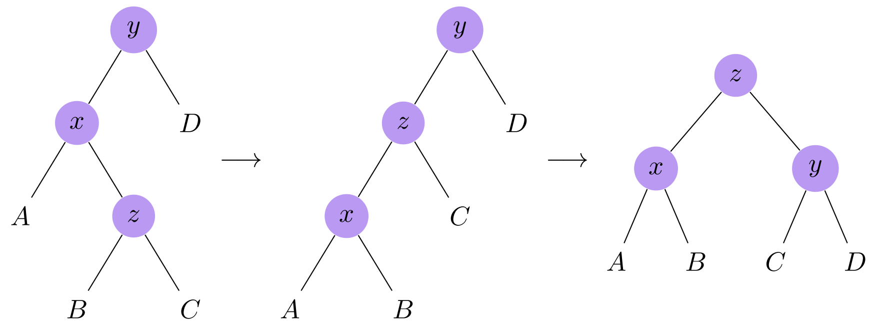


Figure 2.6: Double Left-Right Rotation

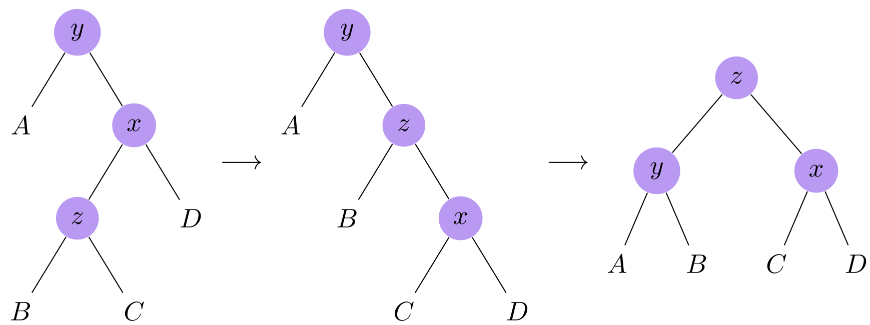


Figure 2.7: Double Right-Left Rotation

2.2.1 Insert

```
procedure AVL-Insert(root, x)
    # Insert x into the tree at root, return new root
    if root = NIL then
        root ← AVL_Node(x) # add x
    else if x.key < root.item.key then
        root.left ← AVL-Insert(root.left, x)
        root ← AVL-Balance-Right(root)
    else if x.key > root.item.key then
        root.right ← AVL-Insert(root.right, x)
        root ← AVL-Balance-Left(root)
    else# x.key = root.item.key
        root.item ← x # replace with x
    end if
    return root
end procedure
```

2.2.2 Delete

```
procedure AVL-Delete(root, x)
    # Delete x from the tree at root, return new root
    if root = NIL then
        pass # x not in tree
    else if x.key < root.item.key then
        root.left ← AVL-Delete(root.left, x)
        root ← AVL-Balance-Left(root)
    else if x.key > root.item.key then
        root.right ← AVL-Delete(root.right, x)
        root ← AVL-Balance-Right(root)
    else  # x.key = root.item.key
        if root.left = NIL then
            root ← root.right # could be NIL
        else if root.right = NIL then
            root ← root.left
        else
            if root.left.height > root.right.height then
                root.item, root.left ← AVL-Delete-Max(root.left)
            else
                root.item, root.right ← AVL-Delete-Min(root.right)
            end if
        end if
        root.height ← 1 + Max(root.left.height, root.right.height)
    end if
    return root
end procedure
```

```
procedure AVL-Del-Max(root)
    # Delete the maximum item from the tree at root, return new root and deleted item
Require: root ≠ NIL
    if root.right = NIL then
        return root.item, root.left
    else
        item, root.right ← AVL-Delete-Max(root.right)
        root ← AVL-Balance-Right(root)
        return item, root
    end if
end procedure
```

2.2.3 Rebalancing

```
procedure AVL-BALANCE-LEFT(root)
Require: root ≠ NIL
    # First, recalculate height
    root.height ← 1 + MAX(root.left.height, root.right.height)
    # Then, rebalance the left, if necessary
    if root.right.height > root.left.height + 1 then
        # Check for double rotation
        if root.right.left.height > root.right.right.height then
            root.right ← AVL-ROTATE-RIGHT(root.right)
        end if
        root ← AVL-ROTATE-LEFT(root)
    end if
    return root
end procedure
```

```
procedure AVL-ROTATE-LEFT(parent)
Require: parent ≠ NIL, parent.right ≠ NIL
    # Rearrange references
    child ← parent.right
    parent.right ← child.left
    child.left ← parent
    # Update heights; parent first because it is now deeper
    parent.height ← 1 + MAX(parent.left.height, parent.right.height)
    child.height ← 1 + MAX(child.left.height, child.right.height)
    # Return new parent
    return child
end procedure
```

2.3 Hashing

- Universe U

  The set of all keys. We assume that $|U|$ is very large.

- Hash Table T

  An array of fixed size m. Each location $T[i]$ is called a *bucket*.

- Hash Function h

  The hash function $h : U \to \{0, 1, \ldots, m-1\}$ maps each key in U to an index in $\{0, 1, \ldots, m-1\}$. For each key $k \in U$, $h(k)$ is called the *home bucket* of k.

  To access item with key k, examine $T[h(k)]$.

A hash table is an effective data structure for implementing dictionaries. Although SEARCH for an element in a hes table can take as long as searching for an element in a linked list – $\Theta(n)$ time in the worst case – in practice, hashing preforms extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $\mathcal{O}(1)$.

2.3.1 Direct Access Table

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. If U is small, then we can use an array T of size $|U|$ to implement a dictionary, called a *direct access table*. The key k is used as an index into T to access the item with key k.

2.3.2 Hash Table

The downside of direct addressing is apparent: if the universe U is large or infinite. storing a table T of size $|U|$ is impractical, and the set K of keys *actually stored* may be so small relative to Y that most of the space allocated for T would be wasted. Instead, we use a hash table.

However, when $m << |U|$, collisions are unavoidable. A *collision* occurs when two keys k_1 and k_2 (with $k_1 \neq k_2$) are mapped to the same bucket $h(k_1) = h(k_2)$. There are two ways to handle collisions: *open addressing* and *closed addressing / chaining*.

Open Addressing

In open addressing, if $T[h(k)]$ is occupied, then we search for the next available location in T to store the item with key k. We call the original hash function h_1 the *primary hash function*, such that $h_1(k)$ is the home bucket of k. We use the *probe sequence* $h(k, i)$ to determine the bucket to try after i collisions.

- *Linear Probing*

  $h(k, i) = (h_1(k) + i) \mod m$

  Note that long clusters of occupied buckets can occur.

- *Quadratic Probing*

  $h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \mod m$

  c_1 and c_2 are constants dependent on m.

- *Double Hashing*

  $h(k, i) = (h_i(k) + i \cdot (h_2(k))) \mod m$, where $h_2(k)$ is a secondary hash function.

2.3.3 Close Addressing / Chaining

In close addressing, we use a linked list to store the items in each bucket. Each nonempty slot points to a linked list, and all the elements that hash to the same slot go into that slot's linked list.

Part II

Analysis

CHAPTER

3

AVERAGE CASE ANALYSIS

CHAPTER 4

AMORTIZED ANALYSIS

4.1 Aggregated Method

In the *aggregated method*, we determine the upper bound $T(n)$ on the total cost of a sequence of N operations, then calculate the average cost per operation as $\frac{T(n)}{n}$.

4.2 Accounting Method

The *accounting method* is a form of aggregate analysis which assigns to each operation an amortized cost which may differ from its actual cost. Early operations have an amortized cost higher than their actual cost, which accumulates a saved "credit" that pays for later operations having an amortized cost lower than their actual cost. Because the credit begins at zero, the actual cost of a sequence of operations equals the amortized cost minus the accumulated credit. Because the credit is required to be non-negative, the amortized cost is an upper bound on the actual cost. Usually, many short-running operations accumulate such credit in small increments, while rare long-running operations decrease it drastically.

Part III

Appendices

BIBLIOGRAPHY

Books

INDEX