



CSC263

Data Structures and Analysis

SINAN LI

2023

CONTENTS

I Data Structure

5

1 | Chapter 1 Priority Queues and Heaps

- 1.1 Implementation 7
 - 1.1.1 Attempts 7
 - 1.1.2 Implementation 8
- 1.2 Operations 9
 - 1.2.1 INSERT 9
 - 1.2.2 FIND-MAX 9
 - 1.2.3 EXTRACT-MAX 10
 - 1.2.4 BUILD-MAX-HEAP 11
- 1.3 Applications 11
 - 1.3.1 Heap Sort 11

2 | Chapter 2 Dictionary

- 2.1 Binary Search Tree 14
 - 2.1.1 INSERT 14
 - 2.1.2 SEARCH 15
 - 2.1.3 DELETE 15
- 2.2 Balanced Search tree 16
 - 2.2.1 INSERT 18
 - 2.2.2 DELETE 19
 - 2.2.3 Rebalancing 20

2.3	Hashing	20
2.3.1	Direct Access Table	21
2.3.2	Hash Table	21
2.3.3	Close Addressing / Chaining	21

3 | Chapter 3 Dynamic Array

II Algorithms 25

4 | Chapter 4 Sorting

4.1	Heap Sort	27
4.2	Quick Sort	27
4.2.1	Deterministic Quick Sort	27
4.2.2	Randomized Quick Sort	28

5 | Chapter 5 Graph Algorithms

III Analysis 31

6 | Chapter 6 Average Case Analysis

7 | Chapter 7 Amortized Analysis

7.1	Aggregated Method	38
7.2	Accounting Method	38

IV Appendices 41

Part I

Data Structure

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.

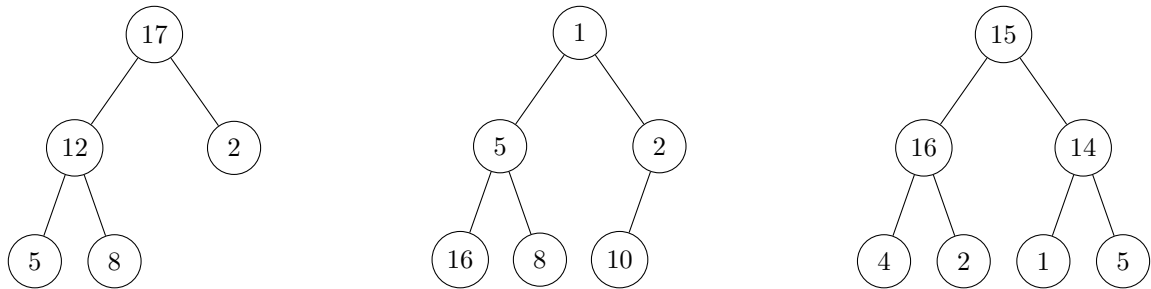


Figure 1.1: A valid max-heap (left), a valid min-heap (middle), and an invalid heap (right)

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 $H.heap-size$, 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 | • LEFT | • RIGHT |
| return $\lfloor i/2 \rfloor$ | return $2i$ | return $2i + 1$ |

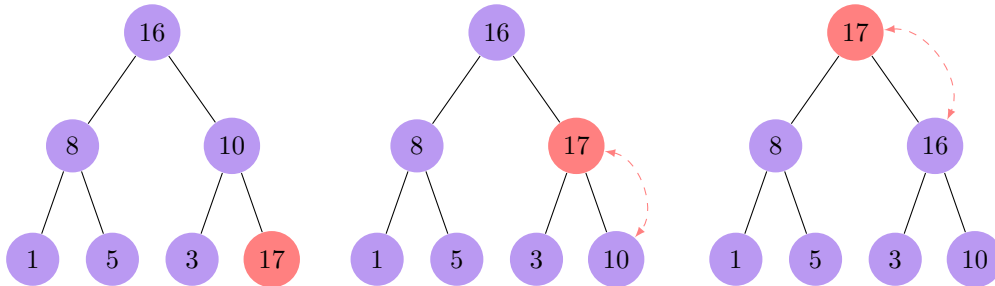
¹That is, the tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

1.2.1 INSERT

To insert element with key p into the heap H ,

- Increment $H.\text{heap-size}$ 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 $\text{INSERT}(H, 17)$ where $H = [16, 8, 10, 1, 5, 3]$



```

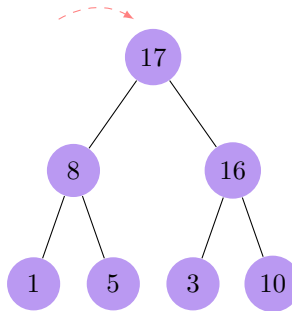
procedure MAX-HEAP-INSERT( $H, p$ )
   $i \leftarrow H.\text{heap-size} \leftarrow H.\text{heap-size} + 1$ 
   $H[i] = p$ 
  while PARENT( $i$ ) > 0 and  $H[i] > H[\text{PARENT}(i)]$  do
    swap  $H[i]$  with  $H[\text{PARENT}(i)]$ 
     $i \leftarrow \text{PARENT}(i)$ 
  end while
end procedure

```

1.2.2 FIND-MAX

To find the maximum key in the heap H ,

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



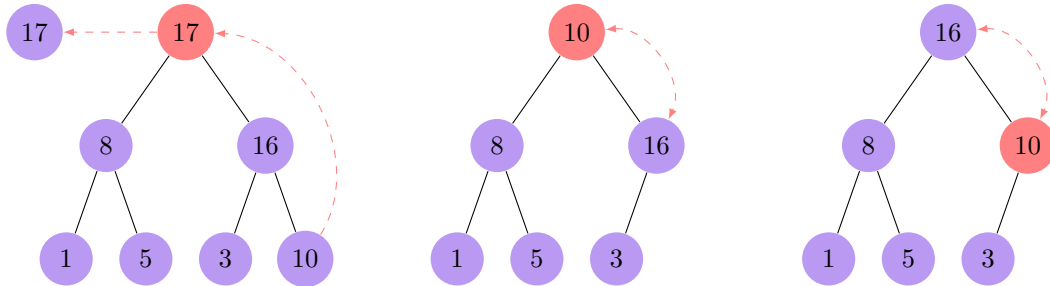
```

1: procedure FIND-MAX( $H$ )
2:   return  $H[1]$ 
3: 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: $\Theta(\lg n)$



```

1: procedure EXTRACT-MAX( $H$ )
2:    $max \leftarrow H[1]$ 
3:    $H[1] \leftarrow H[H.heap\text{-}size]$ 
4:    $H.heap\text{-}size \leftarrow H.heap\text{-}size - 1$ 
5:   MAX-HEAPIFY( $H, 1$ )
6:   return  $max$ 
7: end procedure

```

```

1: procedure MAX-HEAPIFY( $H, i$ )
2:    $l \leftarrow \text{LEFT}(i)$ 
3:    $r \leftarrow \text{RIGHT}(i)$ 
4:   if  $l \leq H.\text{heap-size}$  and  $H[l] > H[i]$  then
5:      $\text{largest} \leftarrow l$ 
6:   else
7:      $\text{largest} \leftarrow i$ 
8:   end if
9:   if  $r \leq H.\text{heap-size}$  and  $H[r] > H[\text{largest}]$  then
10:     $\text{largest} \leftarrow r$ 
11:  end if
12:  if  $\text{largest} \neq i$  then
13:    swap  $H[i]$  with  $H[\text{largest}]$ 
14:    MAX-HEAPIFY( $H, \text{largest}$ )
15:  end if
16: 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: $\Theta(n)$

```

1: procedure BUILD-MAX-HEAP( $A$ )
2:    $H.\text{heap-size} \leftarrow A.\text{length}$ 
3:   for  $i = \lfloor \frac{A.\text{length}}{2} \rfloor$  downto 1 do
4:     MAX-HEAPIFY( $H, i$ )
5:   end for
6: 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

DICTIONARY

Data

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

Operations

- $\text{SEARCH}(S, k)$
Return x in S with $x.key = k$ (or NIL).
- $\text{INSERT}(S, x)$
Add x to S – if S contains y with $y.key = x.key$, then *replace* y with x .
- $\text{DELETE}(S, x)$ ^a
Remove element x from S .

^aIf we are given the key k instead of the element x , we could do $\text{DELETE}(S, \text{SEARCH}(S, k))$

	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 $y.key \geq x.key$.

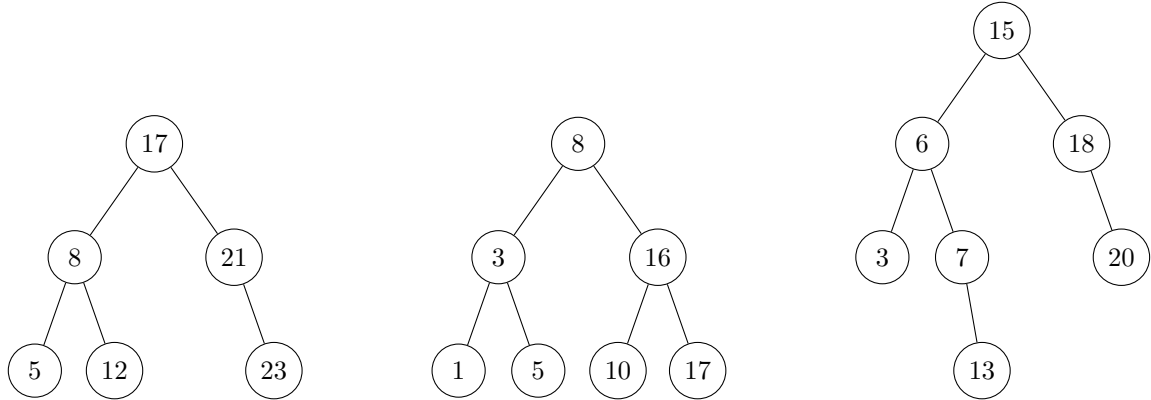


Figure 2.1: Examples of binary search trees

```
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

```
1: procedure INSERT( $S, x$ )
2:    $S.root \leftarrow$  BST-INSERT( $S.root, x$ )
3: end procedure
```

```
1: procedure BST-INSERT( $root, x$ )
2:   # Insert  $x$  into stubtree at  $root$ ; return new
   root
3:   if  $root = \text{NIL}$  then
4:      $root \leftarrow$  BST_NODE( $x$ ) # Add  $x$ 
5:   else if  $x.key < root.item.key$  then
6:      $root.left \leftarrow$  BST-INSERT( $root.left, x$ )
7:   else if  $x.key > root.item.key$  then
8:      $root.right \leftarrow$  BST-INSERT( $root.right, x$ )
9:   else #  $x.key = root.item.key$ 
10:     $root.item \leftarrow x$  # replace with  $x$ 
11:   end if
12:   return  $root$ 
13: end procedure
```

2.1.2 SEARCH

```
1: procedure SEARCH( $S, k$ )
2:    $node \leftarrow$  BST-SEARCH( $S.root, k$ )
3:   if  $node = \text{NIL}$  then
4:     return NIL
5:   end if
6:   return  $node.item$ 
7: end procedure
```

```
1: procedure BST-SEARCH( $root, k$ )
2:   # Return node under root with key  $k$  (or NIL)
3:   if  $root = \text{NIL}$  then
4:     pass #  $k$  not in tree
5:   else if  $k < root.item.key$  then
6:      $root \leftarrow$  BST-SEARCH( $root.left, k$ )
7:   else if  $k > root.item.key$  then
8:      $root \leftarrow$  BST-SEARCH( $root.right, k$ )
9:   else #  $k = root.item.key$ 
10:    pass
11:   end if
12:   return  $root$ 
13: end procedure
```

2.1.3 DELETE

```
1: procedure DELETE( $S, k$ )
2:    $S.root \leftarrow$  BST-DELETE( $S.root, k$ )
3: end procedure
```

```
1: procedure BST-DELETE( $root, x$ )
2:   # Delete  $x$  from stubree at root; return new root
3:   if  $root = \text{NIL}$  then pass #  $x$  not in tree
4:   else if  $x < root.item.key$  then
5:      $root.left \leftarrow$  BST-DELETE( $root.left, x$ )
6:   else if  $x > root.item.key$  then
7:      $root.right \leftarrow$  BST-DELETE( $root.right, x$ )
8:   else #  $x.key = root.item.key$ 
9:     if  $root.left = \text{NIL}$  then
10:       $root \leftarrow root.right$  # could be NIL
11:     else if  $root.right = \text{NIL}$  then
12:       $root \leftarrow root.left$ 
13:     else # Replace  $root.item$  with its successor
14:       $root.item, root.right \leftarrow$  BST-DEL-MIN( $root.right$ )
15:     end if
16:   end if
17:   return  $root$ 
18: end procedure
```

```

1: procedure BST-DEL-MIN(root)
2:   # Remove element with smallest key under root; return item and root of resulting subtree
Require: root  $\neq$  NIL
3:   if root.left = NIL then
4:     return root.item, root.right
5:   else
6:     item, root.left  $\leftarrow$  BST-DEL-MIN(root.left)
7:     return item, root
8:   end if
9: 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.

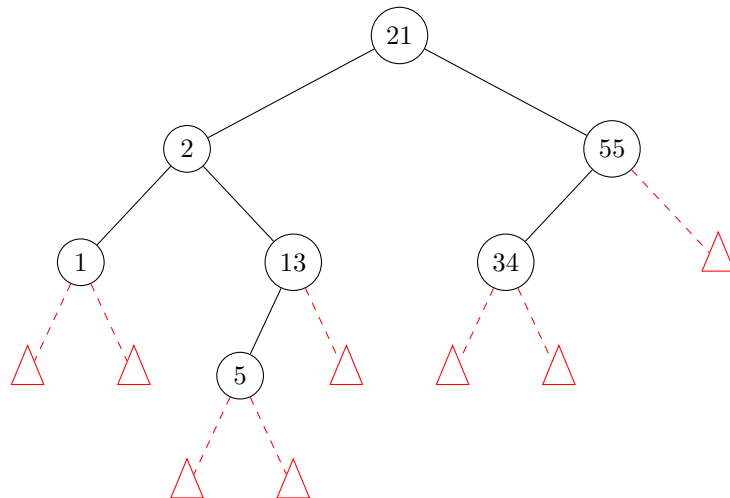


Figure 2.2: AVL balanced binary search tree

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.

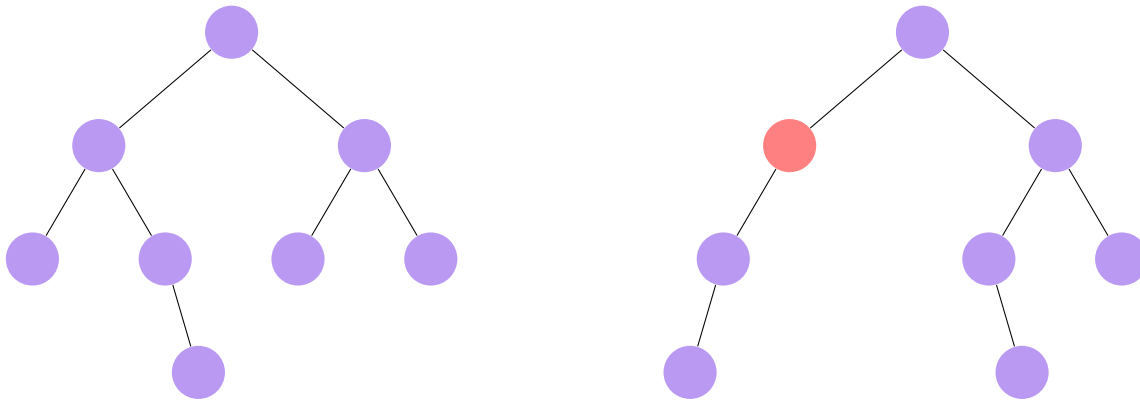


Figure 2.3: AVL balanced tree (left) and unbalanced tree (right)

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.

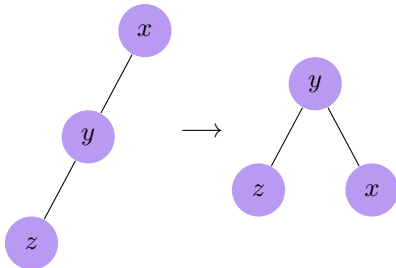


Figure 2.4: Single Left Rotation

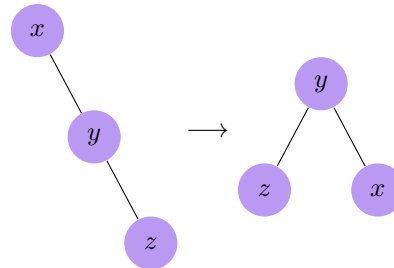


Figure 2.5: Single Left Rotation

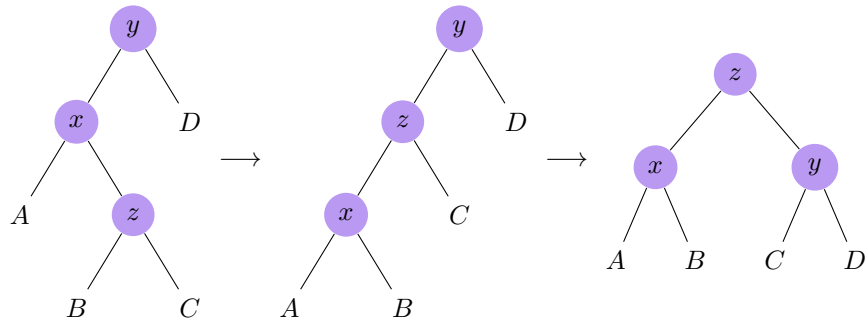


Figure 2.6: Double Left-Right Rotation

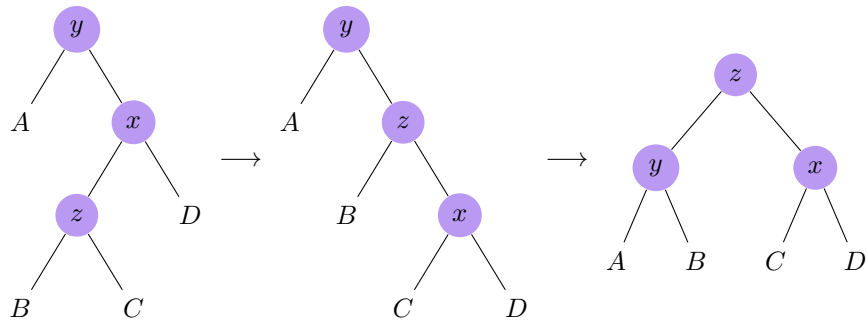


Figure 2.7: Double Right-Left Rotation

2.2.1 INSERT

```

1: procedure AVL-INSERT(root, x)
2:   # Insert x into the tree at root, return new root
3:   if root = NIL then
4:     root  $\leftarrow$  AVL_NODE(x) # add x
5:   else if x.key < root.item.key then
6:     root.left  $\leftarrow$  AVL-INSERT(root.left, x)
7:     root  $\leftarrow$  AVL-BALANCE-RIGHT(root)
8:   else if x.key > root.item.key then
9:     root.right  $\leftarrow$  AVL-INSERT(root.right, x)
10:    root  $\leftarrow$  AVL-BALANCE-LEFT(root)
11:  else # x.key = root.item.key
12:    root.item  $\leftarrow$  x # replace with x
13:  end if
14:  return root
15: 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

```
1: procedure AVL-BALANCE-LEFT(root)
Require: root ≠ NIL
2:   # First, recalculate height
3:   root.height ← 1 + MAX(root.left.height, root.right.height)
4:   # Then, rebalance the left, if necessary
5:   if root.right.height > root.left.height + 1 then
6:     # Check for double rotation
7:     if root.right.left.height > root.right.right.height then
8:       root.right ← AVL-ROTATE-RIGHT(root.right)
9:     end if
10:    root ← AVL-ROTATE-LEFT(root)
11:  end if
12:  return root
13: end procedure
```

```
1: procedure AVL-ROTATE-LEFT(parent)
Require: parent ≠ NIL, parent.right ≠ NIL
2:   # Rearrange references
3:   child ← parent.right
4:   parent.right ← child.left
5:   child.left ← parent
6:   # Update heights; parent first because it is now deeper
7:   parent.height ← 1 + MAX(parent.left.height, parent.right.height)
8:   child.height ← 1 + MAX(child.left.height, child.right.height)
9:   # Return new parent
10:  return child
11: 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 $h : U \rightarrow \{0, 1, \dots, m-1\}$ maps each key in U to an index in $\{0, 1, \dots, m-1\}$. 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 hash table can take as long as searching for an element in a linked list – $\Theta(n)$ time in the worst case – in practice, hashing performs 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 $m \ll |U|$, 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*

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

Note that long clusters of occupied buckets can occur.

- *Quadratic Probing*

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

c_1 and c_2 are constants dependent on m .

- *Double Hashing*

$$h(k, i) = (h_1(k) + i \cdot (h_2(k))) \bmod m, \text{ where } h_2(k) \text{ is a secondary hash function.}$$

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.

The average-case performance of the hash table depends on how evenly the hash function h distributes the keys across the buckets in the table. The *simple uniform hashing assumption* (SUHA)

states that any given key is equally likely to hash into any of the m slots of the table, independently of where any other elements has hashed to. Under this assumption, the expected number of keys in each bucket is the same.

The expected number of keys in a bucket is $\frac{n}{m}$, where n is the number of items in the table and m is the size of the table. This ratio is called the *load factor* of the hash table, and we denote it by α .

DYNAMIC ARRAY

C styled arrays are static, meaning that they have a fixed size. In this chapter, we will learn how to implement a dynamic array, which is a data structure that can grow and shrink in size.

```
class DynamicArray {
    capacity: integer # room for elements
    size:      integer # actual number of elements
};
```

```
1: procedure INSERT( $A, x$ )
2:   if  $A.size = A.capacity$  then
3:      $A \leftarrow \text{RESIZE}(A)$ 
4:   end if
5:    $A.size \leftarrow A.size + 1$ 
6:    $A[A.size] \leftarrow x$ 
7: end procedure
```

```
1: procedure RESIZE( $A$ )
2:    $B \leftarrow \text{DYNAMICARRAY}(2 \times A.capacity)$ 
3:   for  $i = 1$  to  $A.size$  do
4:     INSERT( $B, A[i]$ )
5:   end for
6:   return  $B$ 
7: end procedure
```

To analyze the running time of the above algorithm, see this example using accounting method for amortized analysis. The amortized running time of the above algorithm is $\Theta(1)$.

Part II

Algorithms

SORTING

4.1 Heap Sort

4.2 Quick Sort

```
1: procedure QUICKSORT( $A$ )
2:   if  $\text{LEN}(A) \leq 1$ 
3:     return  $A$ 
4:   end if
5:    $L, p, G \leftarrow \text{PARTITION}(A)$ 
6:   return QUICKSORT( $L$ ) +  $[p]$  + QUICKSORT( $G$ )
7: end procedure
```

4.2.1 Deterministic Quick Sort

In deterministic quick sort, we choose the pivot to be a certain index in the array. We can choose the pivot to be the first element, the last element, or the middle element.

- Run-time depends on input ordering
- Bad ordering would yield bad run-time, while random ordering would generally yield better run-time

4.2.2 Randomized Quick Sort

In randomized quick sort, we choose the pivot to be a random index in the array. Intuitively, the expected run-time would be the same as deterministic quick sort – $\Theta(n \lg n)$ – but the worst case run-time would be much better.

GRAPH ALGORITHMS

5

Part III

Analysis

AVERAGE CASE ANALYSIS

In *average case analysis*, we are interested in the average performance of an algorithm. we take the average, or the expected value, over the distribution of the possible inputs.

For each n , define $S_n = \{\text{all inputs of size } n\}$, and if we consider the inputs to be random, then S_n is the sample space. For each $x \in S_n$, define $P(x)$ to be the probability that x will be chosen as the input. Define $t(x)$ as the number of steps performed on input x . t is the random variable.

Then, the average case running time is defined as

$$\begin{aligned} T(n) &= E[t] \\ &= \sum_{x \in S_n} P(x) \cdot t(x) \end{aligned}$$

Example. Consider the linear search algorithm on a linked list L .

```

1: procedure LINSEARCH( $L, x$ )
2:    $z \leftarrow L.head$ 
3:   while  $z \neq \text{NIL}$  and  $z.data \neq x$  do
4:      $z \leftarrow z.next$ 
5:   end while
6:   return  $z$ 
7: end procedure

```

Let S_n be the sample space of all linked lists of size n . Let $P(x)$ be the probability that x is chosen as the input. Let $t(x)$ be the number of steps performed on input x .

- We need to know S_n with probability

Consider the inputs $\text{input}_1 = ([1, 2, 3], 2)$ and $\text{input}_2 = (["a", "b", "c"], "b")$, note that they will take the same steps. We only need one input for each possible value of t .

Define $S_n = \{([1, 2, \dots, n], 1), ([1, 2, \dots, n], 2), \dots, ([1, 2, \dots, n], n), ([1, 2, \dots, n], 0)\}$.

- We assume all the inputs happen equally likely, then $P(x) = \frac{1}{n+1}$.
- We need an exact formula for $t(x)$.

In practice, we choose some “key operations” s.t. counting **only** these operations is within a constant factor of total time – then set $t(x) = \text{number of key operations}$.

Here, we choose line 3, $z.data \neq x$, as the key operation.

$$\begin{aligned}
 \text{Then, we have } T(n) &= \sum_{(L,i) \in S_n} t(L,i) \cdot P(L,i) \\
 &= \frac{1}{n+1} \sum_{i=0}^n t([1, 2, \dots, n], i) \\
 &= \frac{1}{n+1} \left(t([1, 2, \dots, n], 0) + \sum_{i=1}^n i \right) \\
 &= \frac{1}{n+1} \left(n + \frac{n(n+1)}{2} \right) \\
 &= \frac{n}{n+1} + \frac{n}{2}
 \end{aligned}$$

Example. Consider $\text{SEARCH}(T, k)$ on a hash table T for a key k .

Assume t has m slots, and uses chaining to resolve collisions. Assume that prior to applying the `textscSearch` algorithm, the hash table contains n keys.

Assume the key k is samples uniformly at random from U .

Let $N(k)$ be the number of keys examined during search for k . $N(k)$ is the key operation.

$$\begin{aligned}
 E[N(k)] &= \sum_{k \in U} P[k] \cdot N(k) \\
 &= \sum_{i=0}^{m-1} \sum_{\substack{k \in U \\ h(k)=i}} P[k] \cdot N(k) && \text{regroup terms} \\
 &\leq \sum_{i=0}^{m-1} \sum_{\substack{k \in U \\ h(k)=i}} P[k] \cdot L_i && \text{since } N(k) \leq L_i \text{ when } h(k) = i \\
 &= \sum_{i=0}^{m-1} L_i \cdot \sum_{\substack{k \in U \\ h(k)=i}} P[k] \\
 &= \sum_{i=0}^{m-1} L_i \cdot P[h(k) = i]
 \end{aligned}$$

$$\begin{aligned}
&= \sum_{i=0}^{m-1} L_i \cdot \frac{1}{m} \\
&= \frac{1}{m} \sum_{i=0}^{m-1} L_i \\
&= \frac{n}{m}
\end{aligned}$$

Example. Consider QUICKSORT on an array A of size n .

Let $S_n = \{\text{all permutations of } [1, 2, \text{dots}, n]\}$

We assume an uniform distribution of the inputs, then $P(x) = \frac{1}{n!}$.

Let the random variable $T(A)$ be the total number of comparisons between elements of A .

Define $X_{i,j} = \begin{cases} 1 & \text{if } i \text{ is compared to } j \\ 0 & \text{otherwise} \end{cases}$ for $1 \leq i < j \leq n$.

Then, $T(A) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j}$.

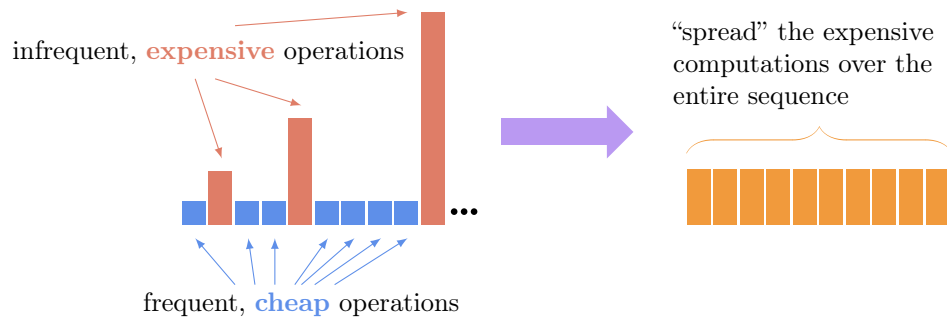
The probability $P(X_{i,j} = 1) = P(i \text{ or } j \text{ appear in } A \text{ before all other values in range } [i \dots j])$

$$\begin{aligned}
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}
\end{aligned}$$

$$\begin{aligned}
\text{Then, } E[T(A)] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{i,j}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^n P(X_{i,j} = 1) \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{\alpha+1} \quad \text{substitute } j-i \text{ with } \alpha \\
&< \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{k} \\
&= \sum_{i=1}^{n-1} \Theta(\lg n) \\
&= \Theta(n \lg n)
\end{aligned}$$

AMORTIZED ANALYSIS

In amortized analysis, we analyze the cost of a sequence of operations, not just a single operation. The amortized cost of a sequence of operations is the average cost per operation.



- The *worst-case sequence complexity* of a sequence S of k operations is the maximum possible total steps performed by S (taken over all possible inputs to the operations in S).
- The worst-case sequence complexity is **at most** $k \times$ the worst-case complexity of any individual operation in S .
- Suppose that the worst-case sequence complexity of a sequence of k operations is $T(k)$. Then the (worst-case) *amortized complexity* per operation of this sequence is $\frac{T(k)}{k}$.
- With amortized analysis, we take the average of the costs of multiple operations (as opposed to average-case analysis, where we calculate the cost of a single operation by averaging over the input distribution)

7.1

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

Example. Consider when we insert into an array. We increase the size of the array by 4 when it is $\frac{3}{4}$ full.

For simplicity, suppose that each insert with no resizing requires c steps, for some constant $c \in \mathbb{N}^+$ (so each insert with resizing requires $c \cdot n + c$ steps, where n is the number of items in the array prior to resizing).

Operation Number	Cost
1	c
2	c
3	c
4	$4c$
5	c
6	c
7	c
8	$7c$
9	c
10	$10c$
\vdots	\vdots

Within a sequence of k insertions, we need to resize $\lfloor \frac{k-1}{3} \rfloor$ times. For all k insert operations, the total cost is

$$T(k) = c \cdot \sum_{i=1}^{\lfloor \frac{k-1}{3} \rfloor} (3i+1) + c \cdot \left(k - \left\lfloor \frac{k-1}{3} \right\rfloor \right)$$

$$\text{Note that } c \cdot \sum_{i=1}^{\lfloor \frac{k-1}{3} \rfloor} (3i+1) = c \cdot \sum_{i=1}^{\lfloor \frac{k-1}{3} \rfloor} 3i + \sum_{i=1}^{\lfloor \frac{k-1}{3} \rfloor} 1$$

$$\begin{aligned} &= 3c \cdot \sum_{i=1}^{\lfloor \frac{k-1}{3} \rfloor} i + c \cdot \left\lfloor \frac{k-1}{3} \right\rfloor \\ &= \frac{3}{2}c \cdot \left\lfloor \frac{k-1}{3} \right\rfloor \cdot \left(\left\lfloor \frac{k-1}{3} \right\rfloor + 1 \right) + c \cdot \left\lfloor \frac{k-1}{3} \right\rfloor \in \Theta(k^2) \end{aligned}$$

Thus, $T(k)$ is $\Theta(k^2)$. The amortized cost per operation is $\frac{T(k)}{k} = \Theta(k)$.

7.2

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.

Example. Consider INSERT for dynamic array.

Part IV

Appendices

BIBLIOGRAPHY

Books

INDEX

- Amortized Analysis, 37
 - Accounting Method, 38
 - Aggregated Method, 38
 - Amortized Complexity, 37
 - Worst-Case Sequence Complexity, 37
- Average Case Analysis, 33
- AVL Tree, 16
 - Balance Factor, 17
 - Rotation, 17
- Balance Factor, 16
- Balanced Binary Search Tree, 16
- Binary Search Tree, 14
 - Binary-Search-Tree Property, 14
- Hash Table, 20
 - bucket, 20
 - Closed Addressing / Chaining, 21
 - collision, 21
 - Direct Access Table, 21
 - hash function, 20
 - home bucket, 20
 - load factor, 22
 - Open Addressing, 21
 - open addressing
 - double hashing, 21
 - linear probing, 21
 - quadratic probing, 21
 - primary hash function, 21
 - Probe Sequence, 21
 - Simple Uniform Hashing Assumption, 21
 - Universe, 20
- Heap, 8