COMPSCI 2C03
Data Structures & Algorithms

Gulkaran Singh

Fall 2023

Contents

1 Stacks

Stacks are linear data structures that arrange elements in sequential order. Here are some key properties:

- LIFO – Last In, First Out. This means the last element added to the stack is the **first** out.
- **push()** – adds an item to the TOP of the stack
- **pop()** – removes an item to the TOP of the stack

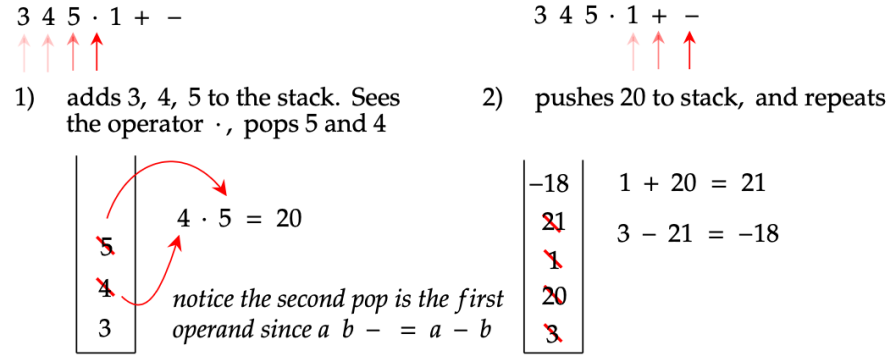
1.1 Postfix Expression Evaluation

Postfix expressions (e.g. 5 3 + turns into 5 + 3 which is infix) can be computed with stacks. The algorithm has the following steps:

- proceed left to right through the expression
- push operands (values) to the stack
- if an operator, pop twice and apply operator
- push the result back onto the stack

1.1.1 Example

Evaluate 3 4 5 · 1 + − (equivalent to (3 − ((4 · 5) + 1))



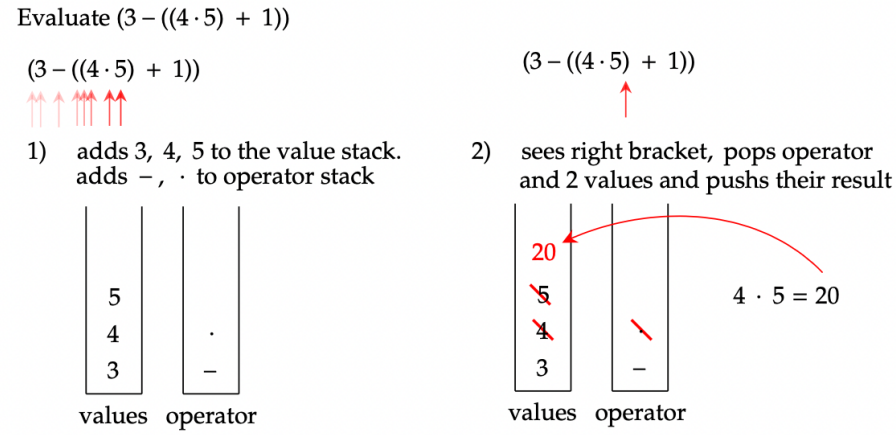
1.2 Dijkstra's Infix Evaluation

This a two-stack approach to solving infix (e.g. 5 + 3) expressions. Note this algorithm requires full and perfect bracketing. Here are the following steps for the algorithm:

- push operands onto value stack

- push operator onto operator stack
- if a right bracket, pop operator and two values
- push the result back into value stack

1.2.1 Example



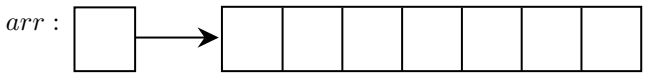
2 Queue

Queues are also linear data structures that arrange elements in sequential order. Here are some key properties:

- FIFO – First In, First Out. The first element in is the first element to leave. Think of queues as lines at the ATM!
- **enqueue()** – add an item to the BACK of the queue
- **dequeue()** – deletes an item at the FRONT of the queue
- elements can only LEAVE the way they entered. There is no variety unlike stacks.

3 Array Implementations

An array is an **indexable** linear data structure that has a **fixed length**. Note the memory is contiguous. In Java and C, arrays are auto-initialized to 'null' or zero.



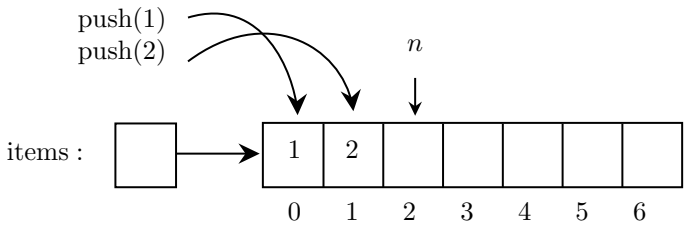
if arr[0] at 0x001, then arr[3] would be at :

$3 \cdot 4$ bytes (if int arr) $= 12$ bytes from arr[0]

3.1 Stack Implementation with Arrays

To implement a stack with a fixed-size array, we need the following:

- an array *items* of size *cap*
- variable n to track the number of items
- **push(item)** – adds item at location n, then $n{+}{+}$
- **pop()** – removes item from location $n-1$, then $n--$



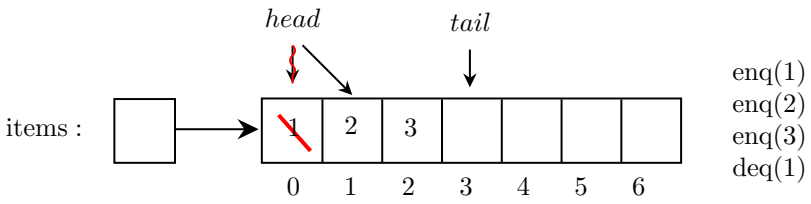
let n= # of items in stack & index of the next available

```
Stack(cap):
  items: array
  n: int <- 0
  size: int <- cap

push(item):
  if stack_full():
    raise error
  items[n] = item
  n++

pop():
  if stack_empty():
    raise error
  n--
  return items[n]
  # notice we don't need to remove items[n] since it will be
  ↪  overwritten if you push another item
```

3.2 Queue Implementation with Arrays

To implement a queue with a fixed-size array, we need the following:

- an array *items* of size *cap*
- variable *head* and *tail* to track the first and last item
- **enq(item)** – adds item at location *tail*, then $tail++$
- **deq()** – removes item at location *head*, then $head++$



This introduces a problem however. What do we do when our *tail* reaches the end of the array? For this, we introduce the 'circular array' implementation. We simply make the *head* and *tail* loop around to the beginning the array by using **modular arithmetic**!

$$head = (head + 1) \ \% \ \text{len}(items)$$
$$tail = (tail + 1) \ \% \ \text{len}(items)$$

```
enq(item):
  if queue_full():
    raise error
  items[tail] = item
  tail = (tail + 1) % len(items)

deq():
if queue_empty():
  raise error
head = (head + 1) % len(items)
```

4 Linked Lists

A linked list is a dynamically sized linear data structure. It includes the following:

- *head* – points to the **first** node in the linked list
- *tail* – points to the **last** node in the linked list
- *node.next* – points to the next node in the linked list

All three are references to Nodes and can be null if necessary. We typically have two special cases with linked lists:

- Singleton – linked list of length 1 → head == tail
- Empty – head == tail == null

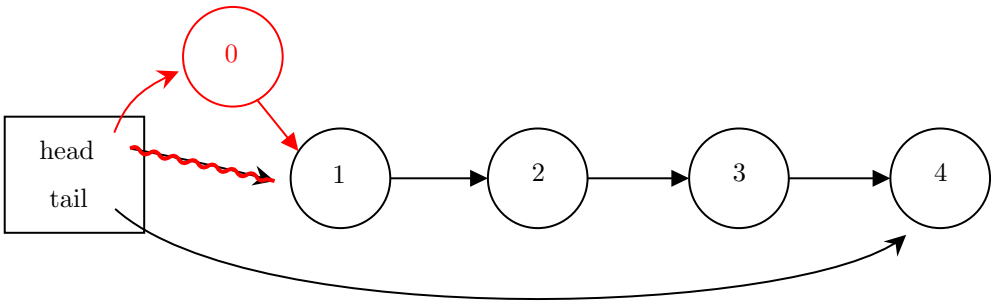
4.1 Basic Operations

Here are the most common examples of algorithms associated with the operations of linked lists.

4.1.1 Prepend

Adding a Node to the head of the linked list.

```
n = Node(item)
if isEmpty():     # empty
  head = n
  tail = n
else:             # non empty
  n.next = head
  head = n
```

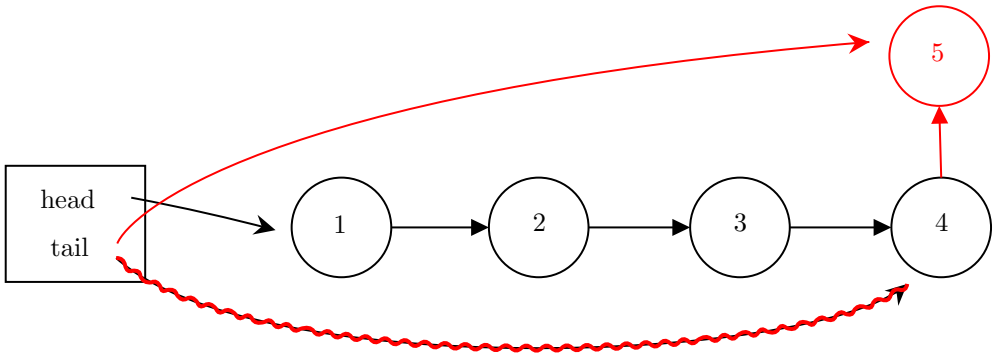


4.1.2 Append

Adding a Node to the end (tail) of the linked list.

```
n = Node(item)
if isEmpty():     # empty
  head = n
  tail = n
else:             # non empty
  n.tail.next = n
```

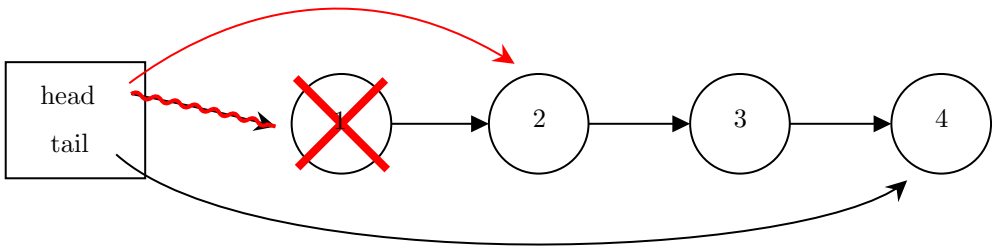
```
    tail = n
    n.next = null
```



4.1.3 Delete First

Deleting the head of the linked list, returning the first item

```
if head == null:      # empty
  raise error

if head == tail:      # singleton
  item = head.item
  head = null
  tail = null
  return item

item = head.item      # general
head = head.next
return item
```



4.1.4 Search

Traversing through the linked list to find if the key exists.

```
curr = head
while curr != null:
  if curr.item == key:
    return curr
  curr = curr.next
return null
```

4.2 Stack and Queue Implementations

- **Stack** – use prepend for push(), delete_first() for pop()
- **Queue** – use append for enqueue(), delete_last() for dequeue()

5 Algorithm Analysis

We denote $T(n)$ to be a time complexity function for an algorithm. It considers the number of basic instructions executed (assignments, operations, indexing, function calls, etc.)

$T(n)$ – worst case $\quad$ $T_{best}(n)$ – best case $\quad$ $T_{avg}(n)$ – avg case

5.1 Formal Definitions

Big-O: $T(n) \in O(f(n))$ if there exists constants $c > 0, n_0 \geq 0$ such that

$$0 \leq T(n) \leq c \cdot f(n)$$

for all $n \geq n_0$. Best approach is direct, find c, n_0, show $T(n) \geq 0$ and show $T(n) \leq c \cdot f(n)$

Big-Ω: $T(n) \in \Omega(f(n))$ if there exists constants $c > 0, n_0 \geq 0$ such that

$$T(n) \geq c \cdot f(n)$$

for all $n \geq n_0$. This definition is equivalent to

$$T(n) \in \Omega(f(n)) \qquad T(n) \notin O(f(n)) \qquad f(n) \in O(T(n))$$

Big-Θ: $T(n) \in \Theta(f(n))$ if and only if

$$T(n) \in O(f(n)) \wedge T(n) \in \Omega(f(n))$$

for all $n \geq n_0$.

6 Working with Big-O

6.1 Complexity Classes

Here are the most common complexities and their names.

Name	Big-O	Example
Constant	$O(1)$	Indexing into an array
Logarithmic	$O(\log n)$	Binary search
Linear	$O(n)$	Searching a linked list
Linearithmic	$O(n \log n)$	Merge sort
Quadratic	$O(n^2)$	Selection sort
Cubic	$O(n^3)$	Shortest path for graphs
Exponential	$O(2^n)$	n-bit cryptographic key
Factorial	$O(n!)$	Travelling Salesperson

6.2 Limits for Complexity

We can use an alternative proof based on the definitions,

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} \geq 0 \text{ and } \neq \infty \iff T(n) \in O(f(n))$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} > 0 \iff T(n) \in \Omega(f(n))$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} > 0 \text{ and } \neq \infty \iff T(n) \in \Theta(f(n))$$

7 Selection Sort

Selection sort **selects** the smallest element in the array, and swaps it into the front 'sorted' portion.

```
def selection_sort(arr, n):
  for pos = 0 to n - 1:
    best = pos
    for i = pos + 1 to n - 1:
      if arr[i] < arr[best]:
        best = i

    swap arr[best], arr[pos]
```

This effectively finds the index of the smallest element (best) and swaps it to the front (pos).

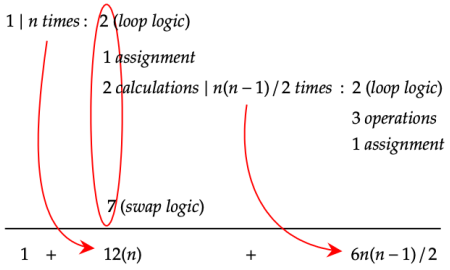
7.1 Linked List Implementation

We can easily translate this code into a linked list implementation.

```
def selection_sort(list):

  if list_empty(list): return

  pos = list.head
  while pos.next != null:
    best = pos
    curr = pos.next
    while curr != null:
      if curr.item < best.item:
        best = curr
      curr = curr.next

    swap best.item, curr.item
    pos = pos.next
```

7.2 Time Complexity

```
def selection_sort(arr, n):
  for pos = 0 to n - 1:
    best = pos
    for i = pos + 1 to n - 1:
      if arr[i] < arr[best]:
        best = i

    swap arr[best], arr[pos]
```



$$\therefore T(n) = 3n^2 + 9n + 1$$
$$T(n) \in O(n^2)$$

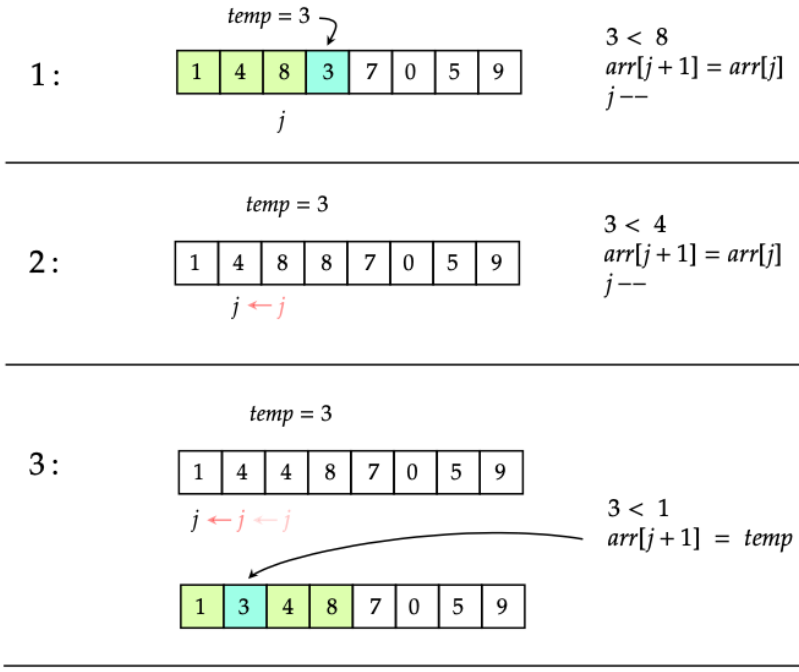
8 Insertion Sort

Insertion sort inserts the next element in the correct spot within the already sorted portion of the array.

```
def insertion_sort(arr, n):
  for next = 1 to n - 1:
    temp = arr[next]
    j = next - 1
```

```
    while j >= 0 and temp < arr[j]:
      arr[j + 1] = arr[j]  # move element one to the right
      ↪  to make space
      j = j - 1
    arr[j + 1] = temp
```

8.1 Example Illustration



8.2 Time Complexity

Worst Case: $T(n) \in O(n^2)$

Best Case: $T(n) \in O(n)$

- this occurs when the array is already sorted, so we don't ever enter the while loop and check the sorted portion of the array.

9 Merge Sort

Merge sort is a divide and conquer recursive algorithm that

- divides the array in two halves

- recursively sorts the two halves
- merge the two halves

9.1 General Algorithm

```
def msort(arr, lo, hi):
  if hi <= lo:
    return

  mid = lo + (hi-lo) // 2
  msort(arr, lo, mid)
  msort(arr, mid + 1, hi)
  merge(a, lo, mid, hi)
```

9.2 Merge Algorithm

The general description of this algorithm is you move left to right through both halves, comparing each element from the halves.

We have two pointers i, j. They start as the first element of each half.

```
i = lo
j = mid + 1
```

We then copy everything to an auxiliary array. This aux array doesn't get sorted, it's just so we have access to the elements after we merge.

```
for k <- lo to hi:
  aux[k] = arr[k]
```

Now we can start the main algorithm. We start with checking if we reach the end of either subarray. With this case, we can copy all the elements of the other half as it will already be sorted!

Otherwise, we simply compare the elements at index i, j and merge!

```
for k <- lo to hi:
  if i > mid:         # end of left half
    arr[k] = aux[j]
    j = j + 1

  else if j > hi:     # end of right half
    arr[k] = aux[i]
    i = i + 1
```

```
    else if aux[i] < aux[j]:  # element i is smaller
      arr[k] = aux[i]
      i = i + 1

    else:                # element j is smaller
      arr[k] = aux[j]
      j = j + 1
```

9.3 Time Complexity

We can use a recursion tree to analyze the time complexity of recursive algorithms.

- nodes are the size of the sub-problem
- sum the amount of work at each problem

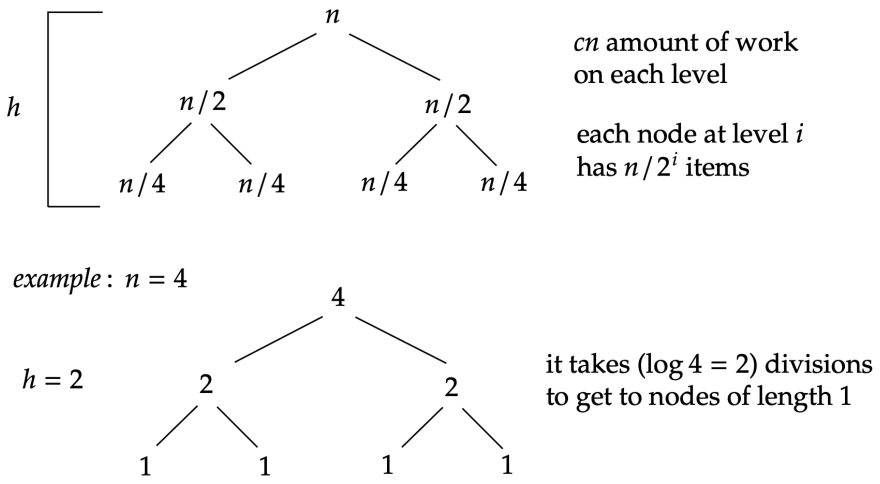
9.3.1 Case 1: n is a power of 2

There are two things to account for the time complexity. One is the work done by the merge algorithm, and the work done by the dividing the array in 2.

- each level of the tree has cn operations for the merge algorithm.
- each node at level i, contains $\frac{n}{2^i}$ items in the array, assuming i starts at 0.
- $\therefore$ for the height of the tree, the number of divisions to get the items at level h to equal 1 is represented by

$$\frac{n}{2^h} = 1$$
$$\log(n) = h$$

  This is simply because we keep dividing the array by 2 until there is 1 array item in each node, then we start our merging process. So we assume h is the height where nodes are of size 1, which would take $\log n$ amount of work to get to.

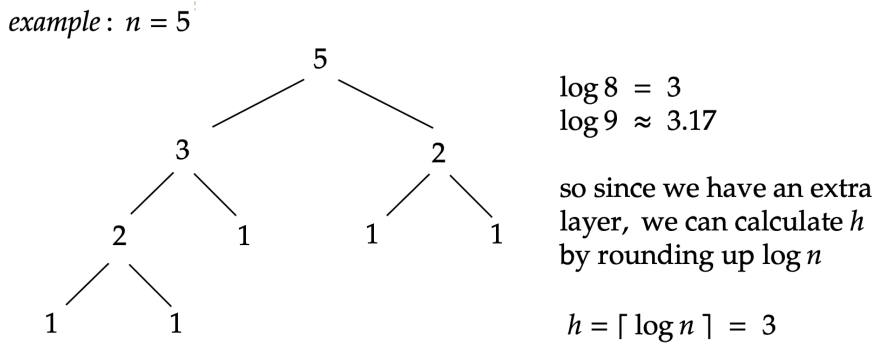


So for the best case where n is a power of 2, $T(n) = cn \log(n) \in O(n \log n)$

9.3.2 Case 2: n is not a power of 2

This occurs when the left split is bigger than the right. This adds only 1 layer at most. If n is not a power of 2, when we do $\log n = h$ we get a decimal. For example, $\log 8 = 3$ but $\log 9 = 3.17$. Since this will add one more layer to our tree, we can use the ceiling function!

$$T(n) \leq cn\lceil \log n \rceil \leq c(n \log n + 1) \in O(n \log n)$$



9.3.3 Recurrence Relations

We can also model $T(n)$ as a recurrence relation. We start with the base cases:

$$T(0) = T(1) \cong c$$

where the constant c is the **constant** amount of work if the array was empty or of size 1.

For the recursive cases, since we are recursively calling **msort()** twice where we divide the array length by 2 $(n/2)$, we can represent $T(n)$ as

$$T(n) = T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + cn$$

where cn is the work done by the merge algorithm! Note we use the ceiling and floor functions here in case n is not even.

9.4 Space Complexity

- cn for original array
- cn for auxiliary array
- $d \log n$ for call stack

Note, the d represents constant factors like space taken by variables, etc. Therefore, $S(n) \approxeq 2cn + d \log n \in O(n)$

10 Quicksort

Quicksort is a divide and conquer recursive algorithm like mergesort. It sorts in place, so there is no need for an auxiliary array.

- partition the array around a **pivot** element
- left side of the pivot are less than the pivot
- right side is larger than the pivot
- this ensures the pivot is in the correct place within the array
- recursively sort the left and right sides

Note: we don't introduce a way to choose the pivot, but there are ways to choose the best pivot (not talked ab in this class).

10.1 General Structure

```
def qsort(arr, lo, hi):
  if hi <= lo:
    return
  pivot_index = partition(arr, lo, hi)
  qsort(arr, lo, pivot_index - 1) # left side
  qsort(arr, pivot_index + 1, hi) # right side
```

10.2 Partition Algorithm

This algorithm takes the pivot and 'puts' it in the right place in the array, ensuring the left side is smaller than the pivot, and the right side is larger.

We start with pointers i, j. This assumes our pivot is the first element.

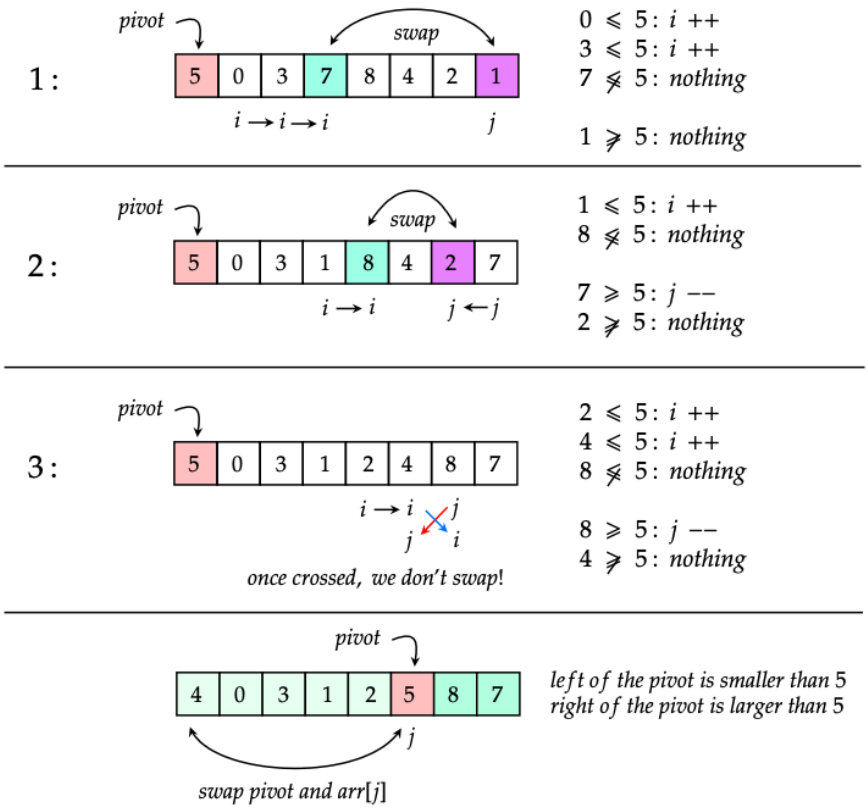
```
i = lo + 1
j = hi
pivot = arr[lo]
```

In our loop, we increment i whenever $arr[i]$ is **SMALLER** than the pivot as we want to keep these items on the left side. We stop if its larger. Similarly, we decrement j whenever $arr[j]$ is **LARGER** than the pivot, and stop when its smaller. Once we've stopped both inner loops, we can swap $arr[i]$ and $arr[j]$. This essentially brings items larger to the right side, and smaller to the left side.

```
loop:
  while arr[i] <= pivot:
    i = i + 1
    if i == hi:
      break

  while arr[j] >= pivot:
    j = j - 1
    if j == lo:
      break

  if i >= j:
    break (infinite loop)

  swap(arr[i], arr[j])

swap arr[lo], arr[j]
return j
```

Once the pointers i, j cross, we no longer swap any items. We can swap the pivot into the index of j and return.

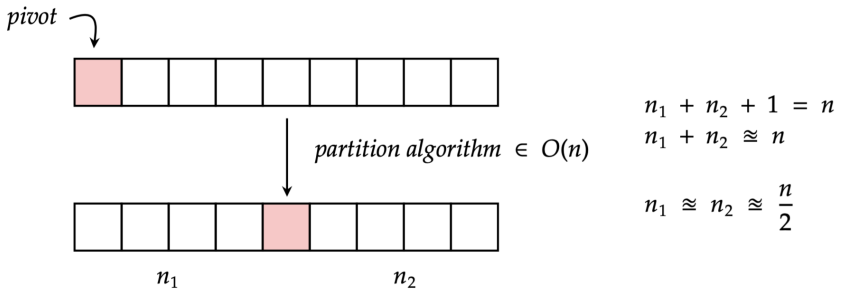
10.3 Example Drawn Out



10.4 Time Complexity

10.4.1 Best Case

The best case is when every partition is even so the 2 subarrays are of same length.



If $n_1 \approxeq n_2 \approxeq n/2$,

$$T(n) = T(n_1) + T(n_2) + cn$$
$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + cn$$
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + cn$$

Notice how this is the same as merge sort! So we can say $T(n) = 2 \cdot T(\frac{n}{2}) + cn \in O(n \log n)$.

10.4.2 Worst Case

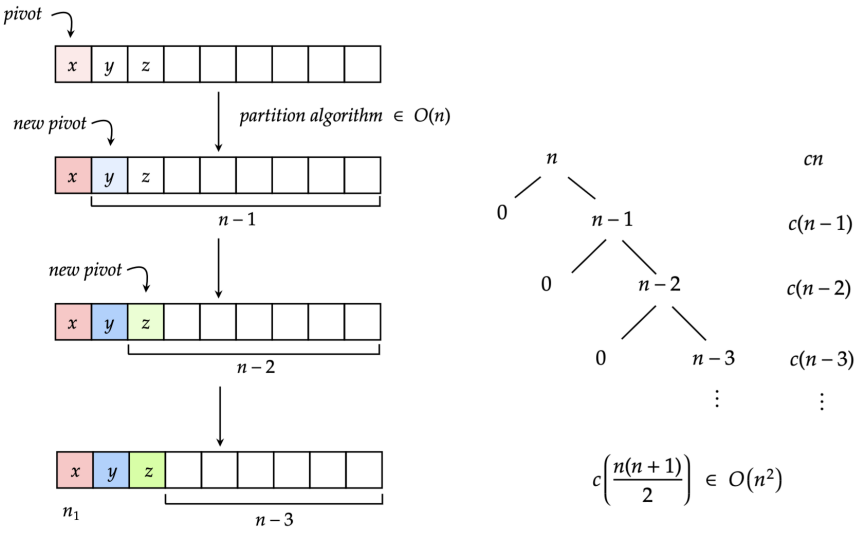
The worst case happens when the array is already sorted or when our pivot is consistently chosen as the smallest element. After every partition, the pivot stays where it is since it is the smallest element of the subarray. This makes our recurrence relation the following,

$$T(n) = T(0) + T(n-1) + cn$$
$$T(n) = T(n-1) + cn$$

Going through the entire tree we can see that,

$$T(n) = c\left(\frac{n(n+1)}{2}\right)$$

which means $T(n) = c\left(\frac{n(n+1)}{2}\right) \in O(n^2)$.



10.4.3 Almost Worst Case

Say the partition always splits the array 90% on one side, 10% on the other. Here $n_1 = \frac{n}{10}$ and $n_2 = \frac{9n}{10}$. We can model the recurrence relation as

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn$$

Modeling this as a tree we can see,

On the left side (10%), we can find h with the following,

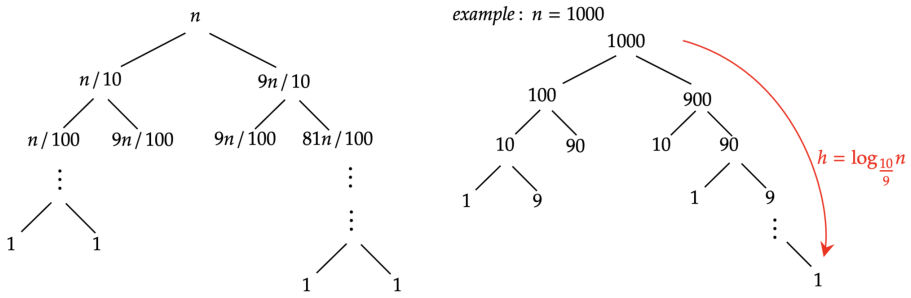
$$\frac{n}{10^h} = 1$$
$$n = 10^h$$
$$\log n = h \log 10$$
$$\frac{\log_2 n}{\log_2 10} = h$$
$$\log_{10} n = h$$

However, the right side (90%) is always deeper, so we calculate that with

$$\frac{9n}{10^h} = 1$$
$$\log 9n = h \log 10$$
$$\frac{\log_2 9n}{\log_2 10} = h$$
$$\log_{\frac{10}{9}} n = h$$

Therefore,

$$T(n) \leq cn \log_{\frac{10}{9}} n \in O(n \log n)$$



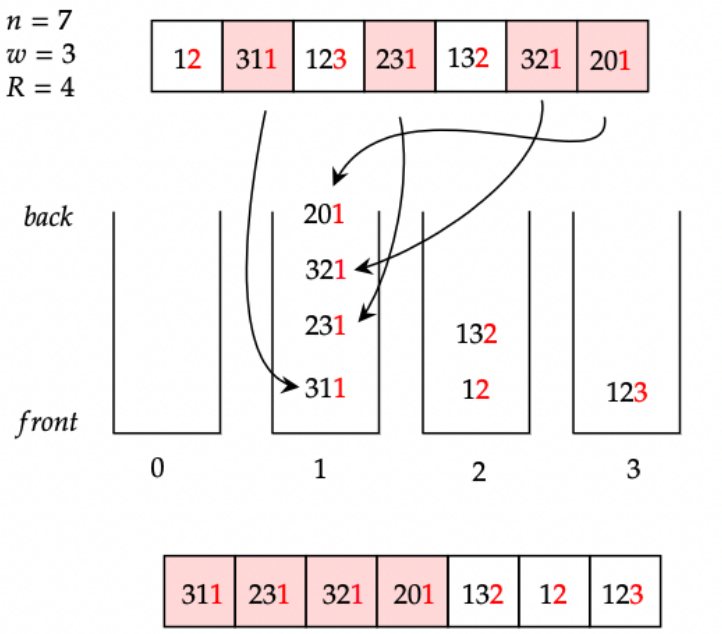
11 Radix Sort

Radix sort compares the least significant digits (LSD) of the numbers, and sorts based on that. For example, [19, 20, 5] would sort 9, 0, 5 on the first pass.

- n integers (size n)
- w digits in the integer
- R – radix, more commonly known as the base of the digit (e.g. decimal is radix 10, binary is radix 2)

11.1 Algorithm Structure

We start by creating an array of R 'buckets' which are queues. We use queues to maintain stability. For each item in the array, we look at its LSD and put it in its respective bucket. Once all the numbers are in their buckets, we dequeue them in order and repeat.



This is what the algorithm looks like in code:

```
def radix_sort(arr, R, w):
  buckets = array(R) of Queues # array of R number of queues

  for p = 0 to w - 1:  # for each sig dig

    # putting element into corresponding bucket
    for item in arr:
      d = item % R^(p+1) // R^p
      enqueue(buckets[d], item)

    # taking element out and sorting
    i = 0
    for bucket in buckets:
      while not is_empty(bucket):
        a[i] = dequeue(bucket)
        i = i + 1
```

The logic behind $d = item \ \% \ R^{p+1} \ // \ R^p$ is shown with an example:

$$\begin{aligned} R = 10, w = 4, p = 1 \\ item = 3758 \\ d = item \ \% \ R^{p+1} \ // \ R^p \\ d = 3758 \ \% \ 10^2 \ // \ 10^1 \\ d = 3758 \ \% \ 100 \ // \ 10 \\ d = 58 \ // \ 10 \\ d = 5 \end{aligned}$$

11.2 Changing Radix Base R

To use a different number of buckets, we can use the following:

- max number in radix $R = R_1^{w_1} - 1$
- $w_2 = \lceil \log_{R_2}(\max) \rceil = \lceil \frac{\log \max}{\log R_2} \rceil$

Here is how we derived w_2

$$\begin{aligned} R_2^{w_2} &= 9999 \\ w_2 \log R_2 &= \log 9999 \\ w_2 &= \log 9999 / \log R_2 + 1 \end{aligned}$$

11.2.1 Example

Change $R_1 = 10, w_1 = 4$ to $R_2 = 8$,

$$\begin{aligned} \max &= R_1^{w_1} - 1 \\ \max &= 10^4 - 1 = 9999 \\ w_2 &= \lceil \log_8(9999) \rceil = 5 \end{aligned}$$

11.3 Time Complexity

For radix sort $T_{best}(n) = T_{avg}(n) = T(n)$. It is based on 4 things:

- creating R buckets ($c_2 R$)
- w outer loops for each digit
- n items going into the buckets ($c_1 n$)
- n items removed from the buckets ($c_1 n$)

Therefore our time complexity looks like the following function:

$$\begin{aligned} T(n) &\approxeq (c_1 n + c_1 n) \cdot w + c_2 \cdot R \\ T(n) &\approxeq 2c_1 nw + c_2 R \end{aligned}$$

If R is held constant, then $T(n) \in O(n)$. However, note that as $w \to n$, $T(n) \to O(n^2)$.

12 Questions

Here are the most common types of questions and the method to solve them!

12.1 Algorithm Analysis

If you have to prove $T(n) \in O(f(n))$, you can use the definition, bump terms up, and prove. For **proving by definition**, you need a pre-proof analysis and a formal proof:

- **pre-proof analysis**: finding values for c, n_0
- **formal proof**: with values for c, n_0, work backwards to show $T(n) \leq c \cdot f(n)$

12.1.1 Example:

Prove $35n^3 - 150n + 1 \in O(n^3)$ Start with the pre-proof analysis. By definition,

$$35n^3 - 150n + 1 \in O(n^3)$$
$$35n^3 - 150n + 1 \leq 36n^3 \text{ for all } n \geq 2$$
$$c = 36, n_0 = 2$$

Now for the formal proof. Choose $c = 36, n_0 = 2$

$$0 \leq 35n^3 - 150n + 1 \text{ for all } n \geq 2$$
$$-150n + 1 \leq n$$
$$n \leq n^3$$
$$35n^3 - 150n + 1 \leq 35n^3 + (n^3)$$
$$35n^3 - 150n + 1 \leq 36n^3 = cn^3$$
$$\therefore 35n^3 - 150n + 1 \in O(n^3)$$

If you have a question asking to prove $T(n) \in \Omega(f(n))$ or $T(n) \notin O(f(n))$, it is best to use limits. For $T(n) \notin O(f(n))$, you want to use proof by contradiction and show that $\infty > c$, otherwise for $T(n) \notin \Omega(f(n))$, you show the limit is > 0.

12.1.2 Example:

Prove $20n^2 - 33n - 22 \in \Omega(n^2)$.

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} > 0$$
$$\lim_{n\to\infty} \left(\frac{20n^2 - 33n - 22}{n^2} \right) > 0$$
$$\lim_{n\to\infty} \left(\frac{n^2(20 - \frac{33}{n} - \frac{22}{n^2})}{n^2} \right) > 0$$
$$\lim_{n\to\infty} 20 > 0$$

Note if this was $T(n) \notin O(f(n))$, we would use the formal definition + limits:

$$20n^2 - 33n - 22 \leq cn^2$$
$$\lim_{n\to\infty} \left(\frac{20n^2 - 33n - 22}{n^2} \right) > c$$
$$20 > c$$

Therefore by contradiction, we can prove n^3 is not bounded above $T(n)$ for $c < 20$.

What if the question asks to prove $T(n) \in \Omega(f(n))$ by definition? There are two cases, one where the degree of $T(n)$ is the same as $f(n)$ like in our example above. Here is the formal proof for that. We can assume $c = 19$ (sort of cheating since we know $c < 20$ for it to be bounded below).

$$-33n - 22 \geq -(n^2) \text{ for all } n \geq 34$$
$$20n^2 - 33n - 22 \geq 20n^2 - n^2$$
$$20n^2 - 33n - 22 \geq 19n^2 = cn^2$$
$$20n^2 - 33n - 22 \in \Omega(n^2)$$

Methods for proving $T(n) \notin \Omega(f(n))$ are similar to $T(n) \notin O(f(n))$, however, with different powers we use the same method as before.

12.1.3 Example:

Prove $11n^2 - 43 \notin \Omega(n^3)$. Here we are saying n^3 is not bounded below $11n^2 - 43$. We can do 2 methods:

$$T(n) < cn^3$$
$$11n^2 - 43 < 11n^2$$
$$11n^2 < 11n^3 \text{ for all} n \geq 2$$
$$11n^2 - 43 < 11n^3$$
$$\therefore c = 11, n_0 = 2$$

and continue with the formal proof. Here is how the professor does it,

$$11n^2 - 43 < 11n^2$$
$$11n^2 < cn^3$$
$$\frac{11}{c} < n$$

Formal Proof: For every $c > 0, n_0 \geq 0$, if we choose an n such that $n \geq n_0, n \geq 1, n \geq \frac{11}{c}$

$$n > \frac{11}{c}$$
$$cn^3 > 11n^2$$
$$11n^2 - 43 < 11n^2$$
$$(11n^2 - 43 < 11n^2) \wedge (11n^2 < cn^3)$$
$$11n^2 - 43 < cn^3$$
Theorem: Transitivity of $<$

Since c and n_0 are arbitrarily chosen, it is proven.

12.1.4 Example:

Prove $n^2 + n - 555 \notin O(n)$. Here we are saying n is not bounded above $n^2 + n - 555$. To prove this via definition, we use the professors method:

Pre-proof analysis: We need to find an n to show $n^2 + n - 555 > cn$.

$$n^2 + n - 555 > n^2 \text{ for any } n \geq 555$$

because its easy to show that

$$n^2 > cn$$

when $n > c$.

Formal Proof: Using the definition of $T(n) \notin O(n)$, we say for every $c > 0, n_0 \geq 0$, we choose an n such that $n \geq n_0, n \geq 555, n > c$.

$$n > c$$
$$n^2 > cn$$
$$n^2 + n - 555 > n^2$$

Since $n^2 + n - 555 > n^2$ and $n^2 > cn$, we can say

$$(n^2 + n - 555 > n^2) \wedge (n^2 > cn)$$
$$n^2 + n - 555 > cn$$
$$n^2 + n - 555 \notin O(n)$$

which is the theorem of transitivity! Since c, n_0 are arbitrarily chosen, it is proven.

12.1.5 Summary

- if same degree: $\in O(f(n))$ and $\in \Omega(f(n))$ bump up/down terms
- if not: $\notin O(f(n))$ and $\notin \Omega(f(n))$ use transitivity.

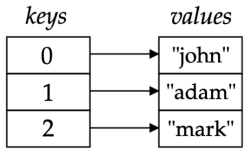
13 Tables and Searching

A **symbol table** is a data structure that associates **keys** with **values**. Think of them like dictionaries in Python.

- keys must be **unique** and **not null**. If a key already exists, it will simply overwrite the value.
- values must be **not null**
- deletion means **removing the key** since values cannot be null.
  - **eager deletion** – physically removing the memory
  - **lazy deletion** – puts the value of the key as null

SymbolTable ADT

create_st()	returns a new **symbol table**
put(st, key, value)	insert a **key-value** pair
get(st, key)	retrieve a **value**, or **null**
delete(st, key)	remove a **key** from **st**
contains(st, key)	**true** if **st** contains **key** else **false**
st_empty(st)	**true** if **st** empty, else **false**
st_size(st)	returns # of **keys** in st
min(st)	returns smallest **key**, or **null**
max(st)	returns largest **key**, or **null**
keys(st)	returns array of **keys** in sorted order



the inclusion of min, max, and keys makes this an **ordered symbol table**

13.1 Unordered Linked List Implementation

```
SymbolTable:
  head: Node

Node:
  key: any
  value: any
  next: Node
```

This adds new **key-value** pairs at the front which is an $O(1)$ operation. However, it does take $O(n)$ to search for **keys** by traversing the list.

13.2 Fixed-Length Ordered Array Implementation

```
SymbolTable:
  keys: array
  values: array
  n: int      # num of items and index of next available
```

This creates two arrays which are related in parallel. The basic approach is to put new **key-value** pairs in the correct sorted order. This allows us to search for keys in a more efficient way using **binary search**.

Another implementation to maintain order is with an array of **nodes**.

```
SymbolTable:
  items: array of Nodes
  n: int
```

When we compare these two implementations, we see that the **get** and **contains** operations are now $O(\log n)$ in the ordered implementation as opposed to the $O(n)$ in the unordered. This is because we can now take advantage of **binary search**.

14 Binary Search

This searching algorithm only works for sorted lists of items. It finds the center of the search area and determines whether the item we are searching for is smaller or larger than that middle element. It then divides its search area in half again and continues.

```
def bsearch(items, key):
  lo = 0
  hi = len(items) - 1

  while lo <= hi:
    mid = lo + (hi - lo) // 2

    # if our key is exactly the middle key
    if key == items[mid].key:
      return mid

    # if our key is larger than the middle key
    if key > items[mid].key:
      lo = mid + 1

    # if our key is smaller than middle key
    else:
      hi = mid - 1

  return hi or lo
```

Note, you would ideally return the **lo** index if you would want to insert a key-value pair and are searching for the correct position to insert it.

14.1 Time Complexity

Starting at $i = 0$ (note this is different from the profs), we will see that similar to merge sort, each node on level i has $\frac{n}{2^i}$ elements in the array. Assuming the worst case where n is not a power of 2 and the search always goes to the largest side, we can determine that

$$T(n) \leq \lceil \log n \rceil \leq \log n + 1 \in O(logn)$$

Here is the professors way. He considers $i = 1, s_1 = n$ so that on any level, each node has $\frac{n}{2^{i-1}}$ elements.

$$\begin{aligned} \frac{n}{2^{h-1}} &\leq 1 \\ n &\leq 2^{h-1} \\ \log n &\leq h - 1 \\ \log n + 1 &\leq h \\ \log n + 1 &\in O(\log n) \end{aligned}$$

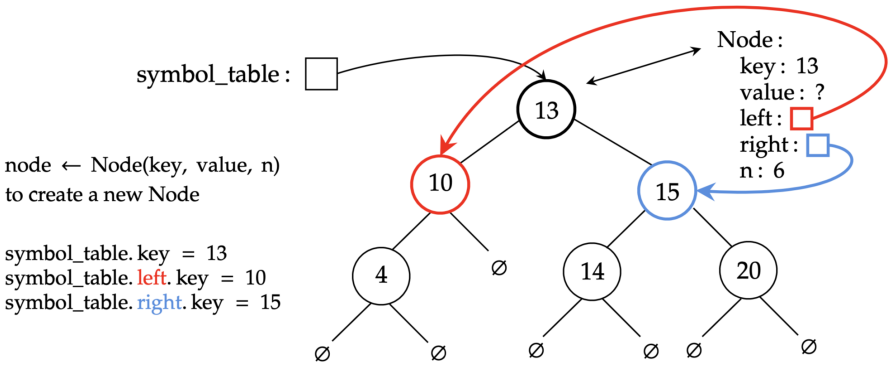
15 Binary Search Trees

A binary search tree (BST) is a data structure where each node has a **key** (and associated value) and satisfies the restriction that its key is larger than the keys in its **left subtree** and smaller than the keys in its **right subtree**.

- combines the efficiency of the binary search algorithm with the insertion/deletion of linked lists
- get, contains, max, min are $O(\log n)$, the search part of put and delete are also $O(\log n)$, the work after searching for put and delete is $O(1)$.
- inherently recursive – every node in a BST is also a BST

15.1 General Structure

```
Node:
  key: any
  value: any
  left: Node   # left subtree
  right: Node  # right subtree
  n: int       # size of THIS subtree

symbol_table: Node   # the st is the bst (points to the
↪  first node)
```



15.2 Basic Operations

Here we define the pseudocode for the basic operations we can use on a BST.

15.2.1 Size

This is as simple as checking if the symbol table is null or returning n.

```
st_empty(st):
  return st == null

st_size(st):
  if st == null:
    return 0
  return st.n
```

15.2.2 Searching

We use the binary search algorithm to search for a key. This can result in a hit or a miss.

```
get(st, key):
  # base case
  if st == null:
    return null

  # larger means right subtree
  if key > st.key:
    return get(st.right, key)

  # smaller means left subtree
  if key < st.key:
```

```
    return get(st.left, key)

  # either the value of the key we are looking for or null
  return st.value
```

Recursion is tricky to visualize with trees, but essentially every recursive call runs through the entire algorithm again but on the subtree. In this algorithm, we return from the recursion if the key == st.key, and all the previous calls don't return anything. This algorithm will return a *miss*, if we hit a subtree of null.

15.2.3 Minimum Key

Since we are utilizing a BST, the smallest key is the farthest left branch.

```
min(st):
  # base cases - empty and singleton
  if st == null:
    return null

  if st.left == null:
    return st

  return min(st.left)
```

Similar logic would apply for the maximum key, so we won't implement it.

15.2.4 Insertion

This algorithm will insert the value to the given key. There are 3 cases to consider:

- key does **not** exist
  - the location where we *would* put the new key-value pair **is** the same location we would find if we searched for the key and got the null subtree.
- key does exist
- empty tree

```
put(st, key, value):
  # base cases - empty tree so we create one
  if st == null:
    return Node(key, value, 1)
```

```
    # if key in tree, update the value
    if key == st.key:
      st.value = value
      return st

    # key smaller so left subtree
    else if key < st.key:
      st.left = put(st.left, key, value)

    # key larger so right subtree
    else:
      st.right = put(st.right, key, value)

    # update and return n
    st.n = st_size(st.left) + st_size(st.right) + 1
    return n
```

Essentially we are searching for where to insert the key with the two recursive calls. If one of these calls finds the key, then we update the value. However, if we run into a null subtree, we know that is the location where the new key should be inserted, so we can just create a new node there!

15.3 Complexity Analysis of Operations

The time complexity of these operations is tied to the **structure** of the BST itself.

Worst Case

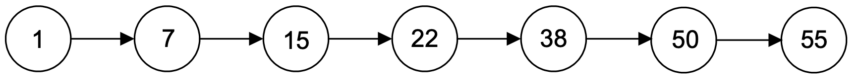
- insertion happens in-order causing the BST to resemble a linked list. This results in the search aspect of these operations to now be $O(n)$.

Best Case

- the keys are inserted so that the tree is as close to **perfect** as possible. If we insert the keys in a binary search fashion, taking the middle of each subarray, we can insert them, so the tree abides to its restrictions, ultimately allowing the search to remain as $O(\log n)$.

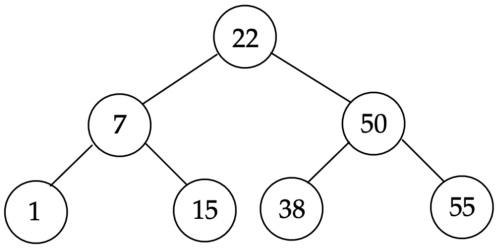
Worst Case: 1, 7, 15, 22, 38, 50, 55

1 2 3 4 5 6 7



Best Case: 1, 7, 15, 22, 38, 50, 55

3 2 4 1 6 5 7



16 More BST Algorithms

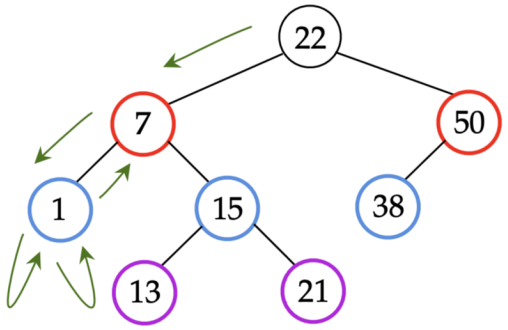
16.1 Tree Traversal

To **traverse** a tree is to **visit** a node exactly once. We typically say visiting a node is preforming some action on that node like printing its value. The recursive algorithm boils down to visiting the left and right subtrees in that order.

Traversal Order – the order of when you visit the node

- pre-order – preform the action on the **first** time you visit the node
- in-order – preform the action on the **second** time you visit the node
- post-order – preform the action on the **third** time you visit the node

```
traversal(22):
  traversal(7):
    traversal(1):
      traversal(∅)
      traversal(∅)
    traversal(15):
      traversal(13):
        traversal(∅)
        traversal(∅)
      traversal(21):
        traversal(∅)
        traversal(∅)
  traversal(50):
    traversal(38):
      traversal(∅)
      traversal(∅)
    traversal(∅)
```



pre – order: 22, 7, 1, 15, 13, 21, 50, 38
in – order: 1, 7, 13, 15, 21, 22, 38, 50
post – order: 1, 13, 21, 15, 7, 38, 50, 22

For in-order you'll simply notice how it visits the nodes in a sorted order.

16.1.1 General Structure

```
def traverse(st):

  if st == null:
    return

  visit(st) # pre-order
  traverse(st.left)
  visit(st) # in-order
  traverse(st.right)
  visit(st) # post-order
```

Notice the time complexity is linear, however, it has a larger constant factor than if the tree was simplified to a linked list.

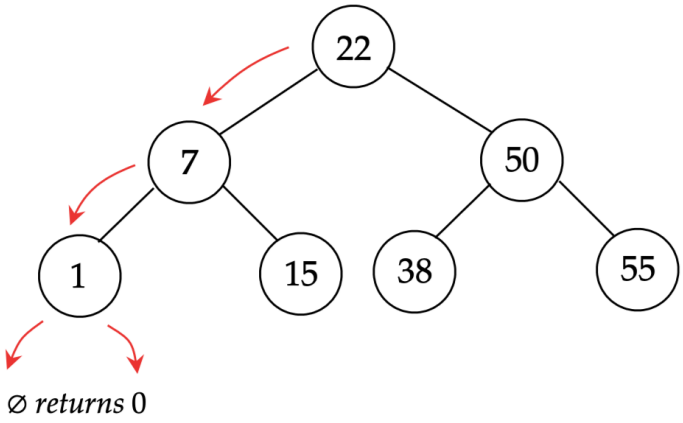
16.1.2 Traversal Example: Height

To get the height of a tree we can simply traverse the tree in any order.

```
def height(st):

  if st == null:
```

```
        return 0

    return max(height(st.left), height(st.right)) + 1
```

This function recursively calls height until we reach the traversal of the null nodes. 'height()' will return a value, so that is used in the outer recursive call as a value as seen below.



(7) returns max((1) , (15)) + 1
so (1) returns max(0, 0) + 1 = 1
same for (15) = 1
∴ (7) will now be max(1, 1) + 1 = 2

(22) returns max(2, (50)) + 1 = *height of tree*

16.2 Deleting From BST

There are 4 cases to consider:

(1) key is not found

- raise error

(2) key node has **no** children (leaf nodes)

- remove link from parent (delete and set parent link to null)

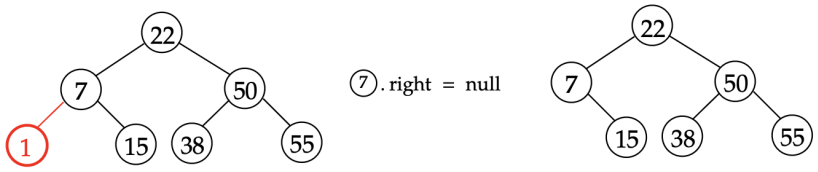
(3) key node has **one** child

- replace with child (make parent of deletion node point to child of deletion node)

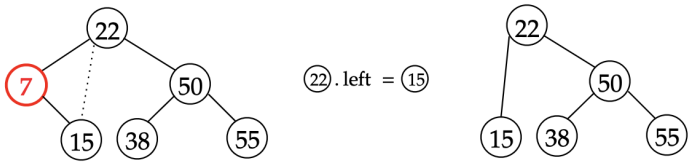
(4) key node has **two** children

- find the minimum key in right subtree and replace deletion node with the min node. Then delete the min node (since it is a leaf node, it is easy to remove).

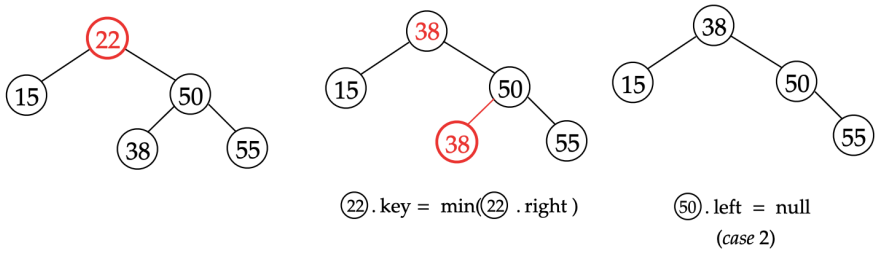
Case 2 : delete leaf node



Case 3 : delete node with one child



Case 4 : delete node with two children



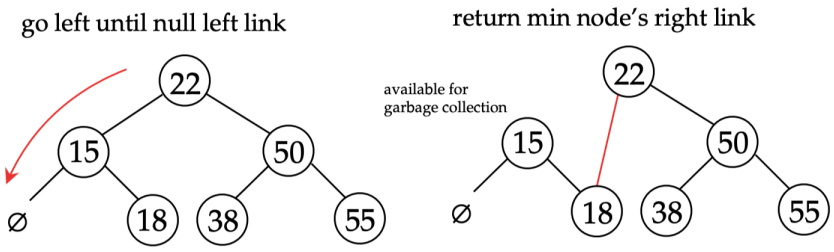
Note: since deletion will require rearrangement of a subtree, we need to return a new root node.

16.2.1 Helper Functions

We can define a helper function to delete a minimum node, starting from the root node!

```
def deleteMin(st):
  if st == null:
    raise error

  # make min node's parent point to min node's .right child
```

```
  if st.left == null:
    return st.right

  # continue going left to find min node
  st.left = deleteMin(st.left)

  # update size
  st.n = size(st.left) + size(st.right) + 1

  return st
```



This algorithm will update the min node's parent node, so it points to the min node's right child. Then it returns the entire subtree after the deletion.

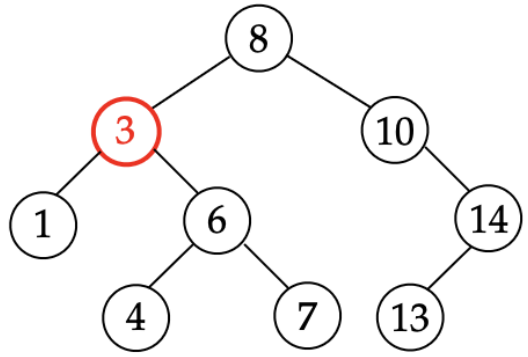
16.2.2 Algorithm

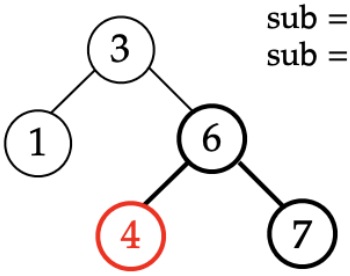
```
def delete(st, key):
  if st == null:
    return error

  # search for the key (larger means go right)
  if key > st.key:
    st.right = delete(st.right, key)

  # smaller means go left
  else if key < st.key:
    st.left = delete(st.left, key)

  # found the key to delete
  else:
    # case 2 and 3 - leaf or one child
    if st.right == null: return st.left
    if st.left == null: return st.right

    # case 4
```

```
        sub = min(st.right)      # finds the min node
        sub.right = deleteMin(st.right)  # deletes min node from
        ↪  right subtree of deletion node (returns st.right)
        sub.left = st.left
        st = sub

    # update size
    st.n = size(st.left) + size(st.right) + 1
    return st
```

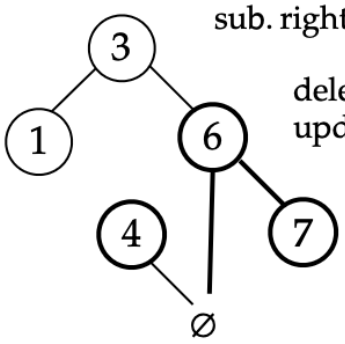
Remember that when we return we are calling it like st.right = delete(...), so the return value alters the node st.right. For example, for case 2, if we are returning st.right which is null, the parent node (st) is now equal to null, effectively deleting it.

delete(st, 3)





sub = min(**st. right**)
sub = (4)



sub. right = deleteMin(**st. right**)

deleteMin in this case will
update (6) . left to null and return (6)

so (4) . right = (6)

---

sub. left = st. left
so (4) . left = (1)

st = sub
so (4) = (3)

17 Balanced Trees

The problem with BST's is that they can be unbalanced, and it all depends on the structure of the tree. Best case our insertion, deletion, and search algorithms are logarithmic ($T(n) \approx c\lceil \log n \rceil$). Worst case we get linear. The best to combat this is to keep the tree **balanced** – accomplished through 2 different approaches.

17.1 2-3 Trees

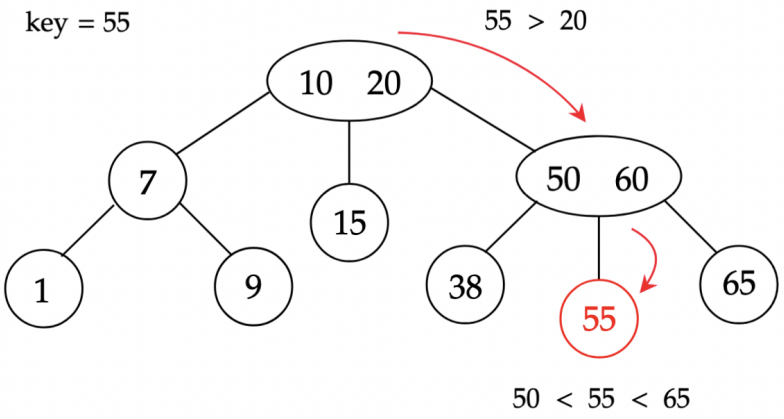
With 2-3 trees, we create them with 2-nodes and 3-nodes:

- 2-node: one key-value pair, two children (< L, > R)
- 3-node: two key-value pairs, **three** children (< L, L > ... < R, > R)

Note the BST ordering (left is smaller, right is bigger) remains the same. The reason why BST structures can get unbalanced is because the insertion of a node means adding a new leaf node. With 2-3 trees, we instead expand a leaf node – ensuring the leaf nodes are on the same level to maintain the tree's height.

17.2 Searching

Searching for a key follows the same rules, however, now we consider the middle child for 3-nodes.



17.3 Insertion

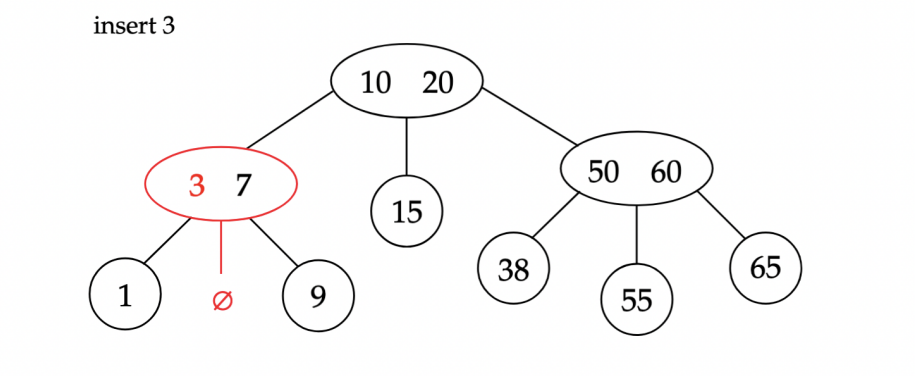
To insert a new node while maintaining balance, we see we have three cases:

(1) inserting into a 2-node (making it a 3-node)

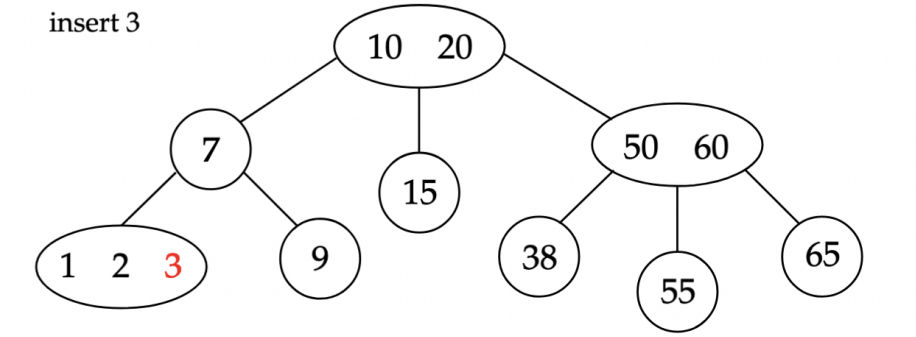
(2) inserting into a 3-node when parent is 2-node

(3) inserting into a 3-node when parent is a 3-node

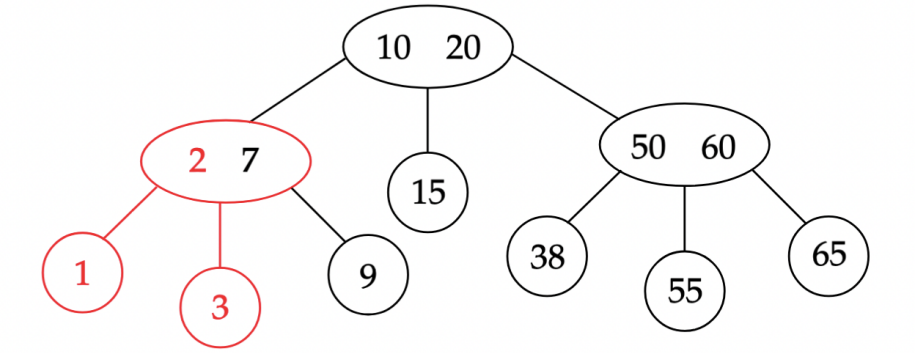
17.3.1 Case 1



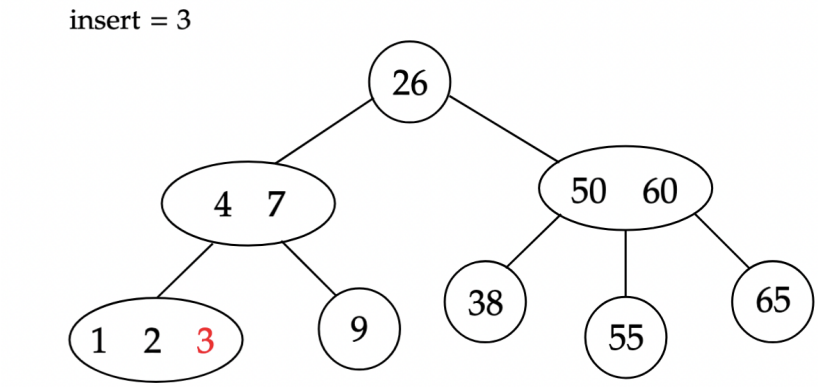
17.3.2 Case 2



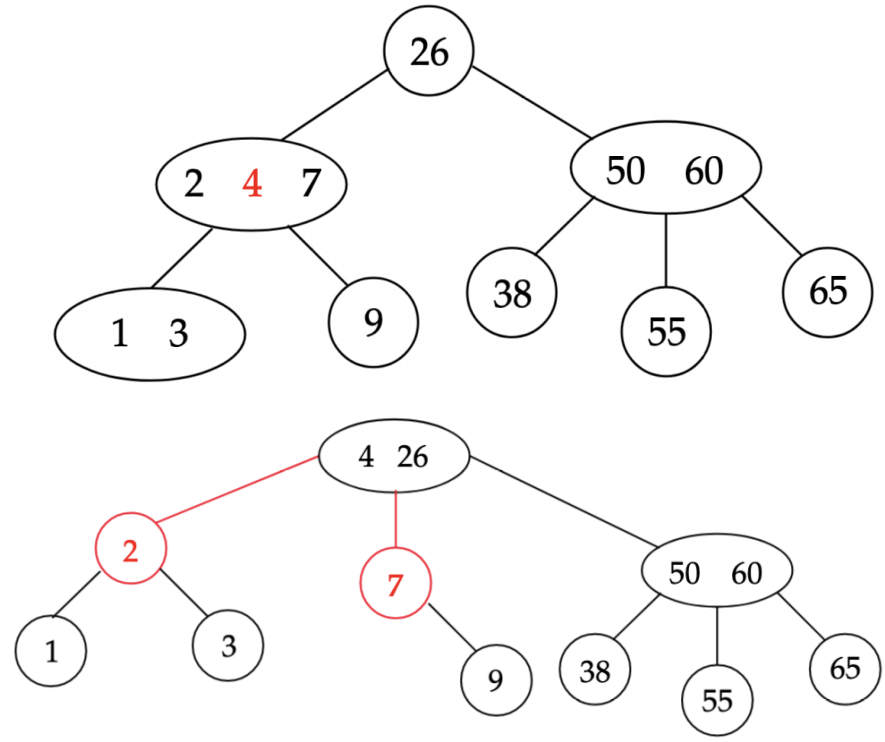
but what happens when we get a **4-node**? We **split** it and pass the **middle** key up.



17.3.3 Case 3



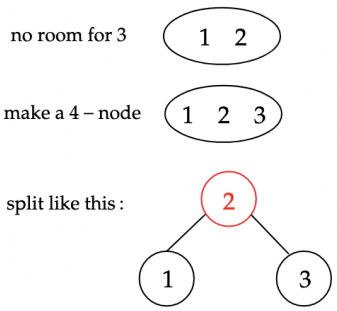
We repeat the same process as before and bring the middle key up!



Notice how we split the nodes so the height of the tree is maintained.

17.4 Splitting The Root

Splitting these 4-nodes into 3-nodes and so on causes the tree from the bottom up. This is the way the split should work:



17.5 Time Complexity

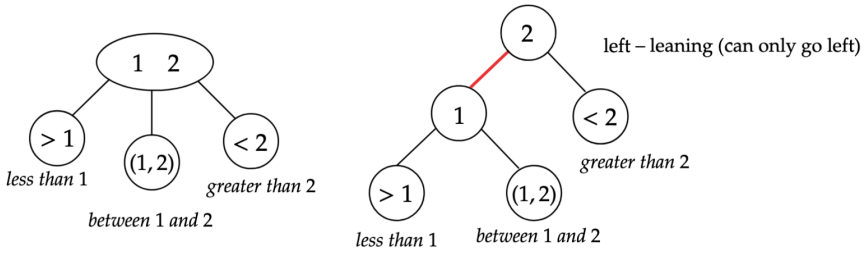
Since the tree is always **balanced** and maintains the fact that the leaf nodes are on the same level, our height is always logarithmic.

The worst case is when we have a BST (**all 2-nodes**) where $h = \log_2 n$. Our best case is when we have **all 3-nodes** so $h = \log_3 n$. This guarantees our algorithms are always logarithmic and eliminates the possibility of the BST worse case of $O(n)$ timed algorithms because the structure was unbalanced!

18 Red-Black Trees

We've discovered the fact BST's can be unbalanced and used 2-3 trees as a solution to maintain balance. However, 2-3 trees are cumbersome to implement, so we now introduce '**Left Leaning Red-Black Trees**'.

This means we can represent 3-nodes with a left-leaning red link where the larger key is the root.



There are 3 restrictions for a tree to be considered a red-black tree.

(1) red links always lean left

(2) no node has two red links connected to it

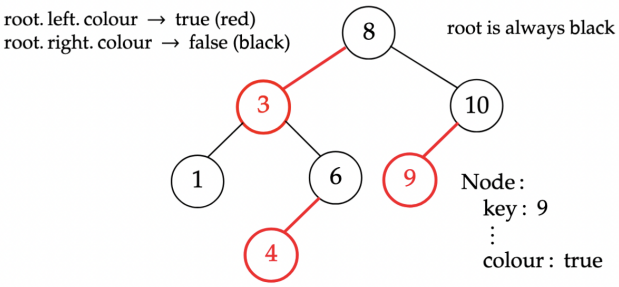
(3) the tree has perfect **black balance** – both right and left subtrees have the same **black height**.

- black height – every path from the root node to null nodes has the same number of black links

18.1 Implementation

It would be difficult to keep track of the colour of links, so instead we can add a colour property to each node.

```
Node:
  key: any
  value: any
  left: Node
  right: Node
  n: int
  colour: boolean (true = red)
```



Notice how the root is always black and that adding red links keeps the black height the same.

In general, we can reuse a lot of BST code.

- empty, size, min, get, print

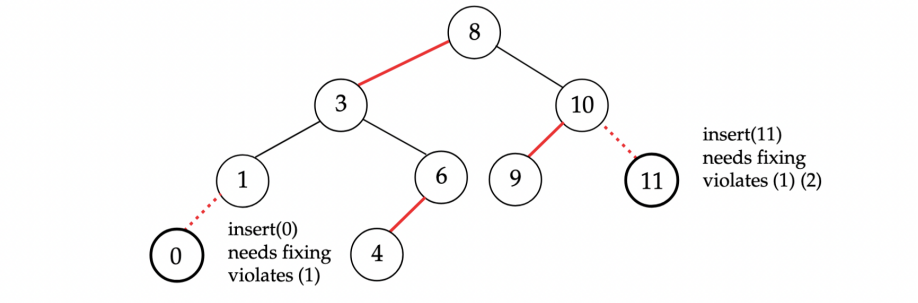
The one algorithm that is quite different is the insertion or put operation.

18.2 Insertion

There are two phases we can consider.

(1) search for the key, if found update it, otherwise create a new **red** leaf node to maintain black balance.

(2) restore the red-black properties – ensure red links are left leaning & no node has two red links



To fix these issues, we have three primitive operations:

(1) Left rotation – fix right-leaning links

- safe to perform on any right-leaning links as it only moves the red link (doesn't change black height)

(2) Right rotation – intermediate step in a larger process

- safe to perform on any left-leaning links

(3) Colour Flip – equivalent to splitting a 4-node / 2 red links on one node

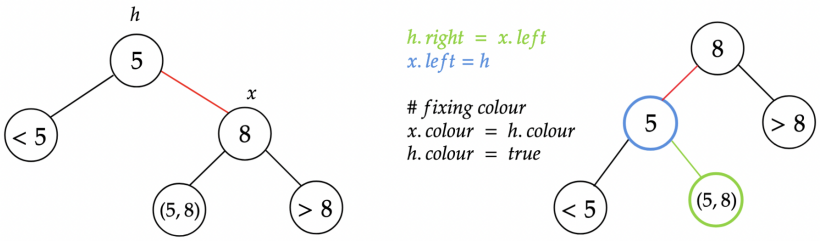
- safe to perform on black node with 2 children

18.2.1 Left Rotation

When we have a right-leaning link, we can rotate the tree to the left. In general left rotations:

- the red link node's left child becomes the old node
- the old node's right child becomes the red link node's left child

In this diagram below, h is the old node, x is the red link rotational node.




```
def rotate_left(h):
  x = h.right
```

```
    # root's right child is red link node's left child
    # red link's left child is the old root (making the red
    ↪  link node the new root)
    h.right = x.left
    x.left = h

    # perserve colour from the previous root node
    x.colour = h.colour
    h.colour = true

    # update size
    x.n = h.n
    h.n = size(h.left) + size(h.right) + 1

    return x
```

18.2.2 Right Rotation

With similar but opposite logic we can do a right rotation. We simply flip the logic for the rotation.

- the red link node's right child becomes the old node
- the old node's left child becomes the red link node's right child

```
def rotate_right(h):
  x = h.left

  h.left = x.right
  x.right = h
```

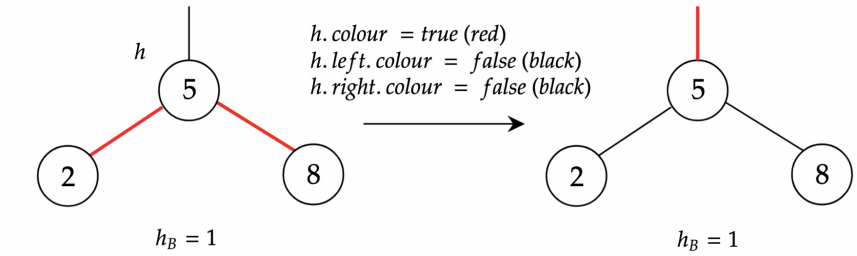
This is the only logic that changes!

18.2.3 Color Flip

We do a colour flip when we have two red link children to a black node. This is the equivalent of splitting a 4-node.

- this still maintains black balance!

```
def colour_flip(h):
  h.colour = true
  h.left.colour = false
  h.right.colour = false
```



Now with these primitive operations we can create an algorithm to restore red-black properties.

Phase 2

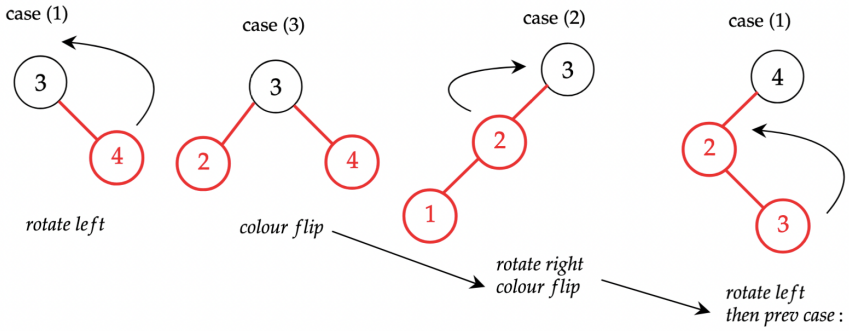
- fix red links while preserving black balance
  - proceed upward from the inserted leaf

  (1) rotate left if you have a right learning red link

  (2) rotate right if your red left child has a red left child

  (3) colour flip if you have two red children
- colour the root black if it's red (add $h_B+ = 1$)

We have 3 basic situations where we can apply these primitive operations to restore our tree.



Case (1) can result in Case (2) which can result in Case (3). Remember to flip the root to black if it is red. This only when the **black height** increased by 1.

We can simply convert this verbal description into code

```
def put(h, key, val):
  # standard BST insertion (finding the key)
  # creating the red node
```

```
    if h == null:
      return Node(key, val, 1, true)
    if key < h.left:
      h.left = put(h.left, key, val)
    if key > h.right:
      h.right = put(h.right, key, val)
    else:
      h.value = val

    # restoring the red-black properties
    # case 1
    if is_red(h.right) and not is_red(h.left):
      h = rotate_left(h)

    # case 2
    if is_red(h.left) and is_red(h.left.left):
      h = rotate_right(h)

    # case 3
    if is_red(h.left) and is_red(h.right):
      h = colour_flip(h)

    # update size
    h.n = size(h.left) + size(h.right) + 1
    return h
```

19 Time Complexities of All Trees

For all these complexities, consider n is the number of keys per height h where $h = 0$ for an empty tree.

19.1 Binary Search Tree - BST

Notice that there are $2^i - 1$ keys on any given level. For BST's we can say

$$2^h - 1 = n$$
$$h = \log_2(n) - 1 \in O(\log n)$$

where h is the total height of the tree.

19.2 2-3 Trees

Notice that there are $3^i - 1$ keys on any given level. For 2-3 Tree's best case (all nodes are 3-nodes) we can say

$$3^h - 1 \geq n$$
$$h = \log_3(n) - 1 \in O(\log n)$$

where h is the total height of the tree. The worst-case is when all nodes are 2-nodes which is a BST.

19.3 Red Black Trees

For this we must consider the black height $bh(x)$ and actual height h of the tree. Notice how $bh(x) = h/2$. Therefore we can say

$$2^{bh(x)+1} \geq n$$
$$bh(x) + 1 \geq \log n$$
$$h/2 \geq \log n + 1$$
$$h \geq 2(\log n) + 1$$

20 Heaps

20.1 Array Based Tree Representation

We primarily use linked lists for trees but array based tree representation is possible.

- children of element i are at $2i + 1$ and $2i + 2$.
- parent of element i is at $\lfloor (i - 2)/2 \rfloor$

Complete Binary Tree – every level is full except last, filled starting from left
Perfect Binary Tree – every node has 2 children, all leaf nodes are on same level

Array based tree representations work best for **complete binary trees** as it's the most efficient.

20.2 Heaps

Heaps are any tree that exhibit the **heap-ordering** property that the key at each root node in a subtree is the maximum (max heap) or minimum (min heap) for that subtree.

Binary Heap – complete, heap-ordered binary tree. This is different than a regular heap!

20.3 Priority Queue

A basic ADT that supports insertion and retrieval of the **highest priority** item.

- **highest priority** – maximum or minimum **key** stored in the priority queue
- insert and remove is $O(\log n)$

** from now on assume maximum key (max heap) **

20.3.1 Primitive Operations

We have two primitive operations to repair ('reheapify') it for when we change the heap.

(1) swim (sift up)

(2) sink (sift down)

reheapify – restore the heap-ordering property when a single key is violated

20.3.2 Swim - Sift Up

- key too large
- $O(\log n)$ – worst case traverses through the height of the tree

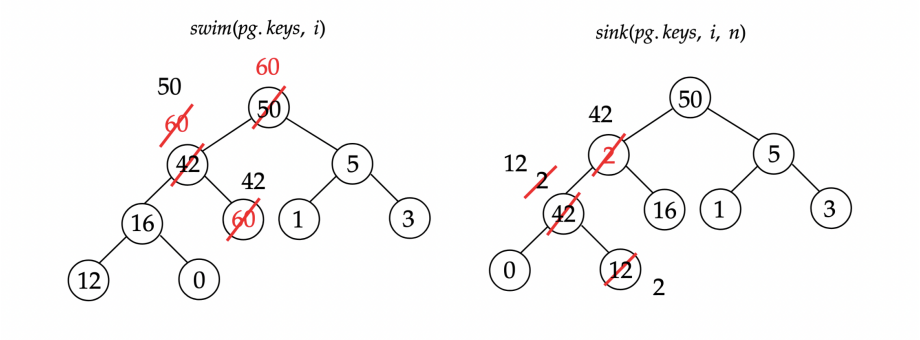
```
def swim(arr, i):
  parent_index = (i - 1) // 2

  # keep swapping the parent and index node
  while i > 0 and arr[parent_index] < arr[i]:
    swap arr[parent_index], arr[i]
    i = parent_index
    parent_index = (i - 1) // 2
```

20.3.3 Sink - Sift Down

- key too small
- $O(\log n)$ – worst case traverses through the height of the tree

```
def sink(arr, i, n):
  while 2 * i + 1 < n:

    # left child
    j = 2 * i + 1
```

```
    # right child
    if j < n - 1 && arr[j] < arr[j + 1]:
      j += 1

    if arr[j] <= arr[i]:
      break

    swap arr[i], arr[j]
    i = j
```



21 Heaps and Heapsort

Here we define two operations for insertion and deletion.

21.1 MaxPQ Insert

- insert the key at location n and swim it up
- worst case complexity of insert $O(\log n)$

```
def insert(pq, key):
  pq.keys[pq.n] = key    # pq[n] = key
  swim(pq, pq.n)         # bubble it up to correct spot
  pq.n += 1        # increase size
```

21.2 MaxPQ delete_max

- move the $n-1$th key to root ($n=0$) and sink down
- return max

```
def delete_max(key):
  max = pq.items[0]
  pq.n -= 1
  pq.items[0] = pq.items[n]  # move last key to top root
  sink(pq, 0)
  return max
```

21.3 Heapsort

We can sort an array using heaps with the following process

(1) **heapify** the array, turning it into a **max heap**

(2) call **delete_max**(), place the max at the end of heap array. Repeat until heap is size 1

Therefore the time complexity for (2) is $\Theta(n \log n)$ and for (1) will be determined below. The space complexity is $S(n) \approx c_1 n + c_2 \in O(n)$

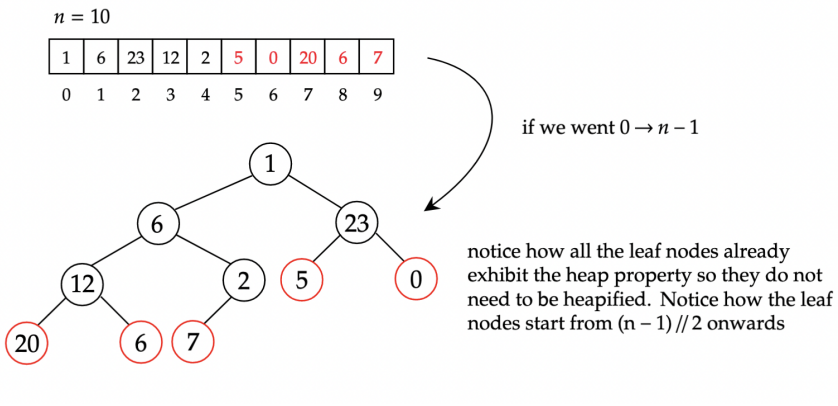
21.3.1 Heapify

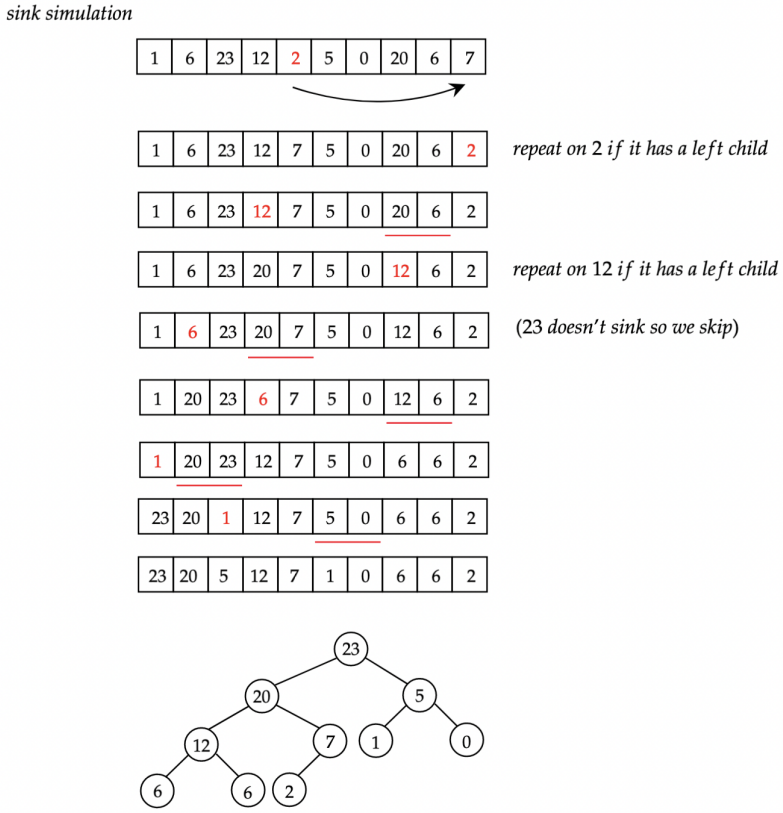
To heapify is to turn any array into a max heap. The naive approach is to go from 0 to n and reheapify each time. However, if we consider the fact leaf nodes do not need to be reheapified, we can start from the last non-leaf node and preform the heapify operation in reverse.

- proceed from $\lfloor n/2 \rfloor - 1$ to 0 and use sink

Therefore this algorithm boils down to simply

```
def heapsort(arr):
  n = len(arr)

  # heapify the array
  for key = (n-1)//2 down to 0:
    sink(arr, key, n)

  # sort using sink
  while n > 0:
    swap arr[0], arr[n-1]
    n -= 1
    sink(arr, 0, n)
```





22 Hash Tables

We implement a symbol table as a hash table to achieve constant time operations.

- hash function – takes in a key and returns an index to a table (size m)
- collisions – two keys return the same index