Contents

I Recursion . . . 1
II Complexity . . . 1
III Sorting . . . 3
IV Linked List & Friend Class . . . 5
V Stack & Queue . . . 6
VI Tree . . . 7
VII Heap . . . 9
VIII Red Black Tree . . . 10
IX Selecting Data Structures and Algorithmic Strategies Quiz . . . 12

I Recursion

- **Tail recursion** uses less memory than non-tail recursion.

I.1 Tail Recursion

I.1.1 Cost of recursion

- Each call to a function adds another frame on the stack
- Each frame contains local variables and parameters and where to return the result

I.1.2 Reducing what needs to be stored

- If we can guarantee we won't need them, we can free the memory for the local variables and parameters.
- We won't need them as long as **we do not use them after the recursive call**.
- Tail recursive functions have the **recursive call as the last thing the function does before it returns**.

II Complexity

II.1 Master Theorem

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$T(x) = \begin{cases} \Theta(n^d) \text{ if } a<b^d \\ \Theta(n^d \log(n)) \text{ if } a=b^d \\ \Theta\left(n^{\log_b(n)}\right) \text{ if } a>b^d \end{cases}$$

a: number of subproblems in the recursion
n/b: size of each subproblem
d: the exponent in the cost of the work done outside the recursive calls, specifically in the non-recursive part of the algorithm (like splitting or merging the problem)

e.g. **Binary Search** a=1 b=2 d=0 $\log(n)$ **Merge Sort** a=2 b=2 d=1 $n\log(n)$

II.2 Bound

II.2.1 Big O - Upper Bound

$f(n) = O(g(n))$ iff (if and only if)
$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 : f(n) \leq cg(n)$

II.2.2 Big Ω - Lower Bound

$f(n) = \Omega(g(n))$ iff (if and only if)
$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 : f(n) \geq cg(n)$

II.2.3 Big Θ - Tight Bound

$f(n) = \Theta(g(n))$ iff (if and only if)
$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$

II.2.4 Little o upper Bound

$\exists c \in \mathbb{R}^*, \exists n_0 \in \mathbb{N}, \text{such that } \forall n \geq n_0 \text{ such that } f(n) < cg(n) \left(\left| \frac{f(n)}{g(n)} \right| < c \right)$
In other words: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

II.3 Quiz

II.3.1

- if $f(n) = \Theta(g(n))$ then $g(n) = \Theta(f(n))$
- if $f(n) = O(n)$ then $f(n) = o(n)$
- if $f(n) = o(n)$ and $g(n) = o(n)$ then $f(n) + g(n) = o(n)$
- $O(n)$ is the complexity in the worst case
- if $f(n) = \Omega(g(n))$ and $f(n) = O(h(n))$, then $g(n) \leq h(n)$ for all $n > 0$

II.3.2

$o(n^2)$ = $\sqrt{n}$ / 1 / $n \log(n)$ / n

II.3.3

$\mathbf{1 < \log(n)} < \log(n^2) < (\log(n))^2 < 5n \log(\log(n)) + 100 < \boldsymbol{n < n \log(n)} < n^{1.5} + 1000000 < \boldsymbol{n^2 < n^3 < 2^n} < 3^n < \boldsymbol{n! < n^n}$

II.3.4

- If $f(n) = O(g(n))$, then $f(n) = o(g(n))$. The opposite is not true, because Little o assumes that $f(n)$ and $g(n)$ can't be of the same order.
- If $f(n) = o(g(n))$, then $f(n) = O(g(n))$. The opposite is not true, because Little o assumes that $f(n)$ and $g(n)$ can't be of the same order.
- $f(n) = o(g(n))$ if and only if $f(n) = O(g(n))$.
- If $f(n) = O(g(n))$ it does not mean $f(n) = o(g(n))$, and if $f(n) = o(g(n))$ it does not mean $f(n) = O(g(n))$.

II.3.5

- $f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$
- $n = o(n^2) \text{ and } n = O(n^2)$
- $\text{If } f(n) = \Omega(n), \text{then } f(n) \geq n \text{ for all } n \text{ starting from some } n_0$
- $\text{If } f(n) = o(g(n)) \text{ and } g(n) = O(h(n)), \text{then } f(n) = o(h(n))$

II.3.6

Suppose we need to find an occurrence of the array of size m inside a larger array of size n. We know that the array of size n is sorted and consists of unique elements. What will be the time complexity of the optimal algorithm?

- $\Theta(\log(n) + m)$

III Sorting

III.1 Selection

不断从未排序部分中选择最小（或最大）元素，并将其与第一个未排序元素交换，每次迭代后将排序边界向前移动一步 repeatedly selects the smallest (or largest) element from the unsorted portion of an array and swaps it with the first unsorted element, moving the sorted boundary one step forward with each iteration

```
for (int i = 0; i < array.size(); i++) {
  // Find min element from i to n-1
  for (int j = i + 1; i < array.size(); j++) {
  }
  // Swap elements at index i and min elements
}
```

III.2 Insertion

逐一将未排序部分的下一个元素插入到（不断交换）已排序部分的正确位置，逐步构建出一个已排序的数组 builds the sorted array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position in the sorted part

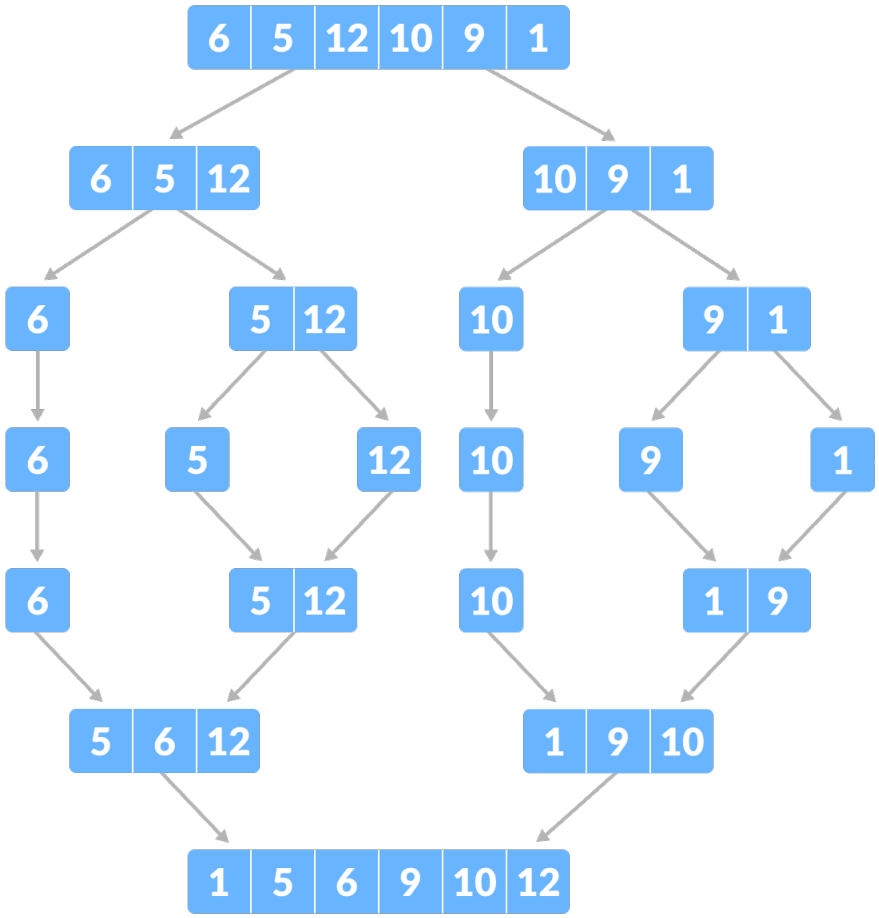
```
for (int i = 1; i < array.size(); i++) {
  for (int j = i; j > 0; j--) {
    if (array.at(j) < array.at(j - 1)) {
      swap(array.at(j - 1), array.at(j));
    } else break;
  }
}
```

III.3 Bubble

repeatedly compares adjacent elements and swaps them if they are in the wrong order, causing larger elements to “bubble” to the end of the array. With each pass through the list, the largest (or smallest) element settles in its correct position, and this process continues until the entire array is sorted.

```
for (int i = array.size() - 1; i > 0; i--) {
    for (int j = 0; j < i; j++) {
      if (array.at(j) > array.at(j + 1)) {
        swap(array.at(j), array.at(j + 1));
      }
    }
  }
```

III.4 Merge




```
if (array.size() == 1) return array;
// Sort left and right subarrays
int mid = array.size() / 2;
// ... Define left and right arrays
vector<int> sortedRightArray = sort(rightArray);
vector<int> sortedLeftArray = sort(leftArray);
// Merge left and right subarrays
vector<int> result;
int l = 0, r = 0;
while (l < sortedLeftArray.size() && r < sortedRightArray.size()) {
  if (sortedLeftArray.at(l) < sortedRightArray.at(r)) {
    result.push_back(sortedLeftArray.at(l)); l++;
  } else {
    result.push_back(sortedRightArray.at(r)); r++;
  }
}
// ... Add remaining elements form sortedLeftArray or sortedRightArray
return result;
```

III.5 Quick

```
if (start >= end) return;
// Select the last element as pivot
int pivot = array.at(end);
```

```
int pivotIndex = start;
for (int i = start; i < end; i++) {
  if (array.at(i) < pivot) {
    swap(array.at(i), array.at(pivotIndex));
    pivotIndex++;
  }
}
swap(array.at(pivotIndex), pivot);
sort(array, start, pivotIndex - 1);
sort(array, pivotIndex + 1, end);
```

IV Linked List & Friend Class

```
template <class T>
class Node {
  T data;
  Node<T>* link;
  Node(T data) {
    this->data = data;
  }
public:
  template <class U>
  friend class LinkedList;
};

template <class T>
class LinkedList {
  Node<T>* head;
  bool isEmpty() {
    return this->head == NULL;
  }
public:
  LinkedList() {
    this->head = NULL;
  }
  void insert(T value) {
    Node<T>* newNode = new Node<T>(value);
    newNode->link = this->head;
    this->head = newNode;
  }
  void traverse() {
    if (isEmpty()) {
      cout << "List is empty\n";
      return;
    }
    Node<T>* tmp = this->head;
    while (tmp != NULL) {
      cout << tmp->data << " ";
      tmp = tmp->link;
    }
    cout << endl;
  }
};
```

IV.1 Quiz

IV.1.1

- Linked lists require at least two methods for insertion, one to cover addition at the front and another for addition at an arbitrary position.
- Linked lists have $O(1)$ access to any node.
- Linked lists are not guaranteed to have their nodes located in adjacent blocks of memory.
- Linked lists have an aggregation relationship with nodes.

IV.1.2

- A singly linked list has all the nodes linking in one direction.
- A singly linked list can have a tail as well as head. When it does, insertion and deletion at both either end becomes $O(1)$
- Insertion/deletion from the front of a singly linked list is $O(1)$.
- A singly linked list cannot have a tail member variable that records where the last node is in memory, as it violates the definition of the singly linked list.

V Stack & Queue

V.1 Stack

- LIFO (Last-in/First-out)
- implemented using linked lists or dynamic arrays
- `push(item)` `pop()` `isEmpty()`
- only have access to the top of the stack

V.2 Queue

- FIFO (First-in/First-out)
- usually implemented using linked lists
- `enqueue(item)` `dequeue()` `isEmpty()`
- You have access to both the front and back of the queue.

V.3 Quiz

V.3.1

- Removing an element from a queue requires the queue to search through all of the remaining elements to update their positions and ensure that the queue is properly ordered, resulting in a time complexity of O(n).
- Solving the Tower of Hanoi maps well to the queue data structure.
- Queues are FIFO/LILO.
- Since queues involve operations at both ends, they must be implemented using a doubly linked list.

V.3.2

- A priority queue implemented using a single queue where each node has a priority value, has both $O(n)$ insertion and $O(n)$ removal.
- A priority queue with n elements that is implemented using multiple queues (with a queue for each priority) have $O(n)$ removal.
- A priority queue with n elements that is implemented using multiple queues (with a queue for each priority) have $O(1)$ insertion.

- A priority queue implemented using a single queue where each node has a priority value, has either $O(n)$ insertion or $O(n)$ removal.

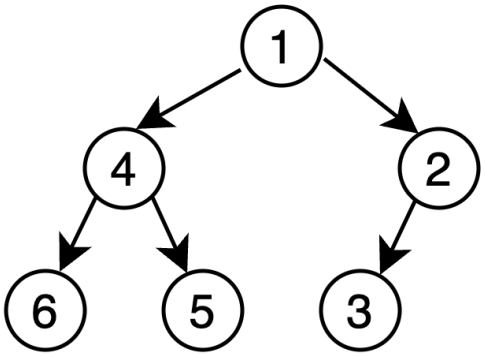
V.3.2.1 Explanation

- Priority queues with n nodes implemented using multiple queues, have:
  - $O(1)$ insertion, as at the time of insertion you know what priority the node will have so you can simply find which queue it should belong to in $O(1)$, and then insert to the back of that queue in $O(1)$.
  - $O(m)$ removal, where m is the number of queues. This is because we need to run the isEmpty function on up to m queues.
- Priority queues implemented using one queue will either have either $O(n)$ insertion or removal. Because you have to check who the highest priority is either:
  - At insertion, to place them in the appropriate position. They "cut" in line.
  - At removal, to take out the appropriate item.

VI Tree

VI.1 Tree Traversals

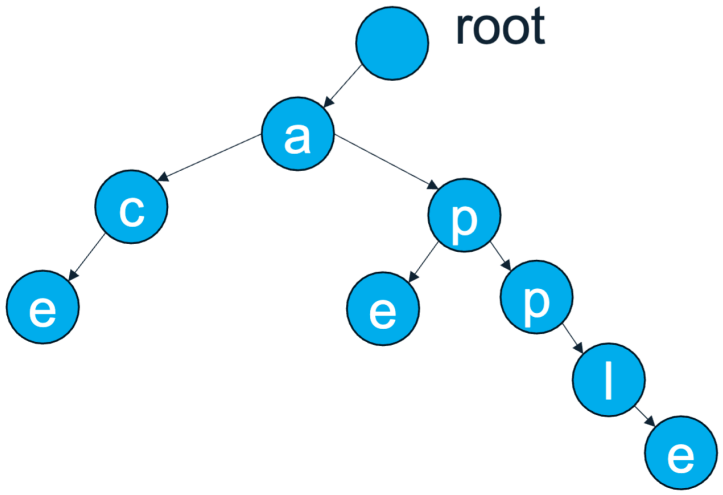
- Level Order (level by level) 1, 4, 2, 6, 5, 3
- In-order (left, root, right) 6, 4, 5, 1, 3, 2
- Pre-order (root, left, right) 1, 4, 6, 5, 2, 3
- Post-order (left, right, root) 6, 5, 4, 3, 2, 1



VI.2 BST Delete

- **0 child**: just delete
- **1 child**: set parent's pointer to point to the one child
- **2 children**: replace with in-order successor child (leftmost child of right subtree)

VI.3 Trie

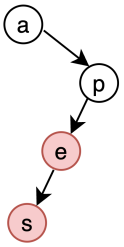


- Tries are a tree data structure that are particularly suited for searching for keys that begin with a specific prefix
- Usually, these keys are strings
- Each path from root represents one key

```
struct TrieNode {
  bool isEndOfWord;
  vector<TrieNode*> childern;
}
```

VI.3.1 Adding keys

- While the key forms a path in the trie, follow the trie while next key character matches the characters in trie
- Add remaining non-matching key characters to tree, mark node with last character as end of word
- Example: Insert “ape” and “apes” - nodes coloured red have `isEndOfWord == true`



VI.3.2 Deleting key

```
// If root is nullptr, key is not in the Trie
if (!root) return nullptr;
// If we've reached the end of the key
if (depth == key.length()) {
  // This node is no longer an end of a word
  if (root->isEndOfWord) root->isEndOfWord = false;
  // If node has no children, delete it (free memory) and return nullptr
  if (root->children.empty()) {
    delete root;
    return nullptr;
  }
  return root;
}
// Recursive case for deleting in child
char ch = key[depth];
root->children[ch] = deleteKey(root->children[ch], key, depth + 1);
```

```
// If no children and not end of word, delete this node
if (root->children.empty() && !root->isEndOfWord) {
  delete root;
  return nullptr;
}
return root;
```

VI.4 Quiz

VI.4.1
The height of a binary tree can not have a tight bound

VI.4.2
Running time of searching a Binary Search Tree with n nodes: $\Theta(\text{height})$

VI.4.3
The minimum number of nodes that could be in a balanced binary tree of height h: 2^h

VI.4.4
In a red black, we keep the same number of black nodes between a given node and any other node. The reason for this is: to ensure search is $\log(n)$

VI.4.5
For which of the following would a Trie be a good choice of Tree data structure
- Storing key-value pairs with string keys
- Storing a collection of unsorted strings
- Storing key-value pairs with integer keys
- Storing a collection of unsorted integers

VII Heap

VII.1 Binary Tree in array
- If a node is at index i
- Its left child is at $2 * i$
- Its right child is at $2 * i + 1$
- Its parent is at $\lfloor \frac{i}{2} \rfloor$

VII.2 Operations
Swap the node with the largest(smallest) child until the heap property is satisfied
- **Insert**: bottom to top
- **Delete**: top to bottom
- **Heapify**: bottom to top (back to front)

VII.3 Quiz

VII.3.1
- A min-heap is a binary tree
- In a min-heap node after deletion, the maximum levels that a parent will sift down during heapify is O(log(n))
- A heap can become an unbalanced tree

- When traversing by level a Heap implemented as a tree, the values will be in order from smallest to largest.

VII.3.2

When deleting the root node, which of the following should take place to restore the Heap

- sift up each of the leaf nodes
- copy the value of the last element to the root and heapify the root element
- copy the value of the last element to the root and delete the last element, heapify the root element.
- copy the value to the root and heapify each parent node starting from the bottom right parent working across tree to the left, level by level until the root node is heapified

VII.3.3

maximum depth of a Heap implemented as a Tree: $O(\log(n))$

VII.3.4

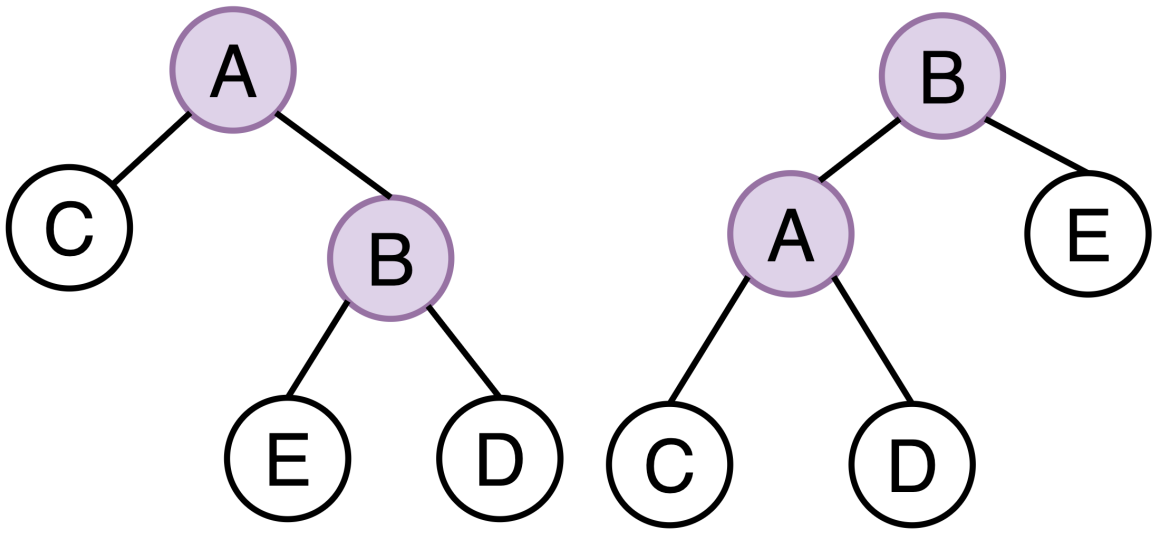
- Implementing a Heap as a vector will allow faster insertion deletion of elements compared to implementing as a Tree
- Implementing a Heap as a vector uses less memory compared to implementing as a Tree
- Implementing a Heap as a vector better matches the Heap abstraction compared to implementing as a Tree
- Implementing a Heap as a vector results in a lower run-time complexity for all operations compared to implementing as a Tree

VIII Red Black Tree

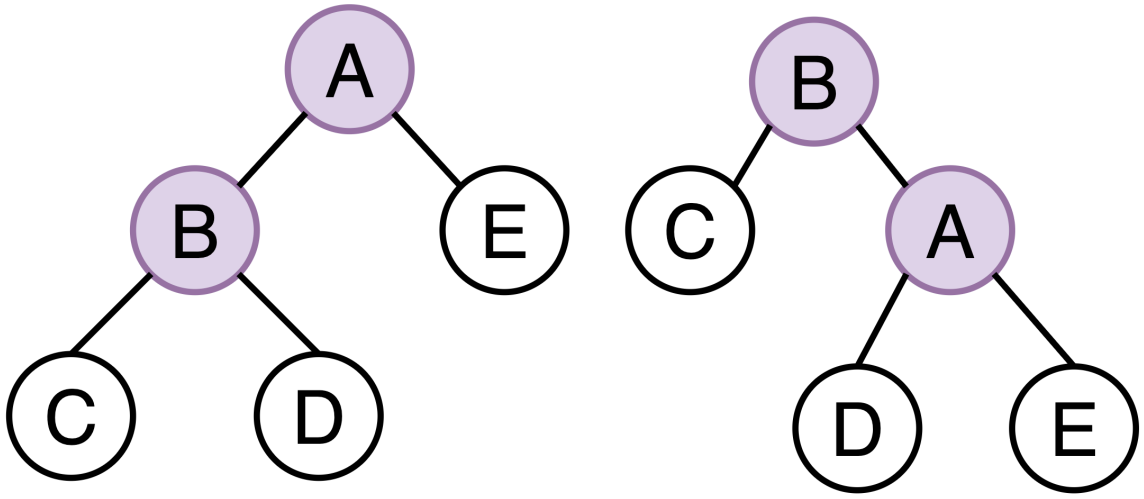
1. Root is always black
2. No two adjacent 临近的 nodes are red
3. Any path between a node and any descendant (lower) node 子孙 has the same number of black nodes

VIII.1 Rotate

Left



Right



VIII.2 Insert

VIII.2.1 Root

Change colour to black

VIII.2.2 Violated 2 & Uncle is red

- Change parent and uncle to black
- Change grandparent to red
- Make grandparent n, and repeat

VIII.2.3 Violated 2 & Uncle is black

VIII.2.3.1 Left Left

rotate right, swap colours of parent and grandparent

VIII.2.3.2 Left Right

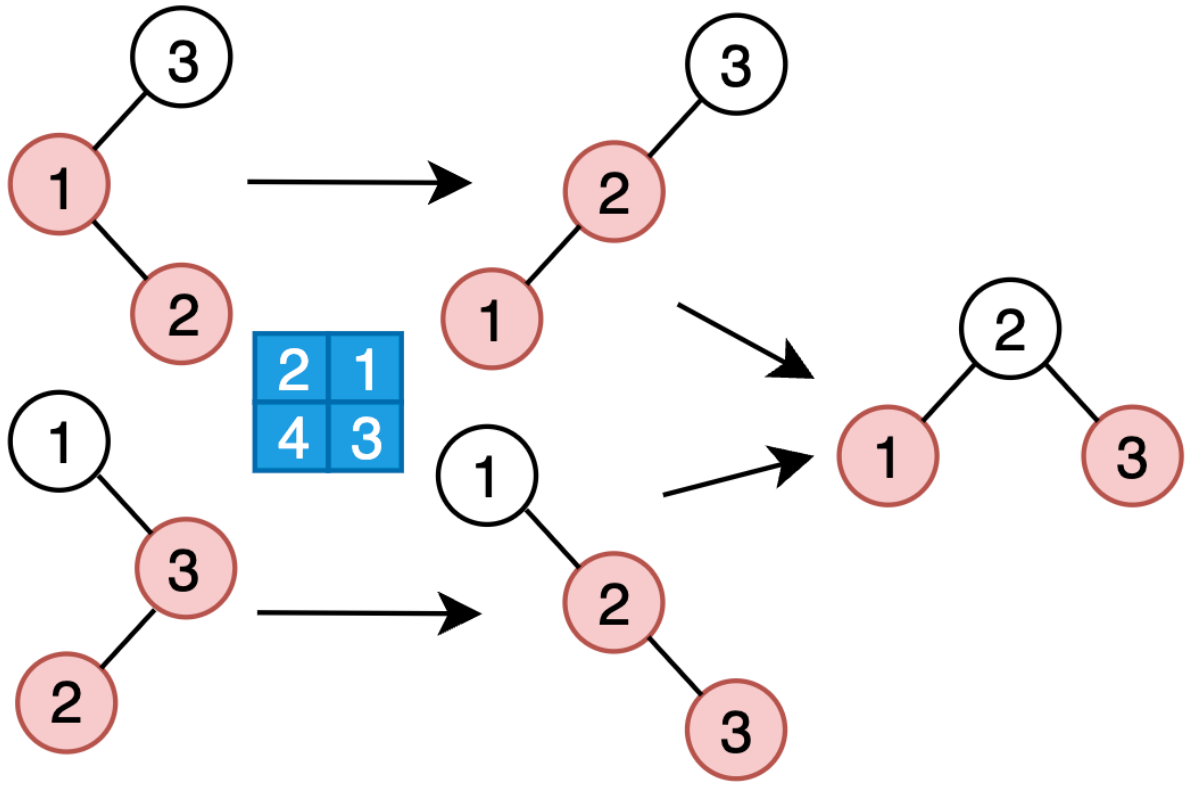
rotate left, then right, swap colours of new node and grandparent

VIII.2.3.3 Right Right

rotate left, swap colours of parent and grandparent

VIII.2.3.4 Right Left

rotate right, then left, swap colours of new node and grandparent



VIII.3 Delete

VIII.3.1 Simple Cases

- If a node is red with nullptr child (no children)
- If a node has 1 child and **either** the node OR child (but not both) is red

1. Delete node 2. Updated node → **black** (node replaced the deleted node)

VIII.3.2 Double Black

If both node to be deleted AND child are **black** (or the node has no children), the updated node becomes **double black**

- n's sibling is **black**
  - with at least one **red** child
    1. Rotate (as per insertion following path to red child)
    2. Recolour red child to **black**, sibling to red
  - with all children **black**
    1. Recolour sibling to red
    2. Push **black** up (**black** parent → **double black**, red parent → **black**)
- n's sibling is **red**
  1. Rotate
  2. Recolour Sibling to **black** Parent to red

IX Selecting Data Structures and Algorithmic Strategies Quiz

IX.1

- **Heap Sort**: Transform and Conquer
- **Memoisation**: Synamic Programming

- **Selection Sort**: Brute Force
- **Quick Sort**: Divide and Conquer

IX.2

You need to store a list of words and know that words that were recently searched for are likely to be searched for again. **ANSWER: Linked List**

$O(n)$ to find a word and when a word is found, move it to the head (most recent will be at front of list) $O(1)$. The most recently searched for words will always be at the start of the list, improving the average case to find these words $O(1)$ if most recently searched word.

IX.3

- **Linear Structures**: Queue, Linked List, Array, Heap, Stack, Vector
- **Non-linear Structures**: Red-Black Tree, Binary Search Tree, Trie

IX.4

Which of the following C++ containers could be shuffled using the C++ shuffle function from the algorithm library? **ANSWER: vector, array**

IX.5

- `list`: Doubly Linked List
- `priority_queue`: Heap
- `map` `set`: Red-Black Tree (balanced binary search tree)

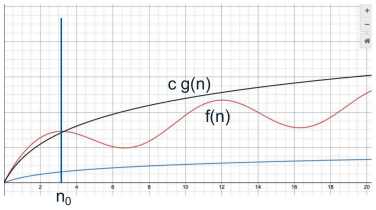
IX.6

We are implementing a scheduler for choosing the next process to run on a computer. System processes, that keep the computer running are more important than user processes (someone's C++ homework); but otherwise processes should be run on a first come first served basis Assuming we decide to keep all of these processes in a single ordered array, which sorting algorithms could be used (assume the base algorithms as you learned in this course, not variations). **ANSWER: Bubble Sort, Insertion Sort, Merge Sort**

Big O – Upper bound

We say that $f(n) = O(g(n))$ iff (if and only if)

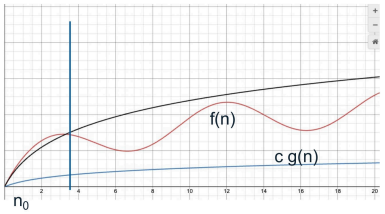
$\exists c \in R^+, \exists n_0 \in N,\ such\ that\ \forall n \geq n_0 : f(n) \leq cg(n).$



Big Ω– Lower bound

We say that $f(n)$ is in $\Omega(g(n))$ iff (if and only if)

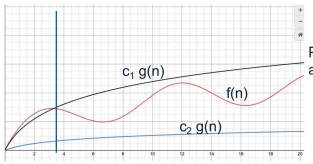
$\exists c \in R^+, \exists n_0 \in N,\ such\ that\ \forall n \geq n_0 : f(n) \geq cg(n).$



Big Θ Tight bound

$f(n)$ =Θ$(g(n))$ iff (if and only if):

$f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$.



Polynomials of degree k are in $\Theta(n^k)$:
$a_k n^k \leq a_k n^k + \ldots + a_1 n + a_0 \leq (a_k + \ldots + a_1 + a_0) n^k$

Little o Upper bound

$\forall c \in R^+, \exists n_0 \in N, such\ that\ \forall n \geq n_0\ such\ that\ f(n) < cg(n)$ $(\left|\frac{f(n)}{g(n)}\right| < c)$.

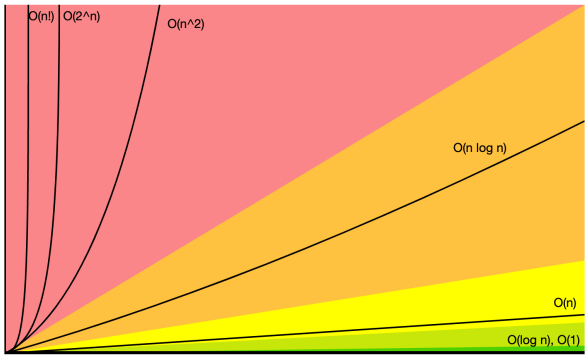
In other words:
$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Examples:

If f(n)=o(g(n)) then f(n)=O(g(n)).

$n = o(n^2)$.
$\log n = o(n)$.

However, Little o does not allow the same growth rate.



	Best	Average	Worst	Space	Stable
Selection	$O(n^2)$			$O(1)$	
Insertion	$O(n)$	$O(n^2)$		$O(1)$	✓
Bubble	$O(n^2)$			$O(1)$	✓
Merge	$O(n \log(n))$			$O(n)$	✓
Quick	$O(n \log(n))$		$O(n^2)$	$O(\log(n))$	

Bucket	$O(n)$	$O(n+k)$ (k: numbers of buckets)	$O(n^2)$	$O(n+k)$	✓
Counting	$O(n+k)$ (k: range of the input values, max-min)			$O(n+k)$	✓
Heap	$O(n \log(n))$			$O(1)$	

	Worst			Average			Space
	Insert	**Delete**	**Search**	**Insert**	**Delete**	**Search**	
Vector Ordered	$O(n)$		$O(\log(n))$	$O(n)$		$O(\log(n))$	$O(n)$
Vector Unordered	$O(1)$*	$O(n)$		$O(1)$*	$O(n)$		
Linked List	$O(1)$			$O(1)$			
Binary Search Tree	$O(n)$			$O(\log(n))$			
Balanced BST (RBT)	$O(\log(n))$						

Priority Queues	**Insert**	**RMH****	**Peek**	**Insert**	**RMH****	**Peek**	
Linked List	$O(n)$	$O(1)$		$O(n)$	$O(1)$		
Heap	$O(\log(n))$		$O(1)$	$O(1)$	$O(\log(n))$	$O(1)$	

*: Amortised - ie over a sequence of this operation. Resizing vector $O(n)$
**: Remove highest priority