SIT221: DATA STRUCTURES & ALGORITHMS

LECTURE #11: REVISION

26th September, 2016

Agenda

- Data structures
- Algorithm design paradigms
- Sample Exam

Data structures - what do we care about?

- Memory complexity/overhead
- Running time to perform key operations
  - Access an element
  - Insert an element
  - Delete an element
  - Search for an element

Arrays

- Stores linear data of similar type
  - Good: Random access – O(1) – using element index – integer
  - Good: Cache locality – elements are stored next to each other.
  - Good: No storage overhead.
  - Bad: Preset size – memory allocated before we can start using the array
  - Bad: Insert/delete are expensive O(n) – shift elements to the left/right
  - Bad: Search is expensive, unless it's sorted – linear O(n)

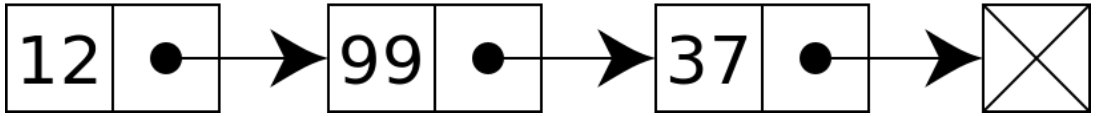
Lists/Vectors/Dynamic Arrays

- One way to overcome fixed array size problem
- Pre-allocated memory, but resized as/when needed
- Removed the headache of resizing array manually
- Still same insert/delete problem

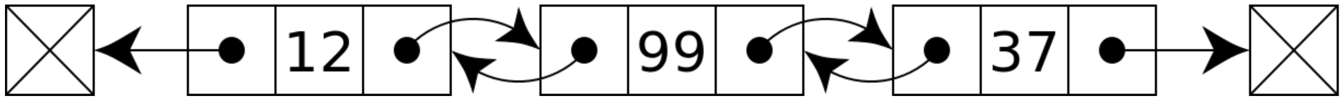
Linked List

- Another way to overcome fixed array size problem
- Allocate memory as/when needed
- Good: insert/delete is easy now – O(1)
- Bad: no random access – only linear access
- Bad: search is still expensive – O(n)
- Bad: extra memory overhead

Singly linked lists

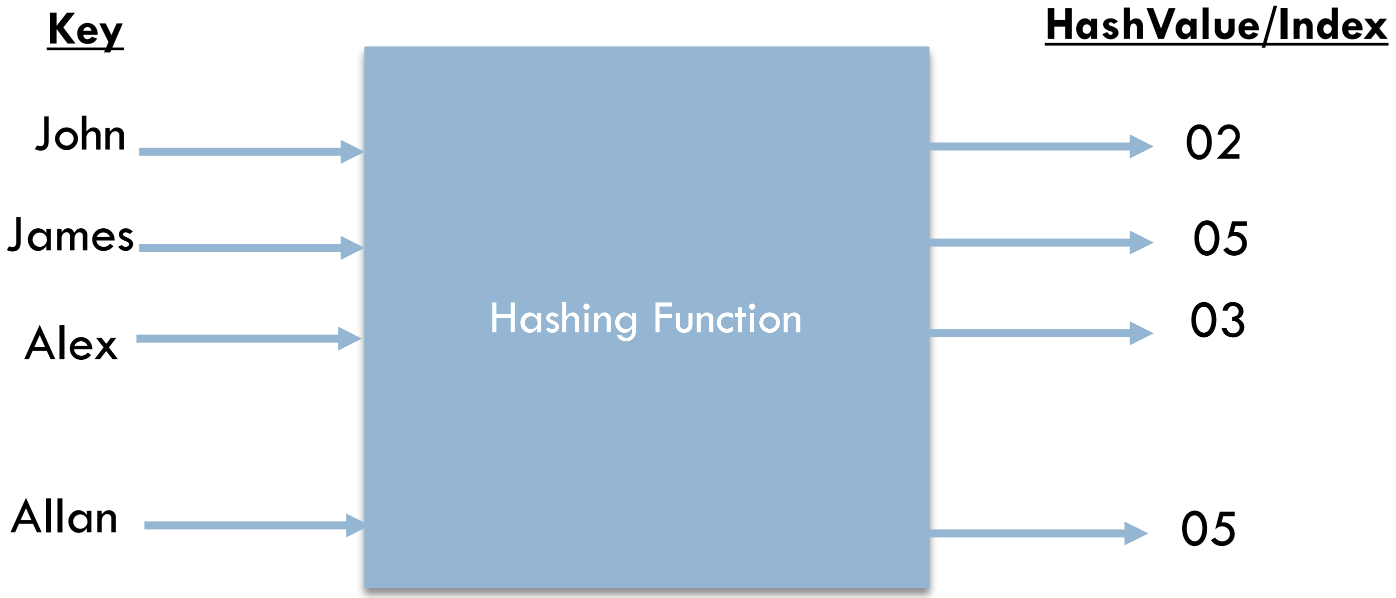


Doubly linked lists



Dictionary/Hashtable

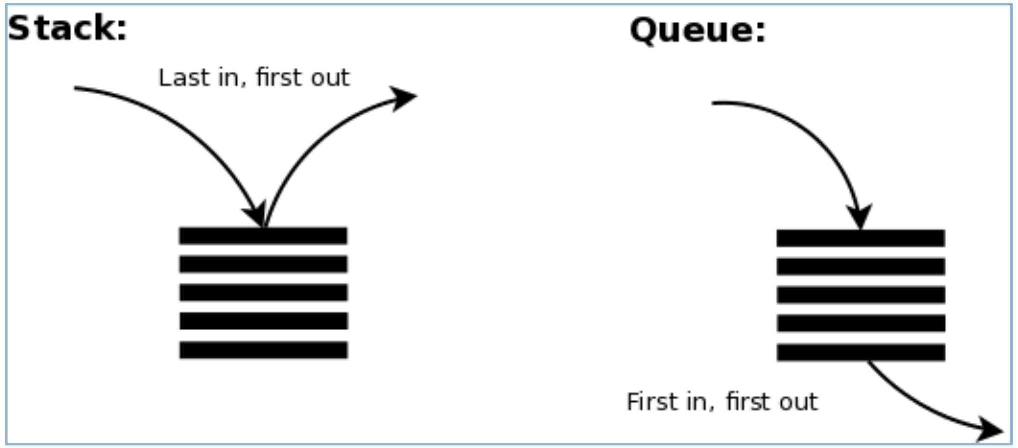
- Stores data as entries of <Key, Value> - **key** could be a string, int, or object. Similarly **Value** could be string, int, or an object.
- Internally the key is translated to entry index using hash function – **collision problem**



00	Jun, Jun data
01	Dave, Dave data
02	John, John data
03	Alex, Alex data
04	
05	James data Or Allan data ??? **Collision!!!**
06	
07	

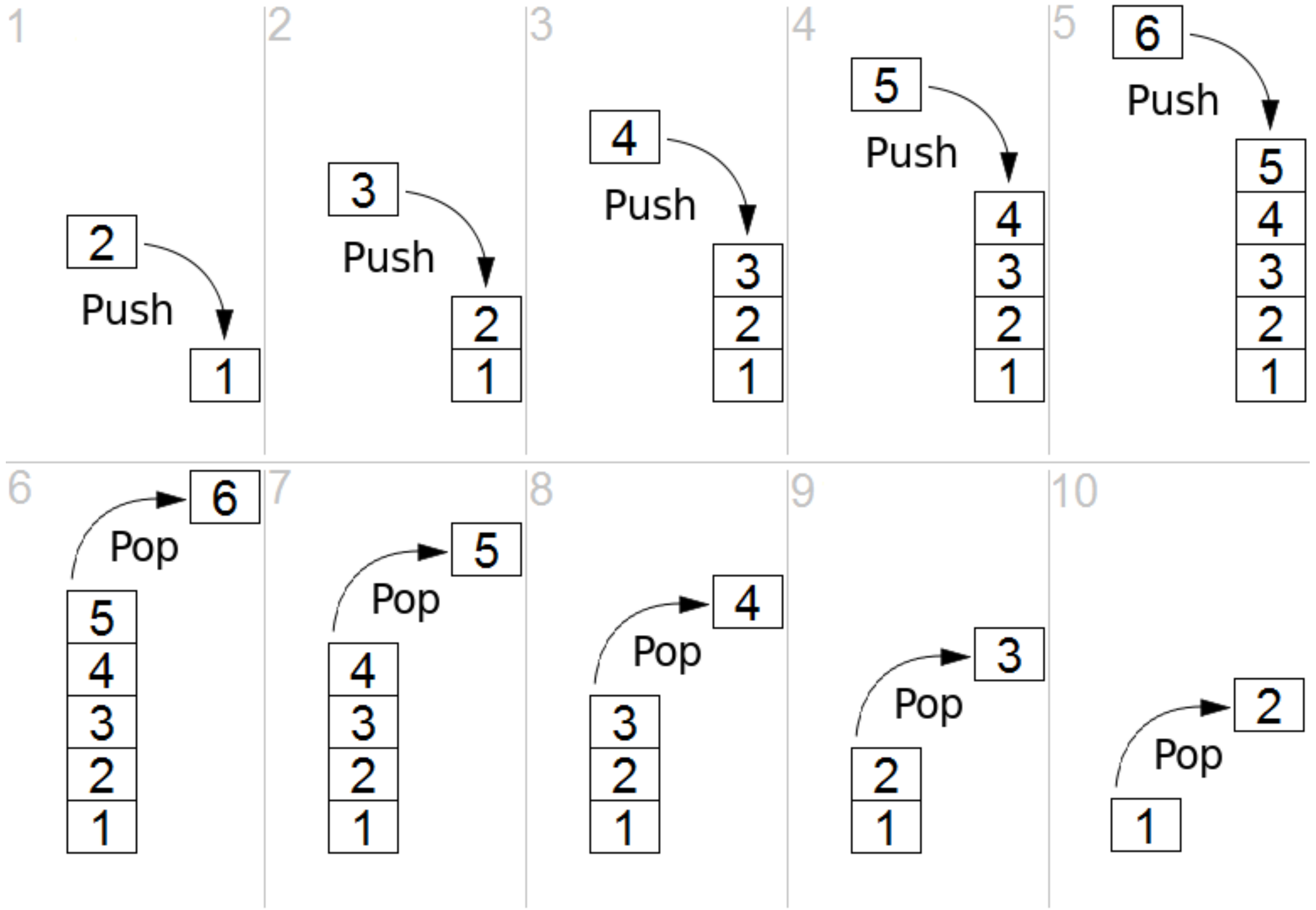
Stacks & Queues

- Stack = Last-in First-out (LIFO) or FILO
  - Push an element: adds an element at the top of the stack
  - Pop an element: removes & returns the element at the top of the stack
- Queue = First-in First-out (FIFO) or LILO
  - Enque an element:
    adds an element to the end of the queue
  - Deque an element:
    removes the element at the front of the queue



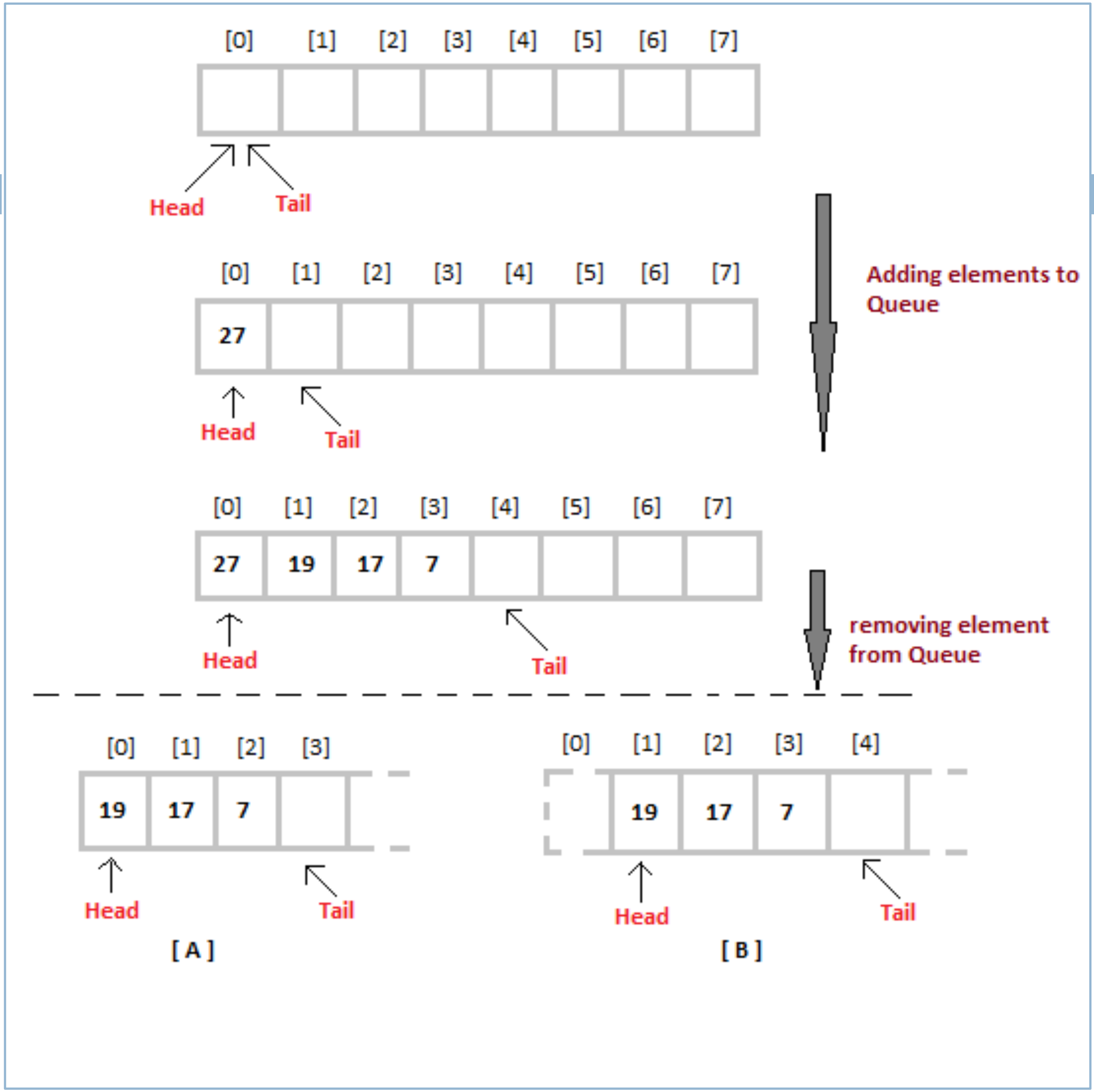
How stacks work?

- Stack has 1
- Push(2)
- Push(3)
- Push(4)
- Push(5)
- Push(6)
- Pop() ---> ?



How Queues work?

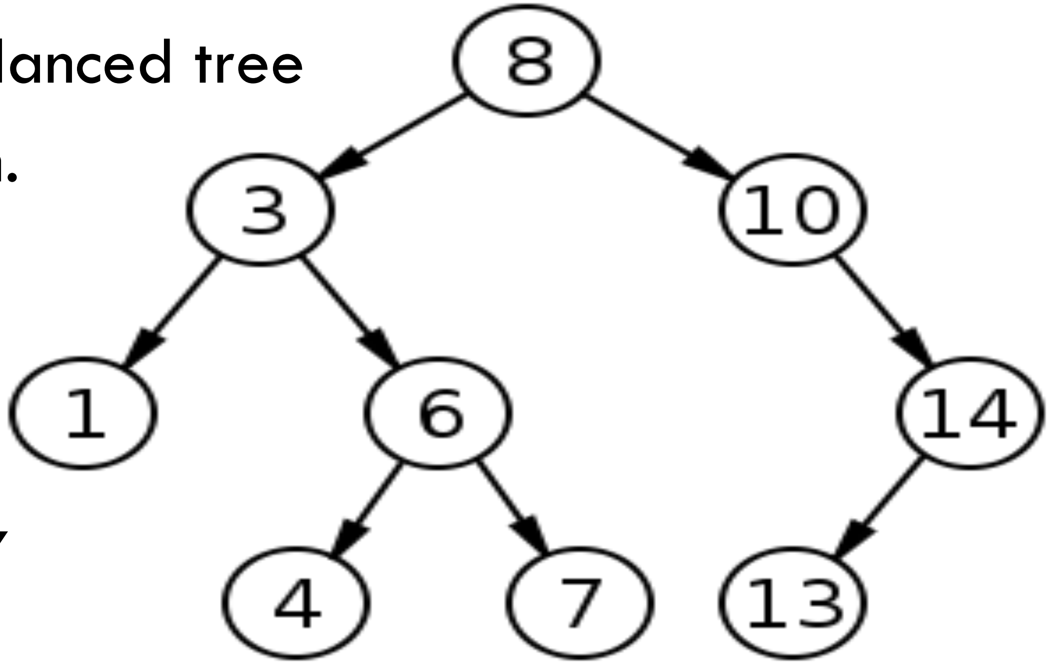
- Start with empty queue.
- Enque(27)
- Enque(19)
- Enque(17)
- Enque(7)
- Deque() --> ?
- Deque() --> ?



Trees

- We use trees to represent hierarchical data.
  - Binary Search tree is a special type of trees – each node has two children.
  - Left child nodes should be less than the parent, and right child nodes are greater.
  - Problem: if data sorted, will lead to unbalanced tree
  - AVL/Red black trees address this problem.
  - Good: insert/delete/access O(logn)
  - Traversal – Preorder Post-order, In-order

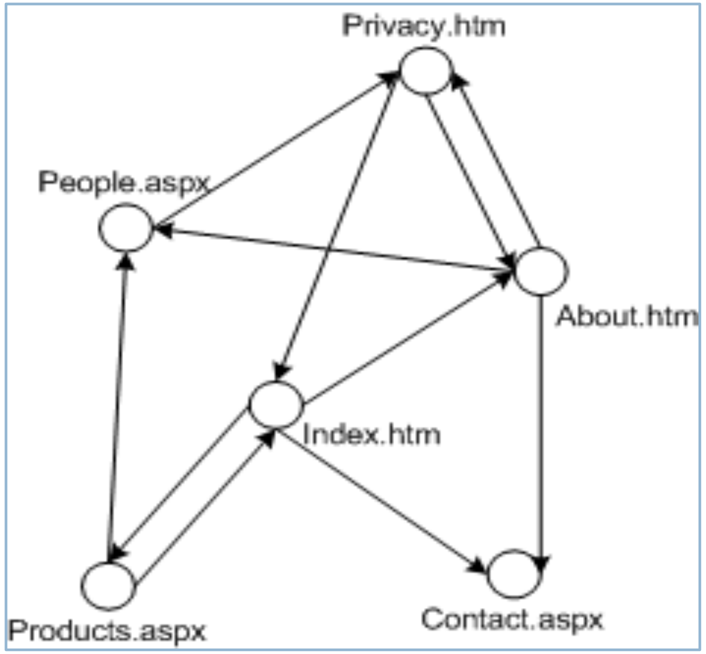
8, 10, 14, 3, 1, 6, 4, 13, 7

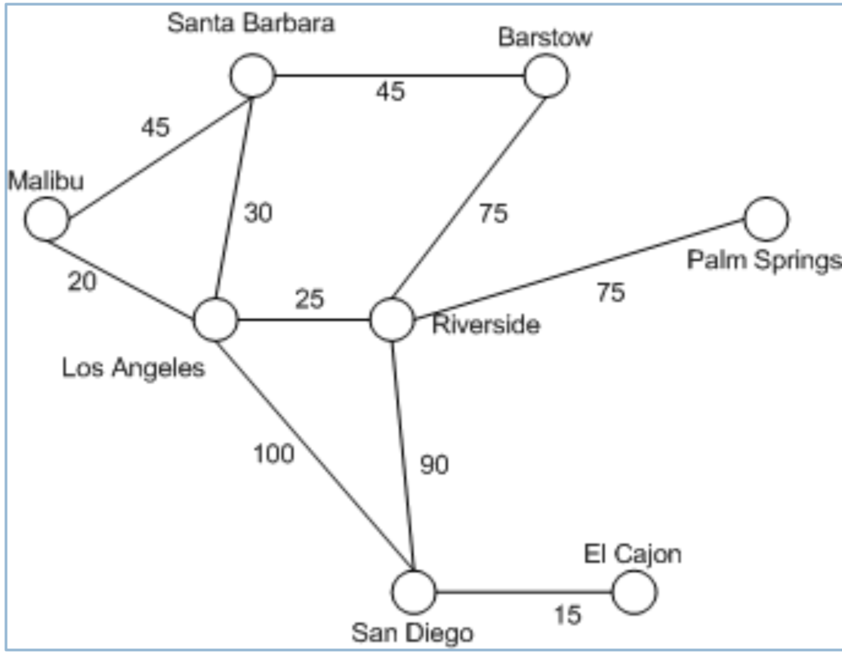


Graphs

- Graphs are represented as G=(V, E):
  - Vertices [Pages / Nodes / Individuals / …]
  - Edges[Navigation from page to another, route from a city to another, …] ==> we call these links as edges

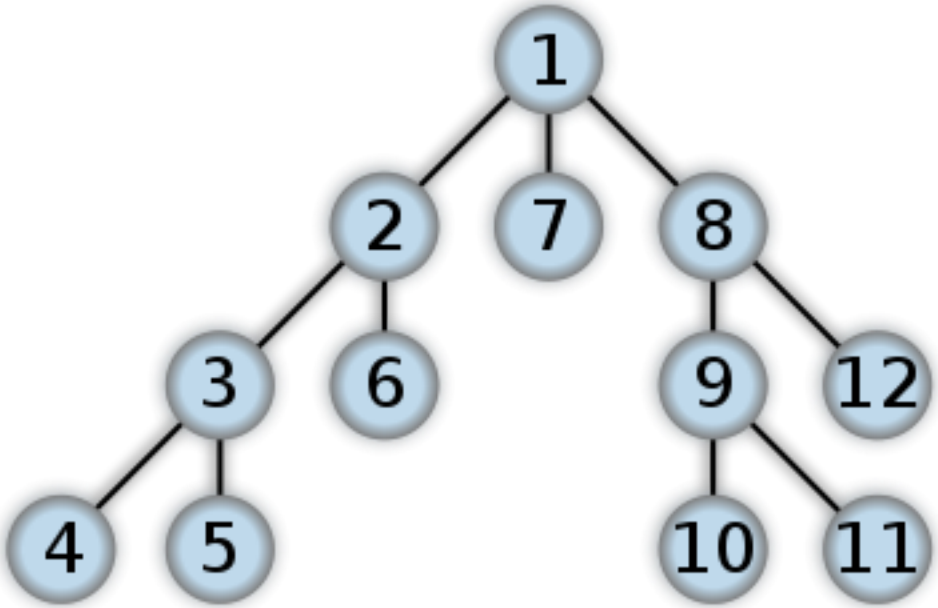
- Thus, any graph can be described as: Vertices & edges
- **G=(V,E)**



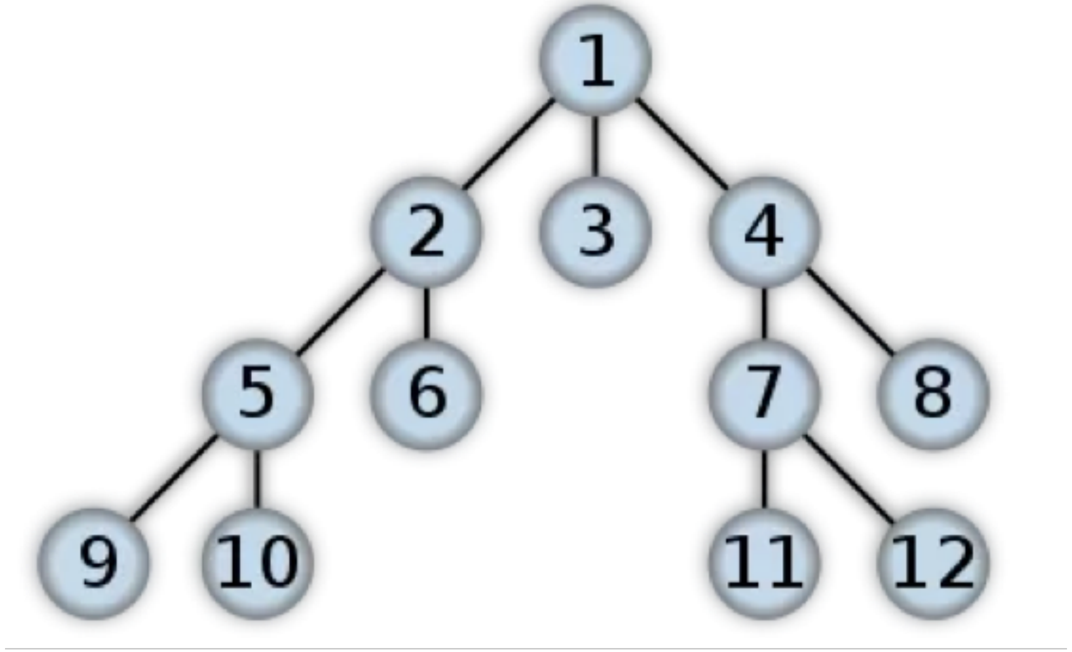


Graph/Tree Traversal

Depth-first search



Breadth-first search



Data structure selection criteria

- What data you want to store/process?
- What operations you expect to perform?
- Which of them is critical & affects the overall performance of the solution?
- Do you have any memory constraints?

Data structures cheat sheets

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

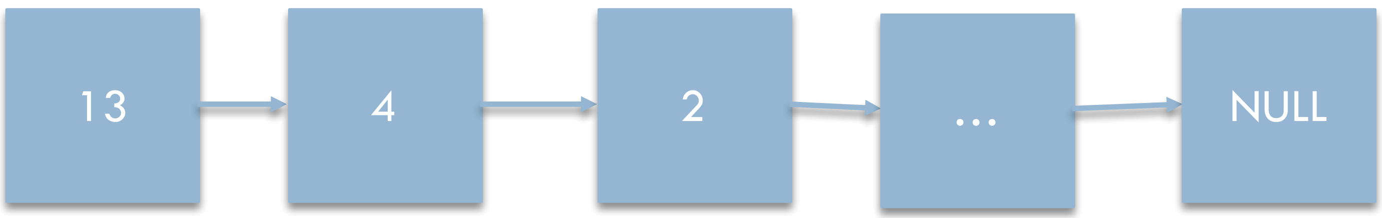
Algorithm design paradigms

- These are techniques that we can use to solve a given problem.
  - Brute Force
    - Find a solution regardless of the efficiency of the solution
  - Reduce to a known problem
    - Reuse a combination of existing solutions to simpler problems
  - Divide & Conquer
    - Break the problem to smaller problems (similar in nature to the original problem), solve sub-problems and merge solutions to solve the bigger problem.
  - Dynamic Programming
    - Similar to D&C, but walks the solution bottom-up, so we can memorize solutions of smaller problems.
  - Greedy Algorithms
    - At every step in our algorithm to solve the problem, we choose the best available solution.

Data structures in use

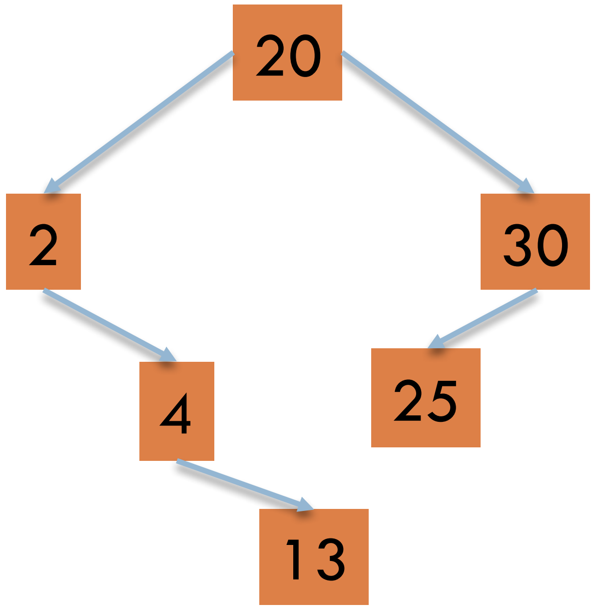
- Input data: [13, 4, 2, 25, 30, 20]

Linked List



Stack

20
30
25
2
4
13



In order: 2,4, 13, 20, 25, 30

Post-order: 13, 4, 2, 25, 30, 20

Data structures in use

Post-order: 13, 4, 2, 25, 30, 20 ---- key = number % 10

Key	Value
20 % 10 = 0, 30 % 10 = 0	30, 20
2%10 = 2	2
13 % 10 = 3	13
4%10 = 4	4
25%10 = 5	25

Programming question

2	74	8	19	20	30	25	38	31	35	70	76

Return dictionary

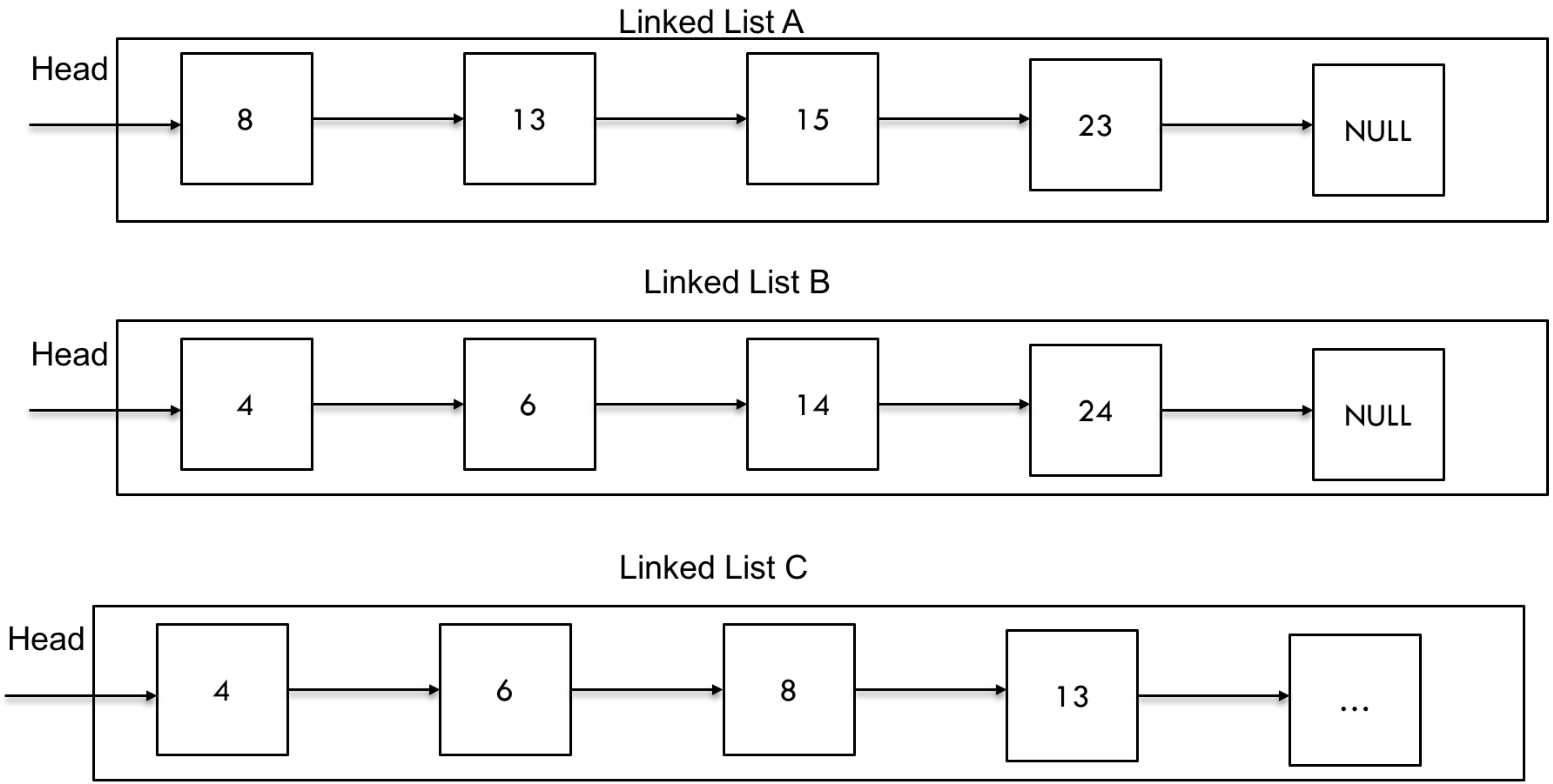
Key [Range]	Value [count/how many elements]	Why?
0	2	2 & 8 when integer divided by 10 give zero
1	1	Only 19, when integer divide by 10 gives 1
2	2	20 & 25 when integer divide by 10 give 2
3	4	30, 31 35, 38, all when integer divide by 10 give 3
7	3	74, 70, 76, all when integer divide by 10 give 7

Implementation of GroupBy10s


```
public Dictionary<int, int> GroupBy10s(Vector<int> data)
        {
            Dictionary<int, int> result = new Dictionary<int, int>();
            for (int i = 0; i < data.Count; i++)
            {
                int key = data[i] / 10;
                if (result.ContainsKey(key) == false) result.Add(key, 1);
                else result[key] += 1;
            }
            return result;

        }
```

Merge Two Sorted Linked Lists



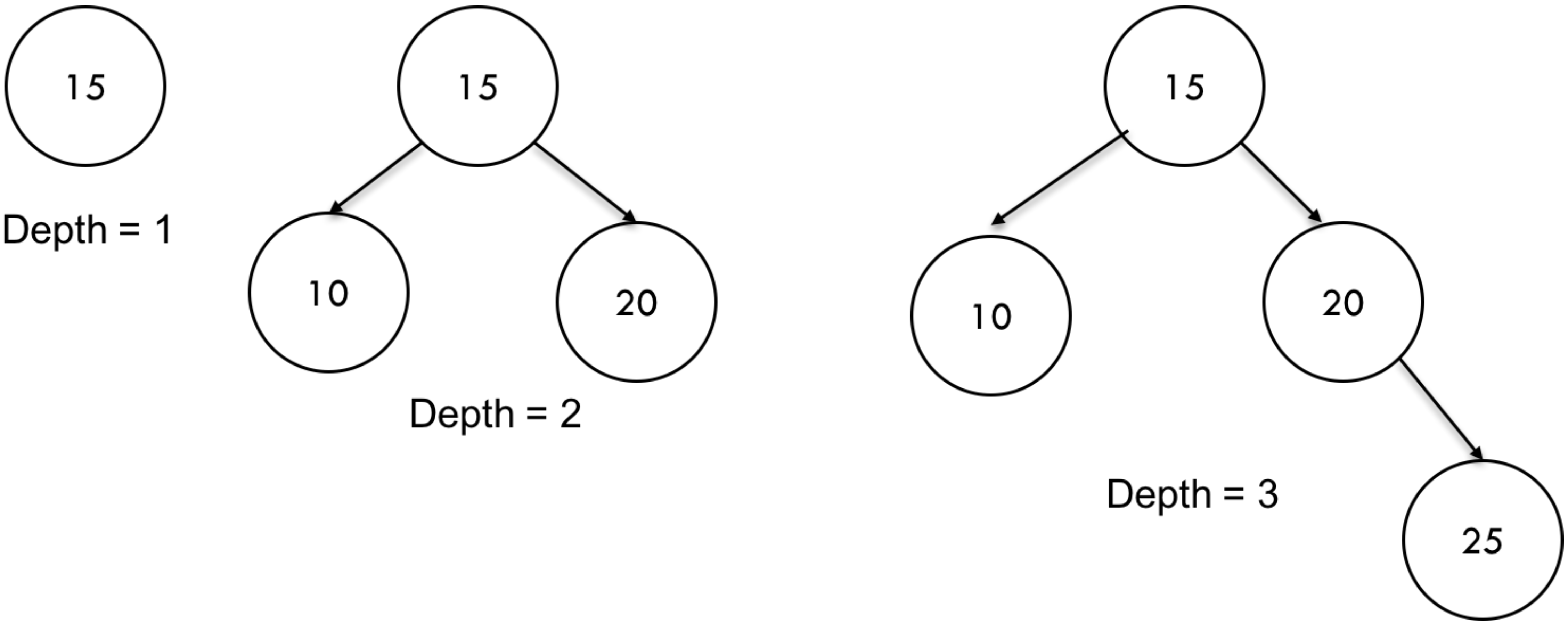
Implementation of MergeSortedLinkedLists


```
public LinkedList<T> MergeSortedLinkedLists(LinkedList<T> B)
{
            LinkedList<T> C = new LinkedList<T>();
            int aIndex = 0, bIndex = 0;
            var comparer = Comparer<T>.Default;
            while (aIndex < this.Count || bIndex < B.Count)
            {

                if (bIndex >= B.Count && aIndex < this.Count) {
                    C.Add(this.ElementAt(aIndex));
                    aIndex++;
                }
                else if (aIndex >= this.Count && bIndex < B.Count)
{
                    C.Add(B.ElementAt(bIndex));
                    bIndex++;
                }
```

```
else if(comparer.Compare(this.ElementAt(aIndex), B
.ElementAt(bIndex)) < 0) {
                    C.Add(this.ElementAt(aIndex));
                    aIndex++;
                }
                else {
                    C.Add(B.ElementAt(bIndex));
                    bIndex++;
                }
            }
            return C;
        } //End of your method
```

Tree depth



Tree depth


```
public int Depth(BSTNode<T> node)
        {
            if (node == null) return 0;
            var leftDepth = Depth(node.LeftChild);
            var rightDepth = Depth(node.RightChild);

          if(leftDepth > rightDepth ) return leftDepth + 1;
          else return rightDepth + 1;

  } //End of your method
```

RestaurantX => O(n * m), n number of groups m is number of table categories


```
public void ServeGroups()
{
    var count = WaitingArea.Count;

    while( count > 0 )
    {
        var group = WaitingArea.Deque();
        count--;

        int suitableTable = GetSuitableTable(group);
        if (suitableTable == -1) WaitingArea.Enque(group);
        else
        {
            TablesCapacity[suitableTable] =
            TablesCapacity[suitableTable] - 1;
            Console.WriteLine(" group of size" + group + " sit on table " + suitableTable);
        }
    }

}
```

RestaurantX


```
int GetSuitableTable(int group)
    {
        int bestTable = MAXCAPACITY;
        foreach (KeyValuePair<int, int> entry in TablesCapacity)
        {
            if (entry.Value > 0
                && entry.Key >= group
                && entry.Key < 2 * group)
            {
                if (entry.Key < bestTable) bestTable = entry.Key;
            }
        }

        if (bestTable == MAXCAPACITY) return -1;
        return bestTable;
    }
```

What else did we cover?

- .NET Collection classes
- Interfaces: IEnumerable, IEnumerator, ICollection, IList, Icomparable, IComparer
- Sorting and searching techniques
- Unit testing
- LINQ

Where to from here?

- Design and architecture patterns
- Software engineering [SDLC]
- Web and/or mobile app development
- Database programming
- Unit testing
- Github + Trello (or JIRA) – online existence [blogs/videos/tutorials/etc.]
- Practice, practice, practice

Good luck!