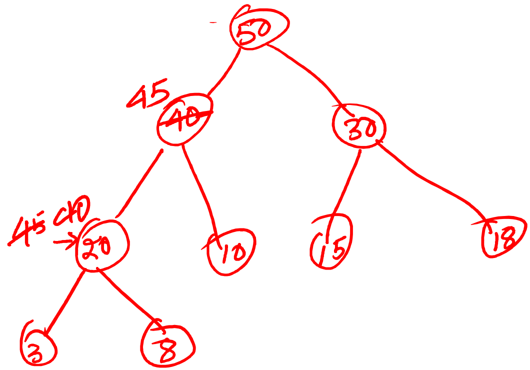
Today’s topics

- Priority queue
- Tries
- Graphs

Exercise

- Max_Heap_Increase_Key(H, i, keyval): Increase key at index i to a new value (keyval) in the max heap H



In the Max_Heap_Increase_Key operation, we replace the value of the key at some index i, with a larger value. After increasing the key, the final tree might not be a max heap. To ensure the max heap property, we can walk all the nodes from nodes in the path from i to the root and swap values until we reach a parent node that is larger than or equal to its child node or we reach the root, similar to insert.

Priority queues

Priority Queues

- There are two types of priority queues
  - max-priority queue
  - min-priority queue

Priority queues

Each element in a priority queue has a priority. In a max priority queue, a larger value means high priority. The elements are inserted in the priority queue using the enqueue operation, but when we delete an element, we always delete a node with the highest priority. In a min priority queue, a smaller value indicates higher priority.

Priority scheduler

- Let’s look at the problem of application scheduling in which every application has a priority associated with it

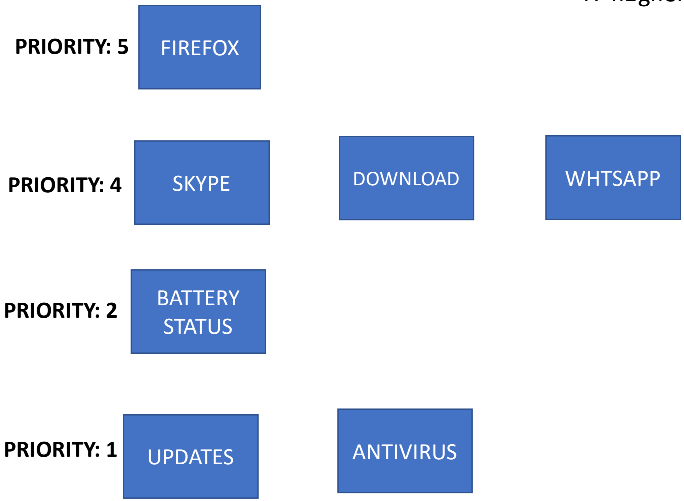
- A new application can come at any time

- We always schedule an application with the highest priority

- Which data structures can be efficient for this problem?

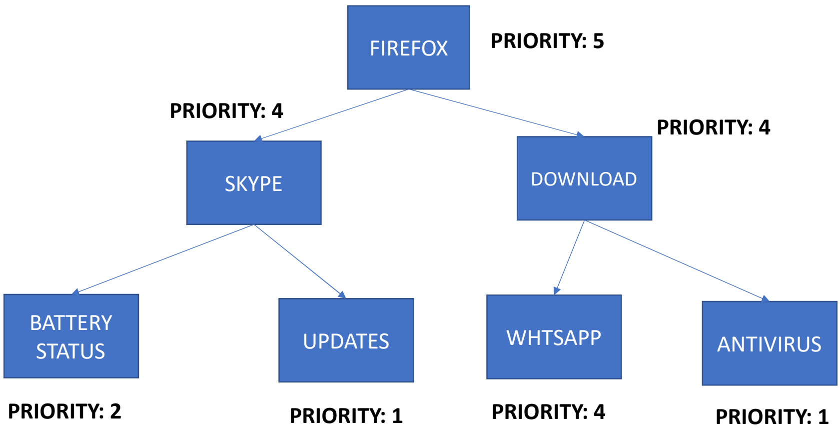
Priority process scheduling

A higher value means high priority.



We can use a max-heap to implement the priority scheduling. If we want some ordering among applications with the same priority, we can keep additional information in a node, e.g., the arrival time of an application (which can be implemented using a counter). We can use arrival time to decide which application to prioritize among the applications with the same priority.

Priority process scheduling



Max-priority queue ADT

- Greater value means higher priority

- Insert(S, x) : inserts an element x to set S
- Maximum(S): returns the element of S with the largest key
- Extract-Max(S): removes and returns the element with largest key
- Increase-Key(S, i, k): increases the key value of the ith element to k

- We can use a max heap to implement max-priority queue

Min-priority queue ADT

- Smaller value means lower priority

- Insert(S, x) : inserts an element x to set S
- Minimum(S): returns the element of S with the smallest key
- Extract-Min(S): removes and returns the element with smallest key
- Decrease-Key(S, i, k): decreases the key value of the ith element to k

- We can use a min heap to implement min-priority queue

Reference book

- Algorithm Design, Goodrich and Tamassia

Tries

Tries

- Read chapter-9.2 from Goodrich and Tamassia
- https://en.wikipedia.org/wiki/Trie

Tries

- An alphabet $\sum$ is a set of characters, e.g., (a, b, c, …, z)

- Let S be the set of n strings from the alphabet $\sum$, such that no string in S is a prefix of another string in S
  - A trie can be used to search a string of length L in O(L) operations from S

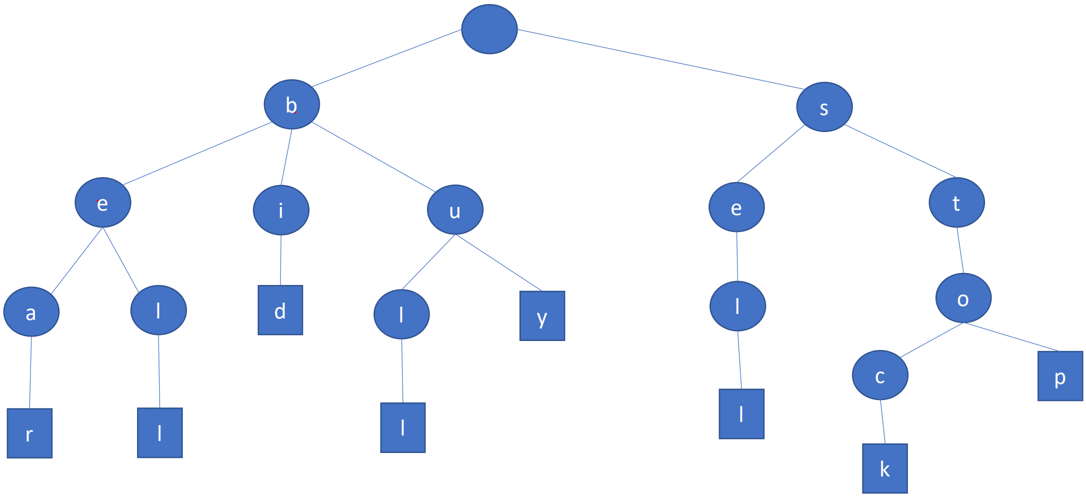
Tries

- What is the time complexity of searching a string of length L from a sequence of n strings stored in an AVL tree?

$O(L \log(n))$

Tries

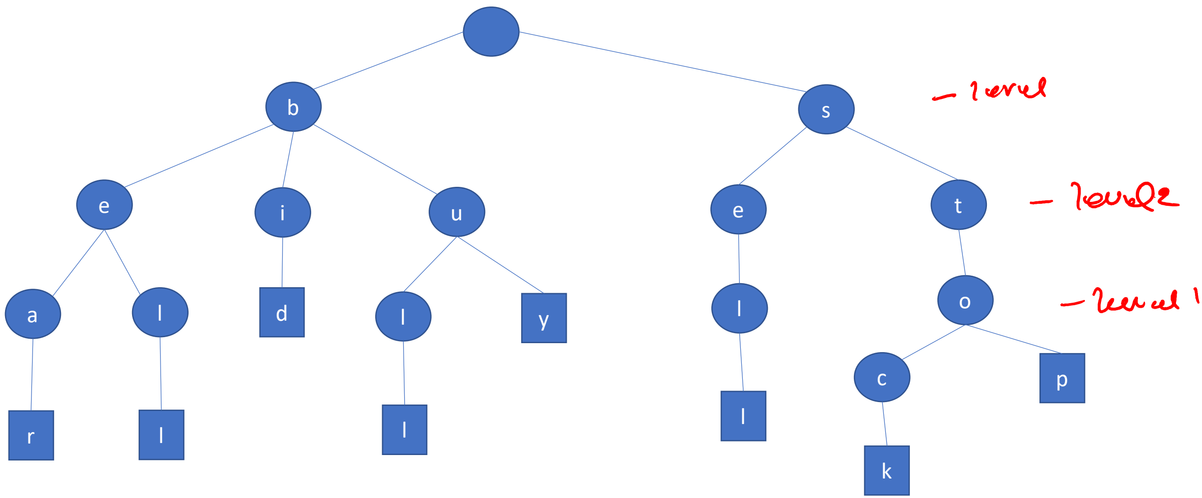
`{bear, bell, bid, bull, buy, sell, stock, stop}`



A trie can be visualized as follows. Lets S be the set of all strings in the trie. S1 contains the set of unique characters at the starting position in the strings in S. If n1 is the size of S1, then there will be n1 children of the root corresponding to the characters in S1. Notice that S1 contains only two characters, b and s. The subtree rooted at child b of the root contains nodes corresponding to all strings that start with b. Let S2 be the set of substrings of all strings in S that start with b after skipping the starting b. If S3 is the set of unique characters at the starting position of the strings in S2 and n3 is the number of elements in S3, then there will be n3 children of node b corresponding to the characters in S3. Notice that S3 contains three characters e, i, and u. The subtree rooted at child e of b contains nodes corresponding to all the strings that start from string “be”.

Tries

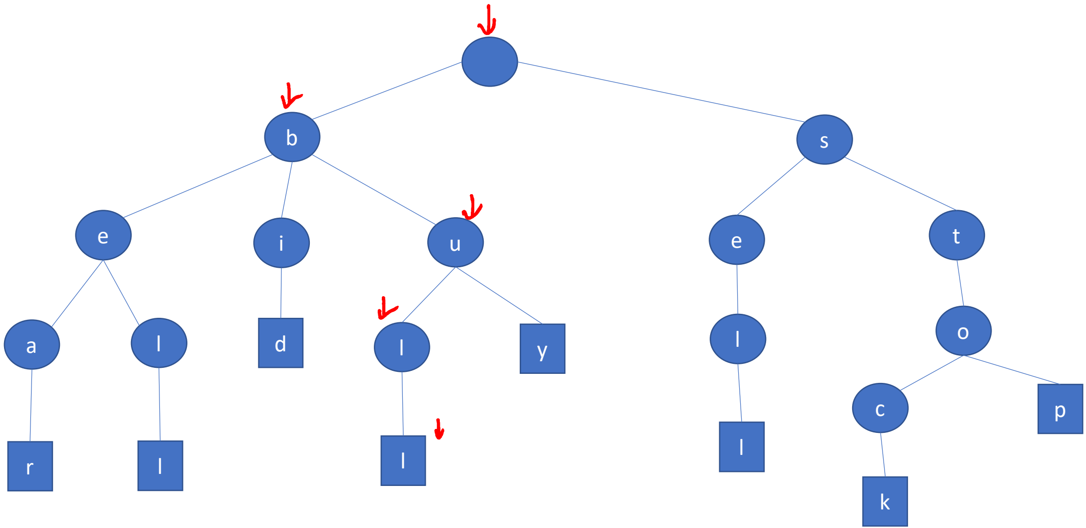
Search “stone”



To search a word stone, we first check if s is a child of the root; if yes, we check if t is a child of s; if yes, we check if o is a child of t; if yes, we check if n is a child of o; if yes, we check if e is a child of n; if yes, we check if e is a leaf (external) node; if yes then “stone” is present in the trie. If the answer to a check is no at any point, then “stone” is not present in the trie. All the square nodes are external nodes representing the last character of a word stored in the trie.

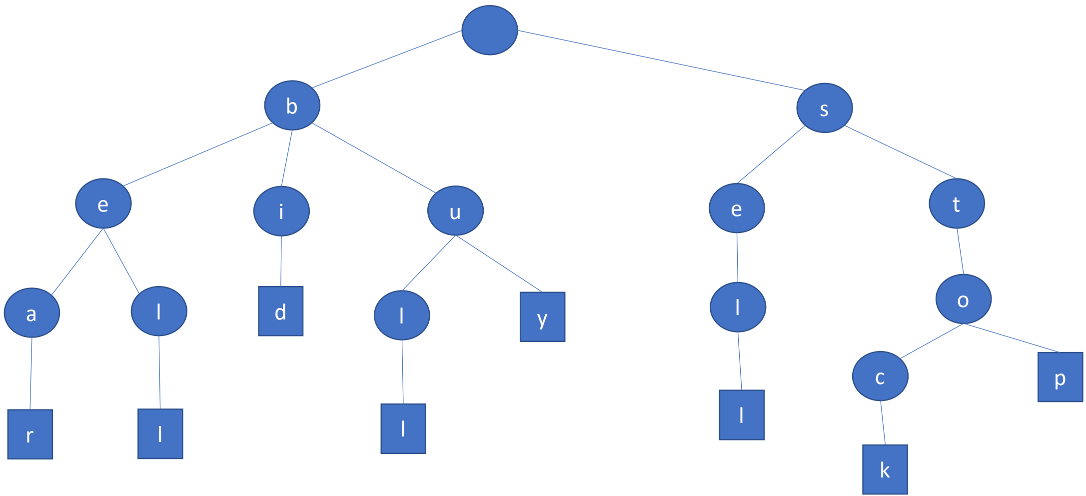
Tries

Search "bull"



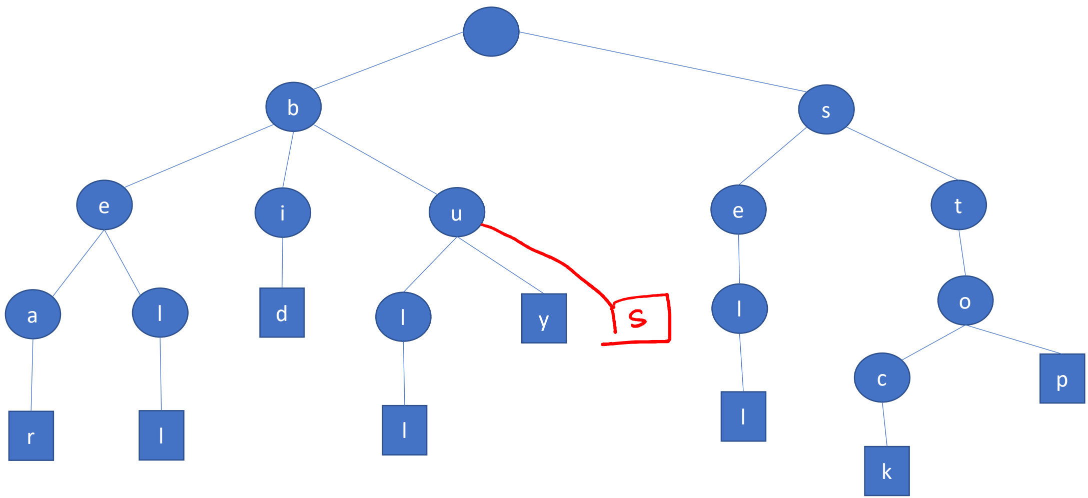
Tries

Search “bea”



Tries

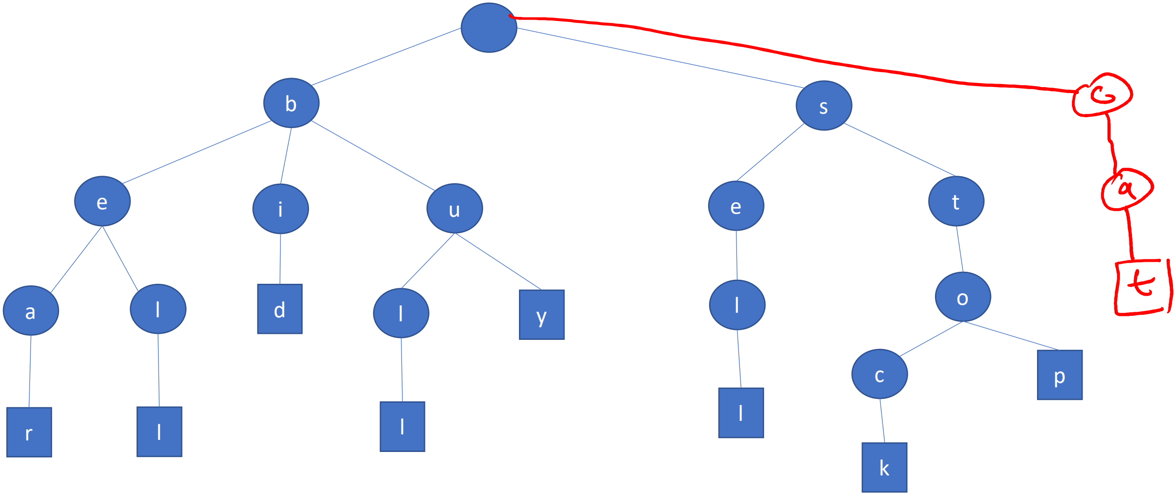
Insert "bus"



To insert "bus", we check if b is a child of the root. Because the answer is true, we move to b. We now check if u is a child of b. Because the answer is true, we move to u. We now check if s is a child of u. Because this is not true, we create a new node containing s. Because s is the last character of the input string, we make this node a square (external) node.

Tries

Insert “cat”



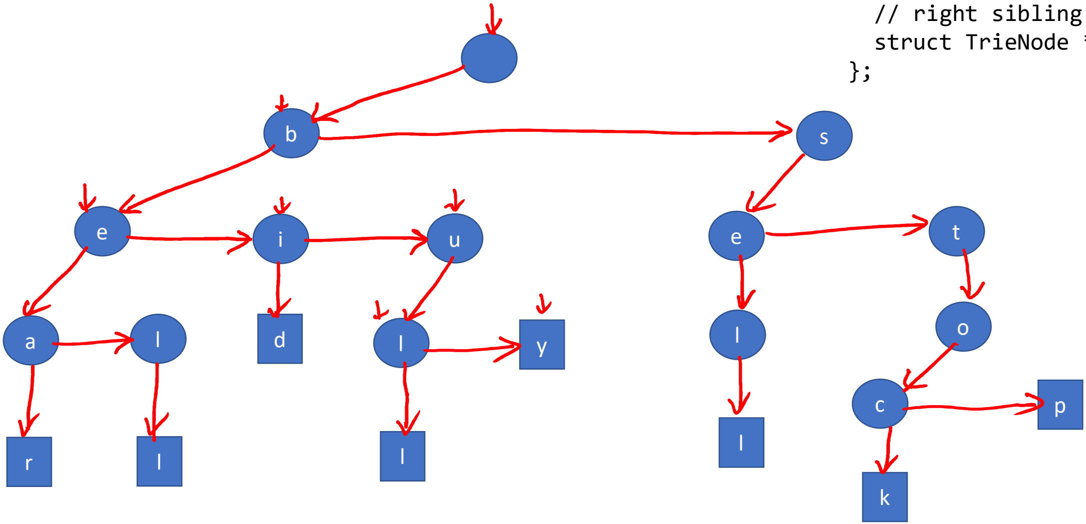
Tries

- A standard trie, T, storing n strings from an alphabet of size d, has the following properties
  - Every internal node of T has at most d children
  - T has n external nodes
  - The height of T is equal to the length of the longest string in S

Linked-list

s t o p
2 2 1 2
26 26 26 26

```
struct TrieNode {
   char val;
   // first child
   struct TrieNode *child;
   // right sibling
   struct TrieNode *next;
};
```



We can store the children of a node in a linked-list. A node contains a reference to its first child. A node also contains a reference to its right (or next) sibling. The resulting tree looks something like this.

External node

- How do we identify an external node?

child is NULL

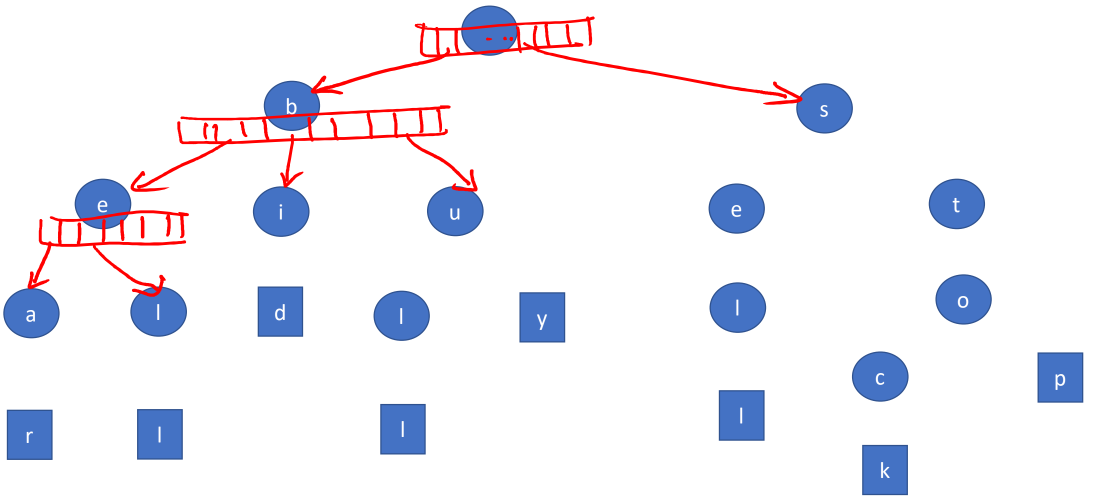
Time complexity

- Searching a word of length L $O(L \times d)$ d is size of alphabet
- Inserting a word of length L $O(L \times d)$ d is size of alphabet

At every level, we may need to walk the entire linked list to search for a particular child. The maximum number of nodes in a linked list can be d, where d is the size of the alphabet.

Array

```
struct TrieNode {
  bool isExternal;
  struct TrieNode *children[26];
};
```



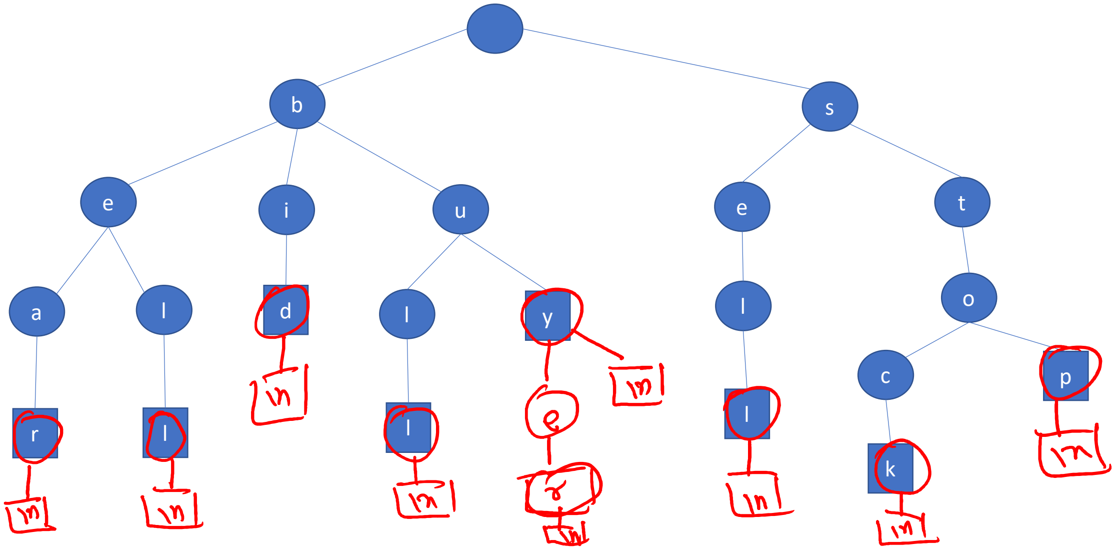
In an array representation, each node contains references to d children, where d is the size of the alphabet. Additionally, a tree node also has an additional field to track the external nodes. To make the kth character of the alphabet a child of node n, we can create a new node and insert it at index k in the children field of n. Searching the kth character in the children can be done in O(1) step using an array. In contrast, the linked-list based implementation takes O(d) operations to search the kth character in the children.

Time complexity

- Searching a word of length L $O(L)$
- Inserting a word of length L $O(L)$

Tries

Insert "buyer"



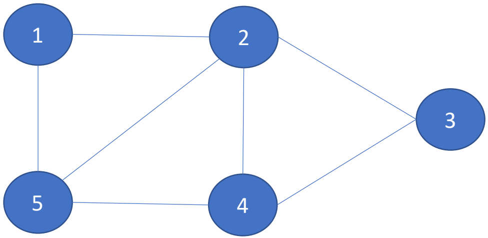
To support the insertion of a word that is a prefix of another word in the trie, we can allow internal nodes to become a square. Alternatively, if we want all square nodes to be leaf nodes, we can append a terminating character (e.g., '\n') that is not part of the alphabet. This ensures that a string can't be a prefix of another string in the trie.

Graphs

References

- Read chapter-20 of the CLRS book
- Read chapter-6 from Goodrich and Tamassia book

Terminology



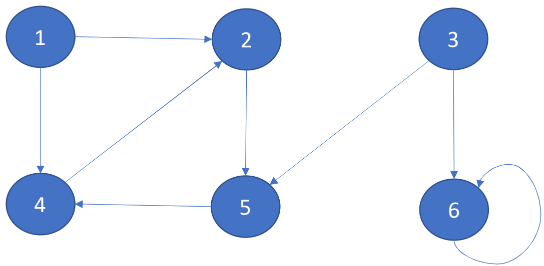
(1, 2), (1,5), (2, 4),
(2,5), (2,3), (3,4), (4,5)

- Graphs have vertices and edges

Nodes 1 2 3 4 5 are vertices.
The unordered pairs
(1, 2), (1, 5), (2, 3), (2, 4), (2, 5),
(3, 4), (4, 5) are edges.

This is also called an undirected graph. In an undirected graph, the vertices in the edges are unordered.

Terminology



- Graphs have vertices and edges

Nodes 1 2 3 4 5 6 are vertices.
The ordered pairs
(1, 2), (1, 4), (2, 5), (3, 5), (3, 6),
(4, 2), (5, 4), (6, 6) are edges.

This is also called a directed graph.
In a directed graph, the vertices in the edges are ordered.

Graph terminology

- A Graph G = (V, E) consists of a set of vertices (V) and a set of edges (E)

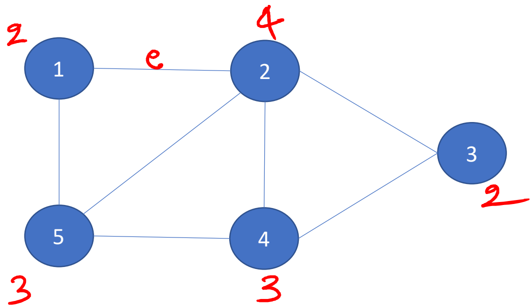
- Each edge is a pair (u, v), where $(u, v) \in V$
- An edge (u, v) is directed if from u to v if the pair (u, v) is ordered
- An edge (u, v) is undirected if the pair (u, v) is not ordered

- If all edges are undirected, then the graph is called an undirected graph
- If all edges are directed, then the graph is called a directed graph

Graph terminology

- Two endpoints of an edge are called end vertices of an edge
- Two vertices are adjacent if they are the endpoints of the same edge
- An edge e is incident on a vertex v, if v is one of the endpoints of e
- The degree of a vertex v is the number of incident edges on v

Terminology



- Which are the adjacent vertices?

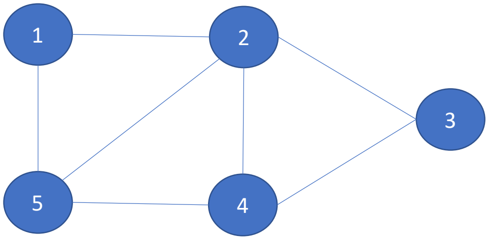
(1,2), (1,5)

(1, 4) NO

- What is the degree of each vertex?

There are two edges that are incident to node 1; therefore, the degree of node 1 is 2. The degree of node 2 is four because four edges are incident on it. Vertices 1 and 4 are not adjacent because they are not the endpoints of the same edge.

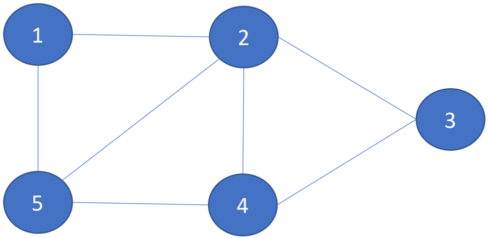
Sum of degree



- If G is a graph with m edges and deg(v) is the degree of a vertex v, then

$$\sum_{v \in G} \deg(v) = 2m$$

Sum of degree



- If G is a graph with m edges and deg(v) is the degree of a vertex v, then

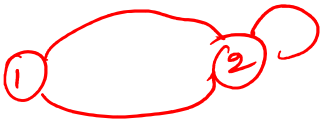
$$\sum_{v \in G} \deg(v) = 2m$$

An edge (u, v) is counted twice. The first time at vertex u and the second time at vertex v. Therefore, the sum is twice the number of edges, 2m.

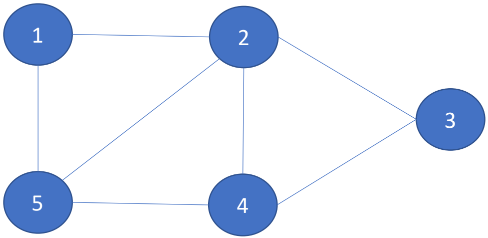
Simple graph

- Simple graph doesn't contain two undirected edges between the same pair of vertices or an edge from a vertex to itself

- The graphs that we are going to study are simple



Maximum edges



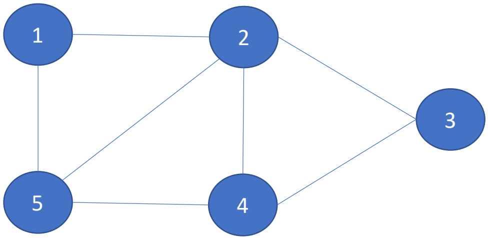
- What is the maximum number of edges in an undirected graph with n vertices?

$^nC_2$

- What is the maximum number of edges in a directed graph with n vertices?

$2 \times {}^nC_2$

Path



- A path in a graph is a sequence of vertices such that two consecutive vertices are adjacent

1 2 4 5 2 3 Yes
2 5 3 2 1 No 5, 3 are not adjacent
1 2 Yes

- A path is simple if each vertex in the path is distinct

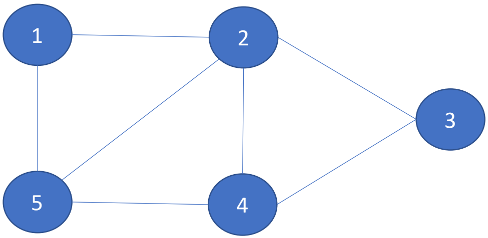
1 2 4 5 Yes
1 2 3 4 2 No 2 is repeated

- A simple cycle is a path that is simple, except the last vertex is the same as the first vertex

2 4 5 2 simple cycle
1 2 4 5 2 1 No.

Subgraph

- A subgraph of a graph G is a graph H whose vertices and edges are a subset of G

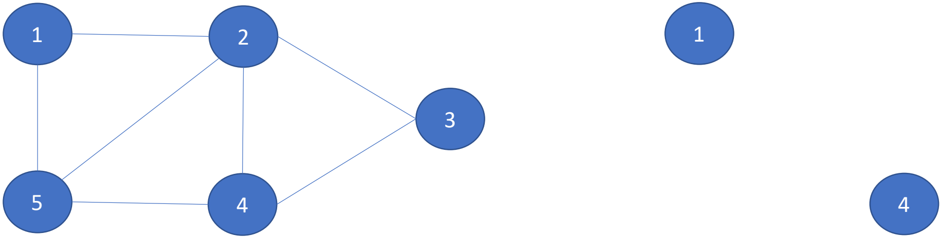


Yes

H is a subgraph of G because it contains no edge (a subset of the edges in G) and vertex 1 (a subset of the vertices in G).

Subgraph

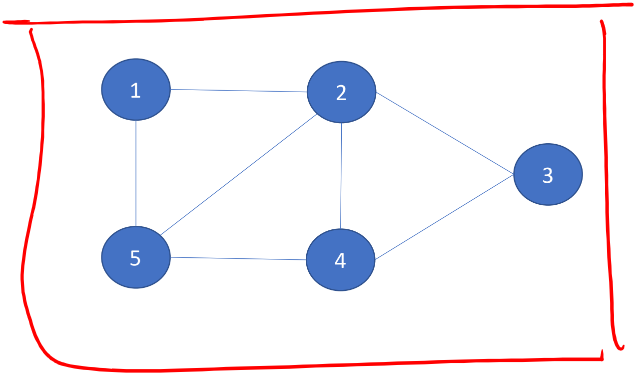
- A subgraph of a graph G is a graph H whose vertices and edges are a subset of G



H is a subgraph of G because it contains no edge (a subset of the edges in G) and vertices 1 and 4 (a subset of the vertices in G).

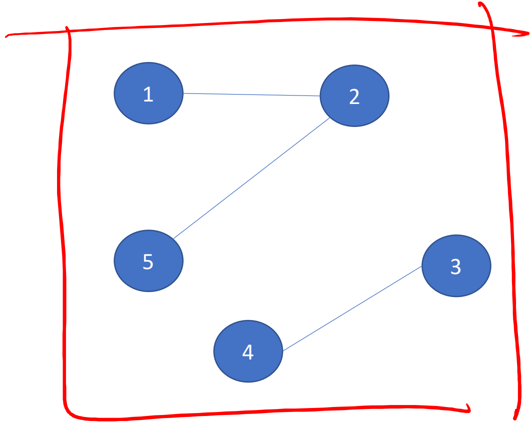
Subgraph

- A subgraph of a graph G is a graph H whose vertices and edges are a subset of G



G

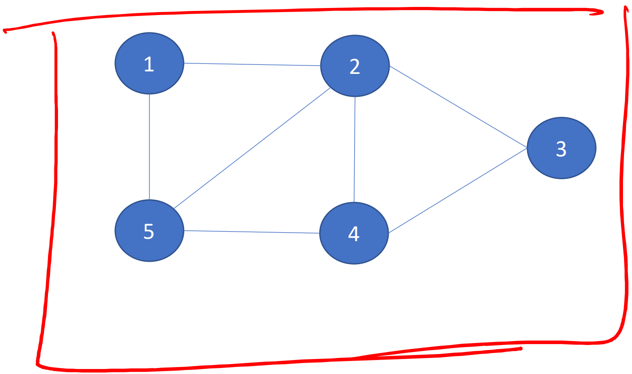
Yes



H

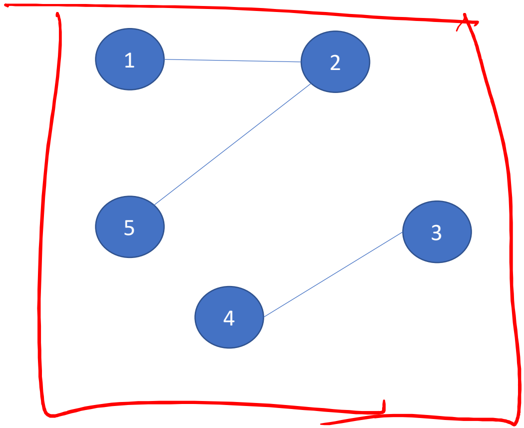
Spanning subgraph

- A spanning subgraph of G is a subgraph H that contains all the vertices of the graph G



G

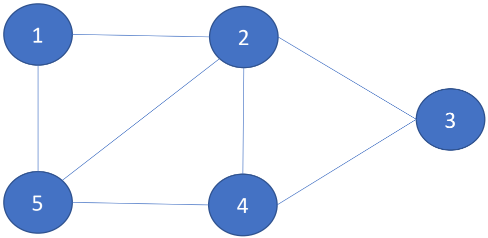
Yes



H

Connected graph

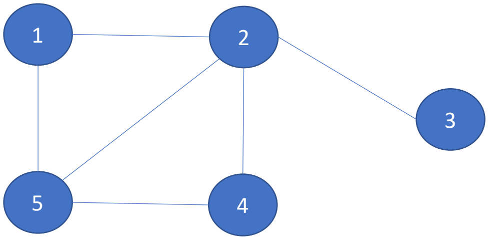
- A graph is connected if there is a path between any two vertices



Yes

Connected graph

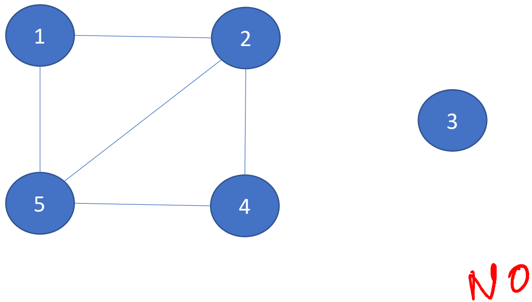
- A graph is connected if there is a path between any two vertices



Yes

Connected graph

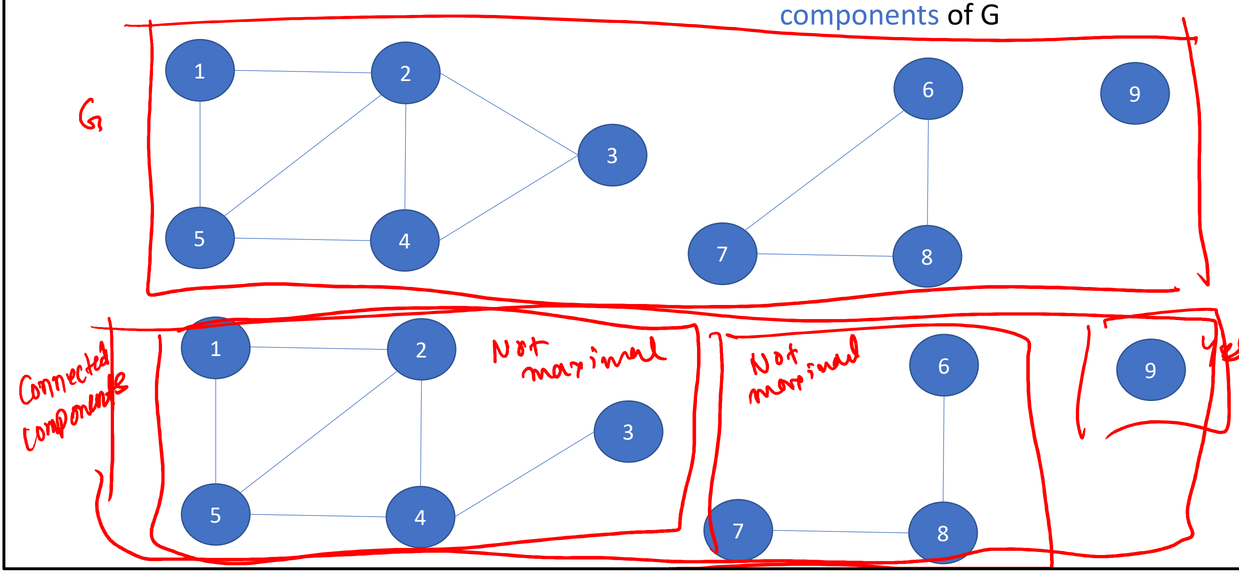
- A graph is connected if there is a path between any two vertices



No path from 3 to any other vertex.

Connected components

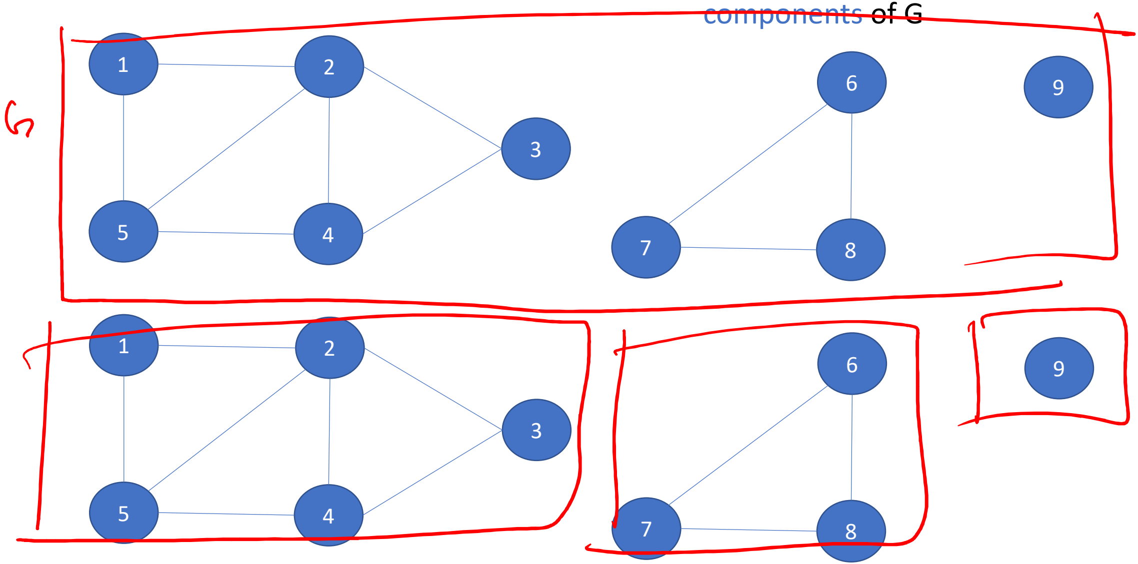
- If a graph is not connected, its maximum connected subgraphs are called the connected components of G



A connected subgraph has two properties. It is a subgraph that is also connected. A maximal connected subgraph is a subgraph to which we can't add more edges or vertices without violating the properties of a connected subgraph. The top rectangle is the original graph. We are trying to determine whether the subgraphs in the bottom half are maximal connected subgraphs. The leftmost subgraph at the bottom is a connected subgraph; however, it is not a maximal connected subgraph because we can add more edges to it without violating the property of a connected subgraph. Similarly, the middle subgraph at the bottom is not a maximal connected subgraph. Node 9 is a maximal connected subgraph because we can't add more edges or vertices to it while preserving the property of a connected subgraph.

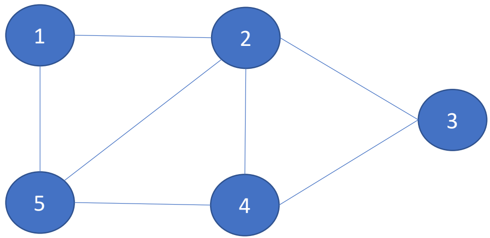
Connected components

- If a graph is not connected, its maximum connected subgraphs are called the connected components of G



The three subgraphs at the bottom are maximal connected subgraphs.

Tree

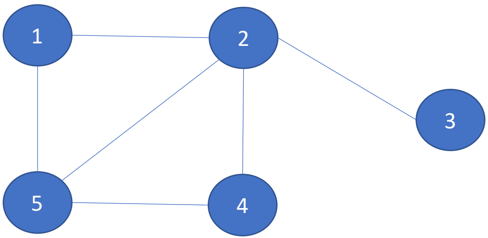


- A tree is connected graph without cycles
  - No notion of a root node, unlike the rooted tree that we discussed earlier
- Is this a tree?

No

This is not a tree because the path 1 2 5 1 is a cycle.

Tree

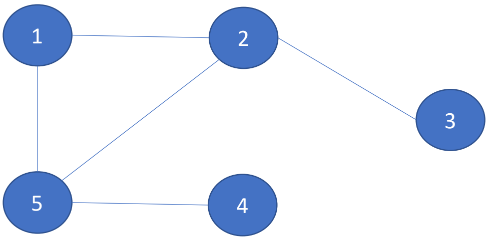


- A tree is connected graph without cycles
  - No notion of a root node, unlike the rooted tree that we discussed earlier
- Is this a tree?

NO

This is not a tree because the path 1 2 5 1 is a cycle.

Tree

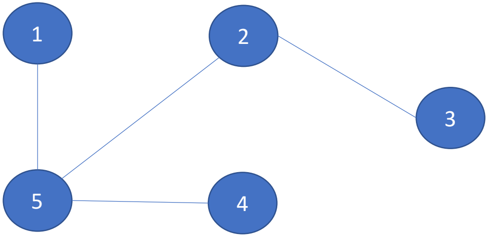


- A tree is connected graph without cycles
  - No notion of a root node, unlike the rooted tree that we discussed earlier
- Is this a tree?

No

This is not a tree because the path 1 2 5 1 is a cycle.

Tree

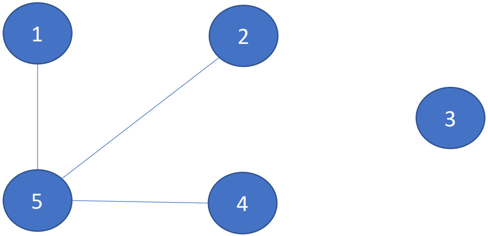


- A tree is connected graph without cycles
  - No notion of a root node, unlike the rooted tree that we discussed earlier
- Is this a tree?

  Yes

This is a tree because there are no cycles.

Tree



- A tree is connected graph without cycles
  - No notion of a root node, unlike the rooted tree that we discussed earlier
- Is this a tree?
  - No. Not a connected graph

This is not a tree because the graph is not connected.