EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

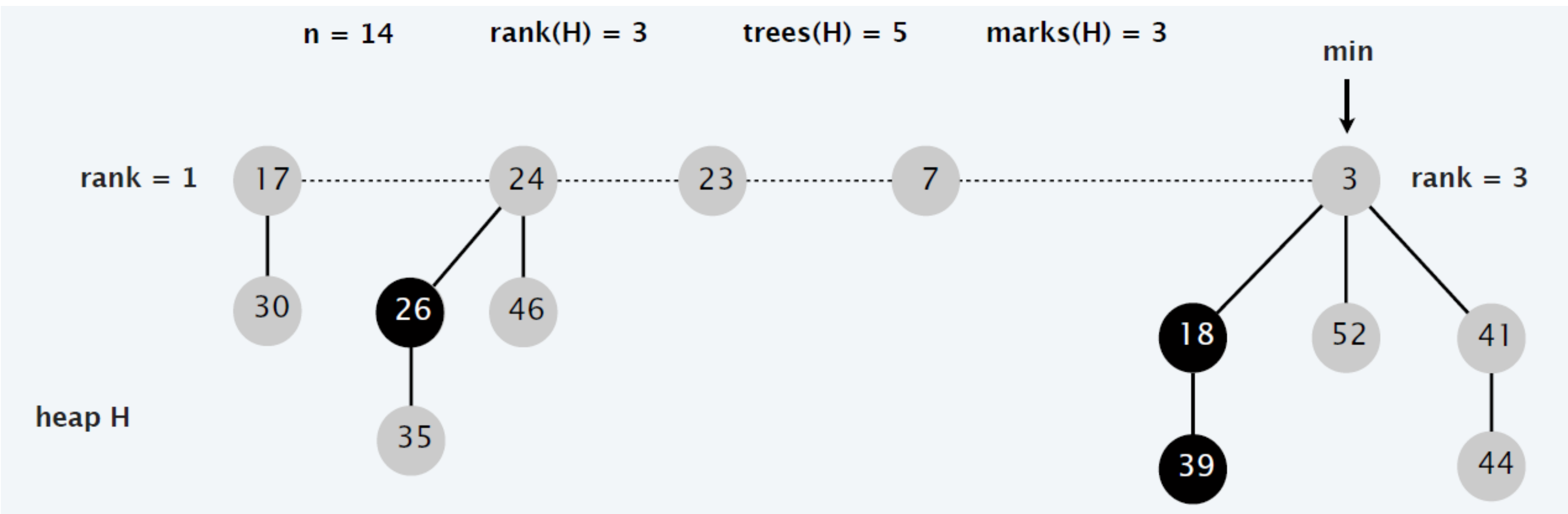
GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Amortized Analysis of DecreaseKey

- Amortized analysis: Potential function $\Phi(\mathscr{H})$ = number of trees + 2 * number of marked nodes



Fibonacci Heaps: Demonstration

- Illustrate use in shortest path computations

- https://kbaile03.github.io/projects/fibo_dijk/fibo_dijk.html

Recent Ideas in Priority Queues

- Try to get complexity of Fibonacci heaps for DecreaseKey with simpler data structures

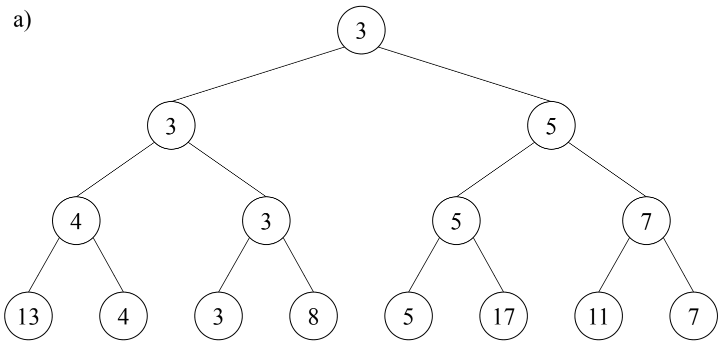
- Novel concepts
  - Pairing Heaps
  - Quake heaps
  - Violation heaps
  - Rank-Pairing heaps

- Focus on rank-pairing heaps

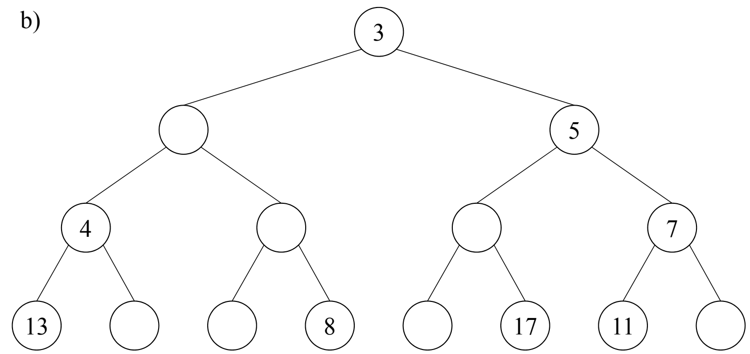
Rank-Pairing Heaps

- Background: Tournament trees

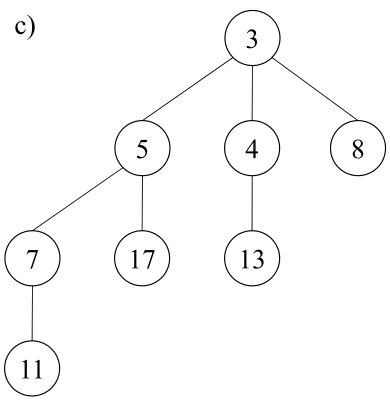
a)



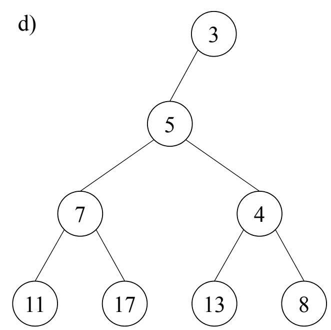
b)



c)



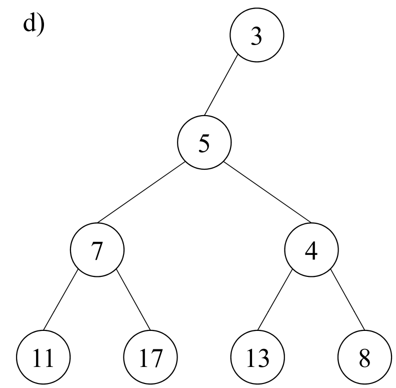
d)



Rank-Pairing Heaps

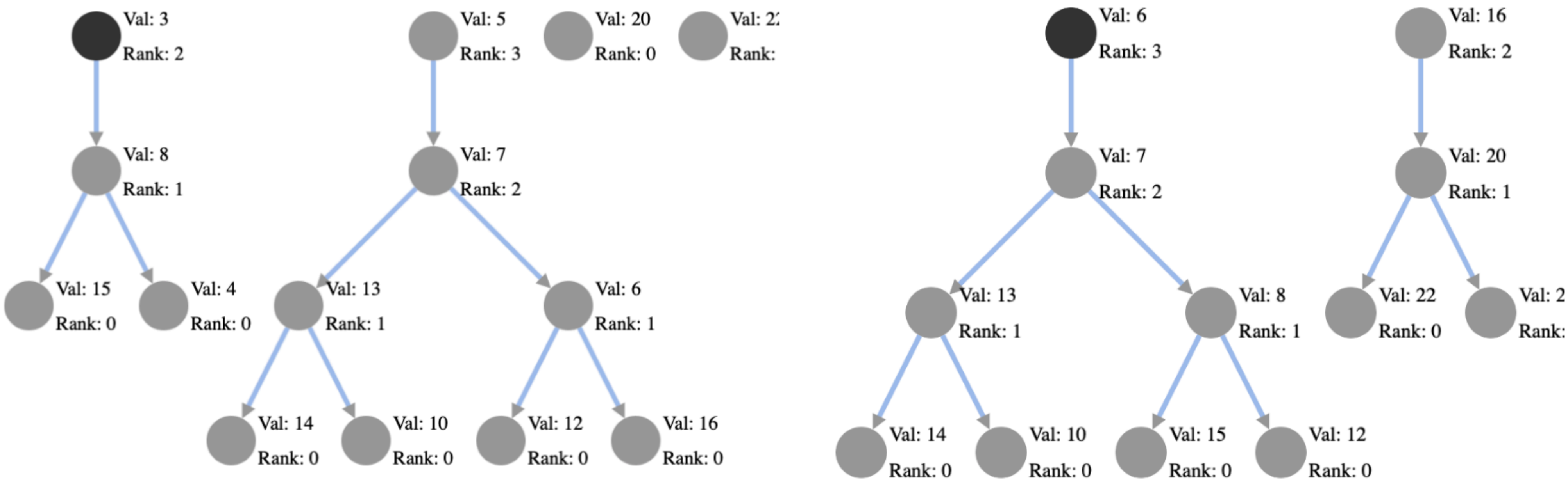
- Half-ordered binary trees (half-trees)
  - Root has single left child
  - Half-heap property: parent key less than or equal to left child

d)



- Rank of a node in half trees: similar to height…
  - If node is leaf, rank = 0
  - If it is root, rank(parent) = 1+rank(left child)
  - If |rank(left child) - rank(right child)| <= 1: rank = 1 + max(rank(L),rank(R))
  - If |rank(L) - rank(R)| 1: rank(parent)= max(rank(L), rank(R))

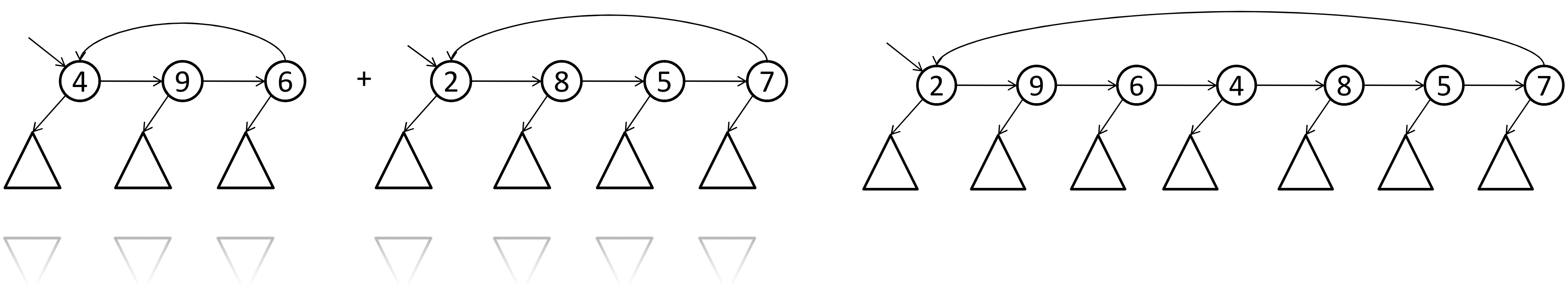
Half-Binary Trees



Rank-Pairing Heaps

- Rank-Pairing Heaps are a linked list of binary half-trees, with pointer to minimum root

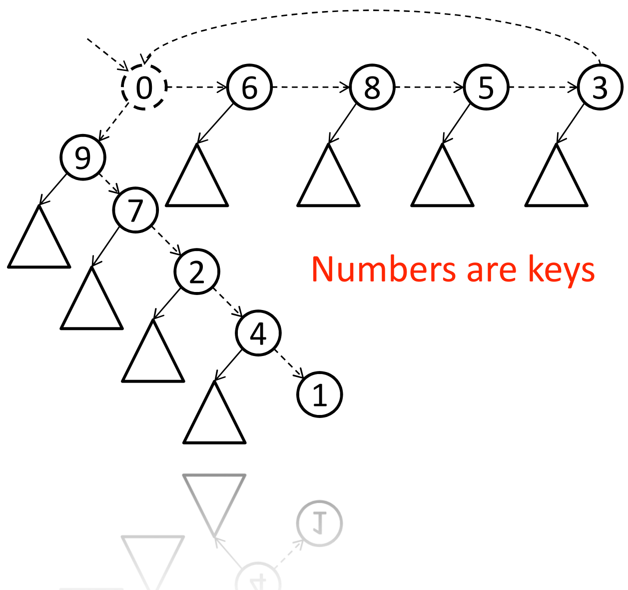
- Operations:
  - Insert: add a new single-node half-tree to list: O(1)
  - Merge: just link two lists, update min pointer: O(1)

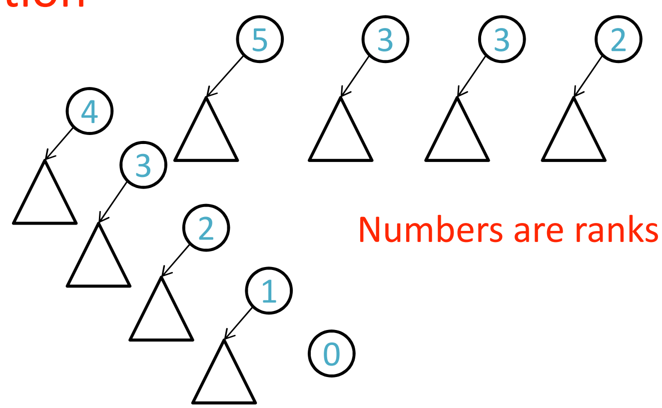


Rank-Pairing Heaps

- Operations (cont)
  - DeleteMin: Delete min-root.
    - Cut edges along right path down from new root to give new half-trees
    - Compress: merge roots of equal rank until no two roots have equal rank

Cut Operation

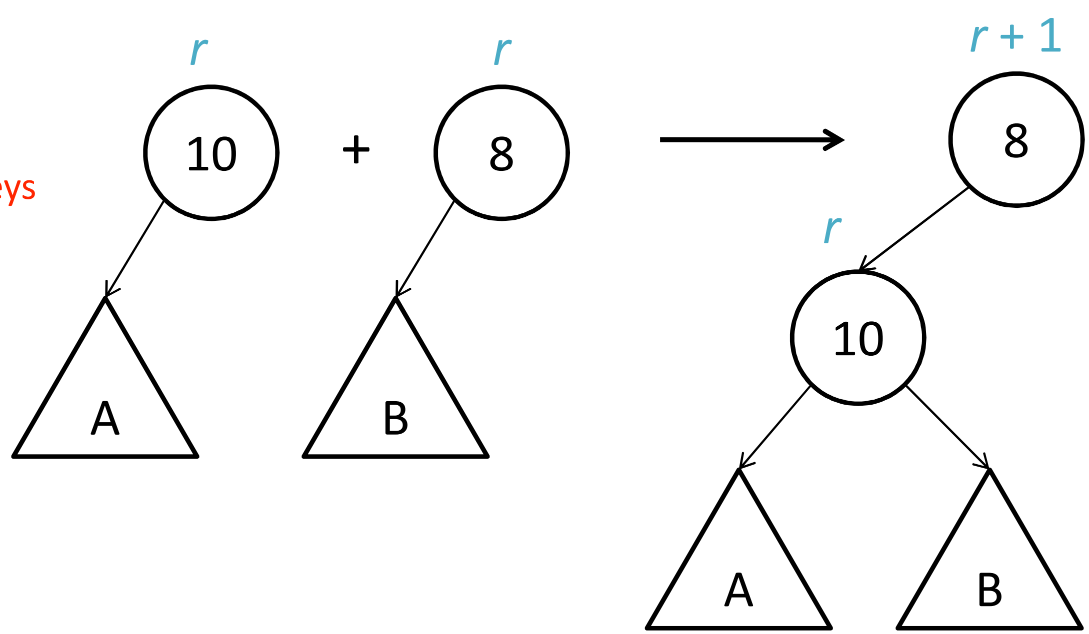




Rank-Pairing Heaps

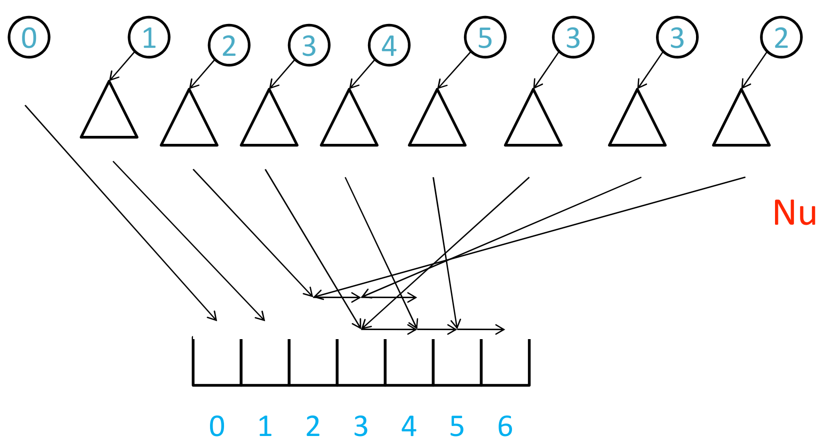
- Operations (cont)
  - Compress two trees of equal rank:

Numbers are keys



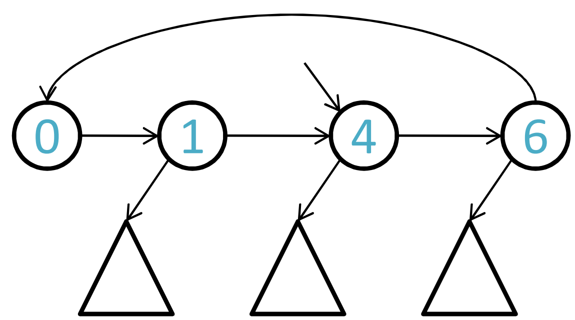
Rank-Pairing Heaps

- Operations (cont)
  - DeleteMin: Delete min-root.
    - Cut edges along right path down from new root to give new half-trees
    - Compress: merge roots of equal rank until no two roots have equal rank
    - Variation: Lazy compress: don't compress recursively



Merge

Numbers are ranks

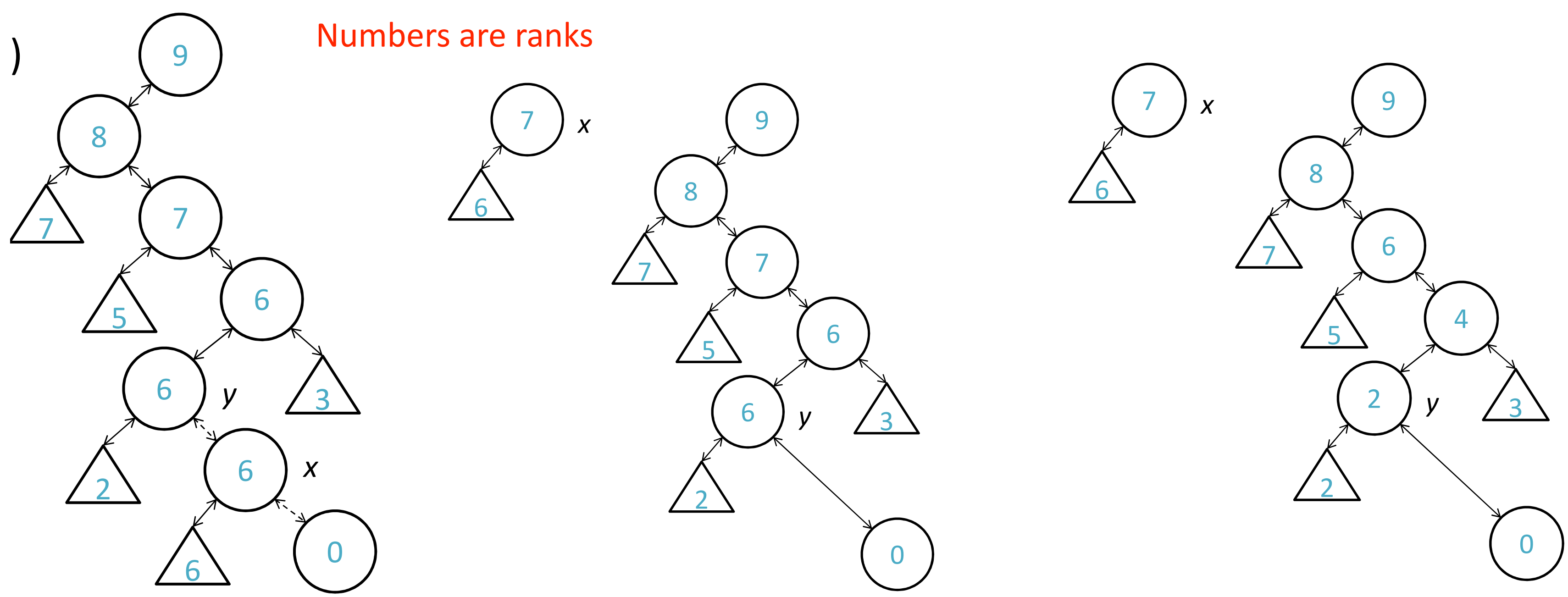


Rank-Pairing Heaps

- Operations (cont)
  - DecreaseKey: Remove x and its left subtree to a new half-tree
    - Replace x by its right child. Change key of x to k. Add x to the list of half tree roots. Update the min-root.
    - Update the ranks

Rank-Pairing Heaps

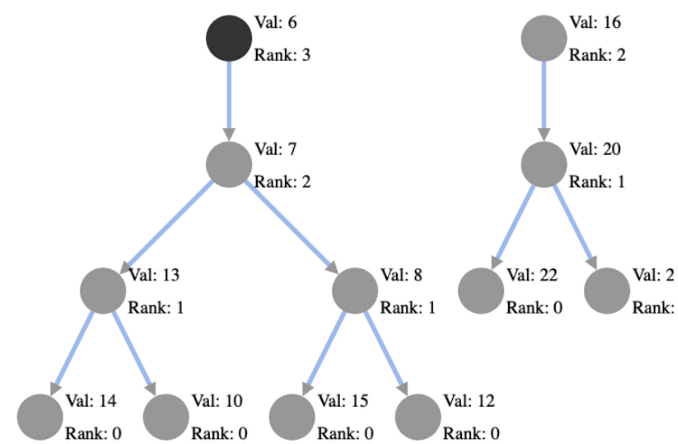
- Operations (cont)
  - DecreaseKey: Update the ranks



Rank-Pairing Heaps

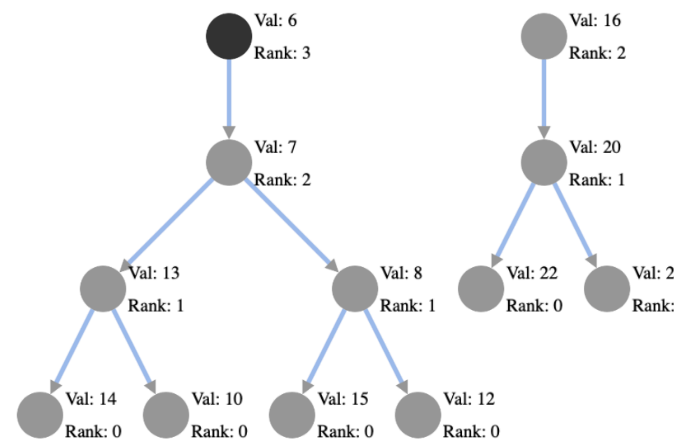
- Amortized analysis
  - Key result: number of nodes in a half tree with rank k is again bounded below by F_{k+2}, the Fibonacci number
  - Implies that the maximum height of half-trees in forest is O(log(n))

- Analysis based on potential function
  - Potential of a node: 1 if root, 0 if both children have same rank, 2 if one child's rank is 1 greater than other, j-1 if single child of rank j
  - Potential of heap: sum of potential of all node



Rank-Pairing Heaps: Demo

- https://skycocoo.github.io/Rank-Pairing-Heap/



Rank-Pairing Heaps

- Simpler implementation than Fibonacci heaps
- Same amortized complexity:
  - Insert: O(1)
  - DeleteMin: O(log(n))
  - DecreaseKey: O(1)
  - Merge: O(1)
- A subtle point in all heaps: need additional data structure to find nodes in O(1)
  - Not implemented in most standard libraries
  - Makes generic standard heap implementations have little utility …

String Matching and Tries

- Consider the following problem:
  - Given a string T and k nonempty strings $P_1, P_2, \ldots, P_k$, find all occurrences of $P_1, P_2, \ldots, P_k$ in T.
  - T is called the text string and $P_1, P_2, \ldots, P_k$ are called pattern strings.
  - This problem was originally studied in the context of compiling indexes, but has found applications in computer security and computational genomics

- Simpler problem: k = 1
  - Knuth-Morris-Pratt (KMP) algorithm (CLRS)

- Harder problem: k > 1
  - Aho-Corasick tries

Single Pattern Matching

- Given a string T[1:n-] of n characters in finite alphabet set S
- Nonempty string P = P[1:m] of characters in S

- Pattern P occurs with shift s in T[s:s+m-1] = P[1:m]

- Problem: Find all shifts s where Pattern occurs with shift s

- Naive algorithm:
  - For s in 1 to n-m+1:
    - If T[s:s+m-1] = P[1:m], add s to set of shifts where pattern occurs
  - Complexity: O(m(n-m+1))

Single Pattern Matching

- Example:
  - T[] = “AAAAAAAAAAAAAAAAAB"   (n=18)
  - P[] = “AAAAB"   (m=5)

One shift matches: s=13

- Example
  - T[] = “ABABABCABABABCABABABC" (n=20)
  - P[] = “ABABAC" (m=6)

No shifts match!

Knuth-Morris-Pratt Algorithm

- Idea: exploit that whenever we find a mismatch, we have already looked at a subset of the pattern
  - T[] = “AAAA A AAAAAAAAAAAAB" (n=18)
  - P[] = “AAAA B" (m=5)
  - If we get a mismatch on T[5], we don’t have to start searching from k=2 for the next match! We know no shift can match before s = 6
- Idea: patterns have a prefix function
  - A string w is a **proper prefix** of a string x if x = w + v for some string non-empty string v, where + is concatenation
  - A string w is a **proper suffix** of a string x if x = v + w for some non-empty string v

Knuth-Morris-Pratt Algorithm

- Algorithm: preprocess pattern P[0:m-1] to compute function π[j]
  - π[j] = longest proper prefix of P[0:j] that is also a suffix of P[0:j], for j = 0 to m-1
  - Examples: P = “AAAA” —> π[] = [0, 1, 2, 3]
  - P = “ABCDE” —> π[] = [0,0,0,0,0]
  - P = “AABAACAABAA” —> π[] = [0,1,0,1,2,0,1,2,3,4,5]
  - P = “AAACAAAAAC” —> π[] = [
  - P = “AAABAAA” —> π[] =

Knuth-Morris-Pratt Algorithm

- Analysis
  - String P: “a b a b a b a b c a”; Prefix function: $\pi(i)$: [0,0,1,2,3,4,5,6,0,1]
- Match into T: “a b a b **d** a a b a b a b a b c a d c d”

```
a b a b d a a b a b a b a b c a d c d
a b a b a b a b c a    Shift by π(3)

a b a b d a a b a b a b a b c a d c d
    a b a b a b a b c a shift by π(1)

a b a b d a a b a b a b a b c a d c d
        a b a b a b a b c a
```

```
a b a b d a a b a b a b a b c a d c d
          a b a b a b a b c a

a b a b d a a b a b a b a b c a d c d
            a b a b a b a b c a
```

Knuth-Morris-Pratt Algorithm

- Computing π[j] from P[0:m-1]:

```
k= 0; j = 1;
π[0]=0;
While j < m:
  If P[k]==P[j]:
    k++; π[j]=k; j++;
  else if k == 0:  # first character did not match
     π[j] = 0; j++;
  else:  #mismatch after first character
     k = π[k-1]
```

w	c	a	b	c	a
arr	0	0	0	1	2

P	A	A	B	A	A	C	A	A	B
Pi	0	1	0	1	2	0	1	2	3
K	0	1	0	1	2	0	1	2	3
J	1	2	3	4	5	6	7	8	9

Knuth-Morris-Pratt Algorithm

```
k = 0; j = 0;
while (j < n)
    if P[k] == T[j]
        k++; j ++;
    if  k == m:
        print("match at  ", j-i)
        k=pi[k-1]
    else if (j < n and P[k] != T[j]):
        if k != 0
            k= pi[k-1]
        else
            j++;
return
```

T	A	A	B	A	A	A	A	A	B
P	A	A							
pi	0	1							
s	1	0	0	1	1	1	1	0	0
k	0/1	1/0	0	0/1	1/0	1/0	1/0	1/0	0
j	0	1	2	3	4	5	6	7	8

T	A	A	B	A	A	A	A	A	B
P	A	A	A						
pi	0	1	2						
s	0	0	0	1	1	1	0	0	0
k	0/1	1/2	2/0	0/1	1/2	2/1	2/1	2/1	0
j	0	1	2	3	4	5	6	7	8

Knuth-Morris-Pratt Algorithm

- Complexity analysis
  - For each position j in the string T:
    - If it does not match, you slide the pattern to all possible suffixes and check for a match
    - Worst case pattern: "AAAAAAA": $\pi[k] = k$ so lots of prefixes

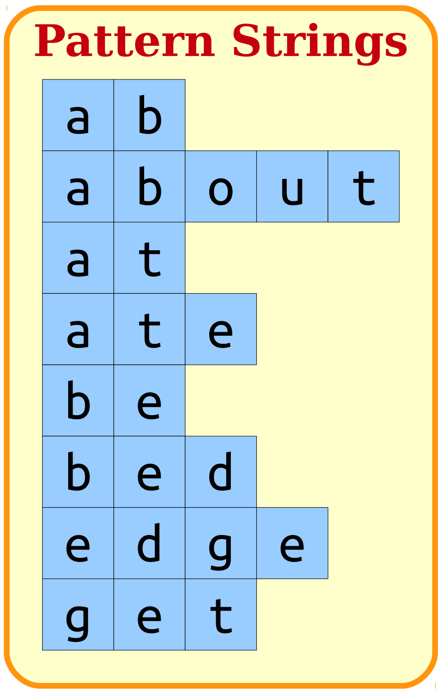
- Let q = number of digits matched when checking position j.
  - Potential $\Phi(j) = q$. Note $\Phi(0) = 0$, $\Phi(end) = 0$
  - Cost of checking position j: c(j) = number of iterations of k = $\pi[k-1]$ for this j
    - Cost is 1 + c(j)
  - Each iteration k = $\pi[k-1]$ decreases number matched by at least one
  - Amortized cost for position j: 1 + c(j) + $\Phi(j) - \Phi(j-1)$ = 1 + c(j) - c(j): $\Theta(1)$
- To do n iterations: $\Theta(n)$

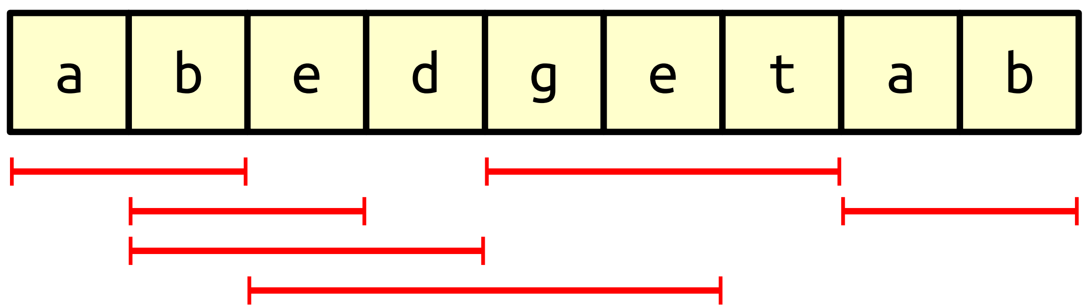
Knuth-Morris-Pratt Algorithm

- Demo: http://jovilab.sinaapp.com/visualization/algorithms/strings/kmp

Search for Multiple Strings

- KMP good for 1 pattern P
- What if we are looking for k patterns $P_1, P_2, \ldots, P_k$?



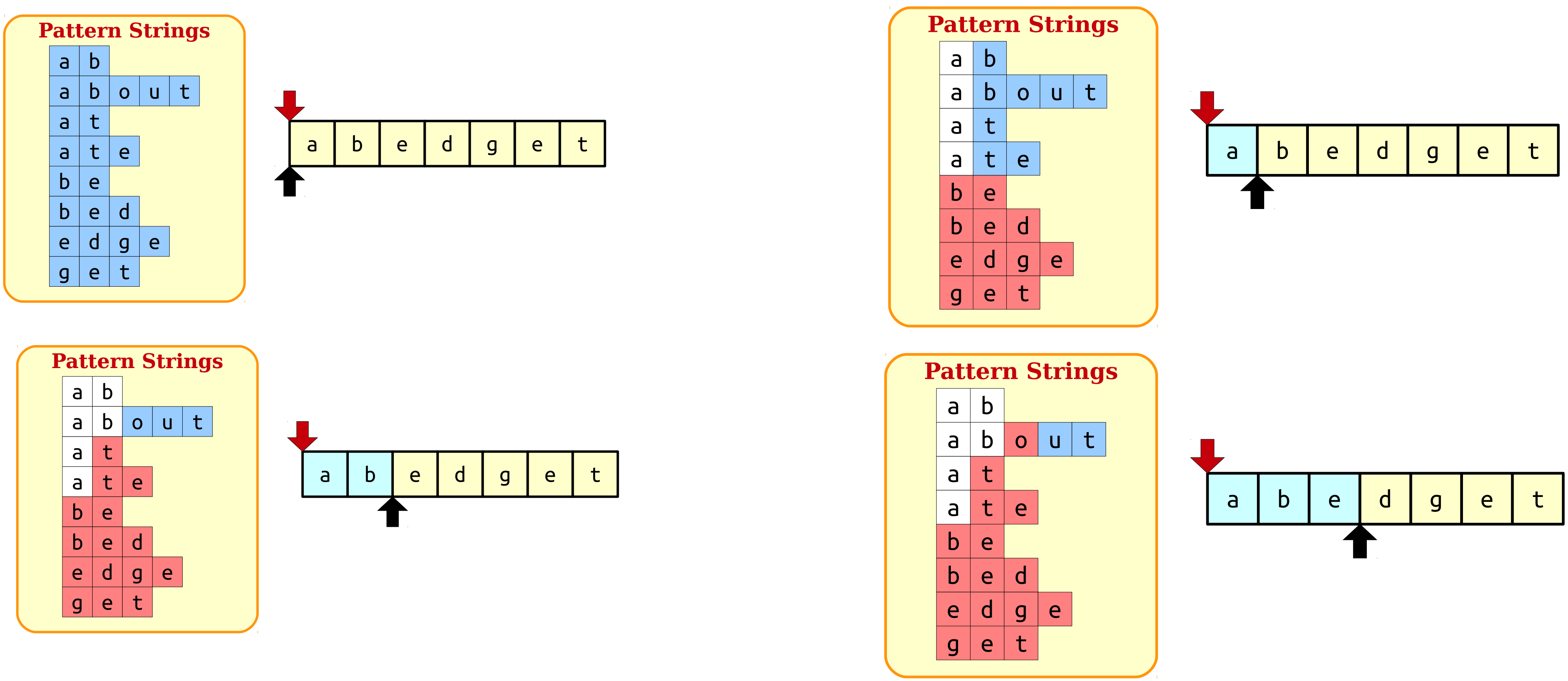


Search for Multiple Strings

- Notation: m is T string length; n is total length of k patterns; L_{max} length of longest pattern
- Naive approach:
  - For each position in T:
    - For each pattern string P_j:
      - Check if P_j appears at that position
  - $\Theta(mn)$
  - Can we do better?

Search for Multiple Strings

- Can we do better?



Search for Multiple Strings

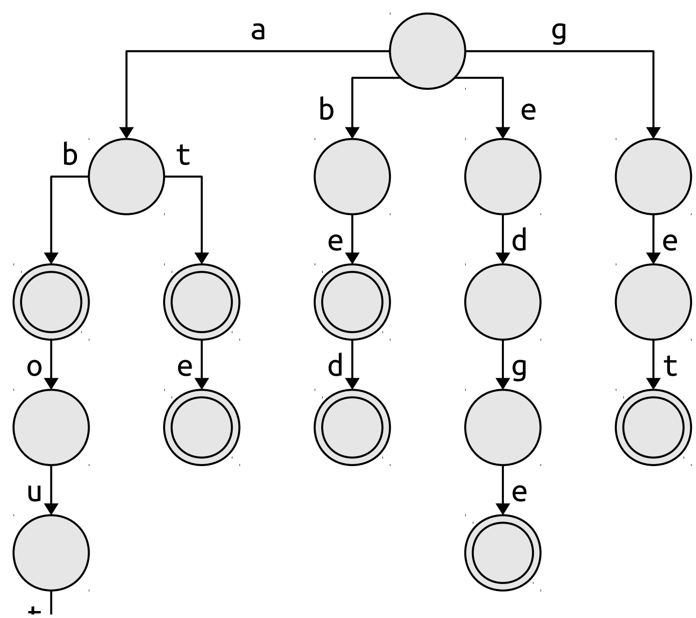
- Idea: Search for all strings in parallel
- How? Form the strings into a **trie**: word comes from retrieval
- A trie: a tree structure where the branches at each level correspond to symbol values
  - Example trie: index tables at end of documents
  - Important idea: In a trie, you wait to subdivide a node until there is a collision of two elements…saves storage! Can identify nodes with key words as potential end nodes

Search for Multiple Strings

- Example trie for words

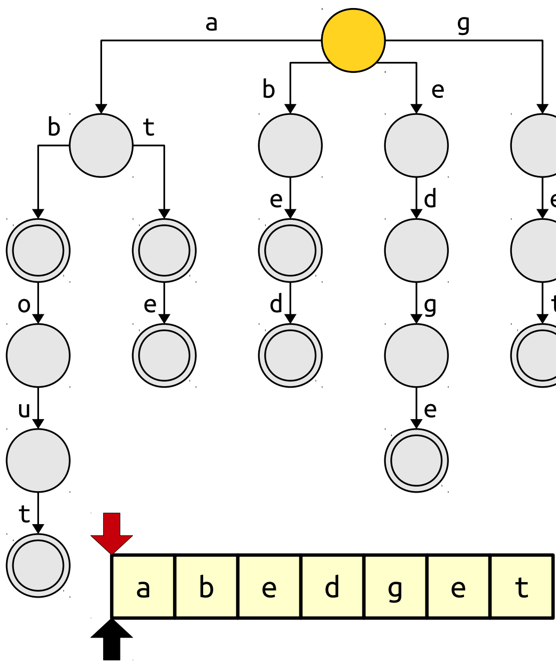
Pattern Strings

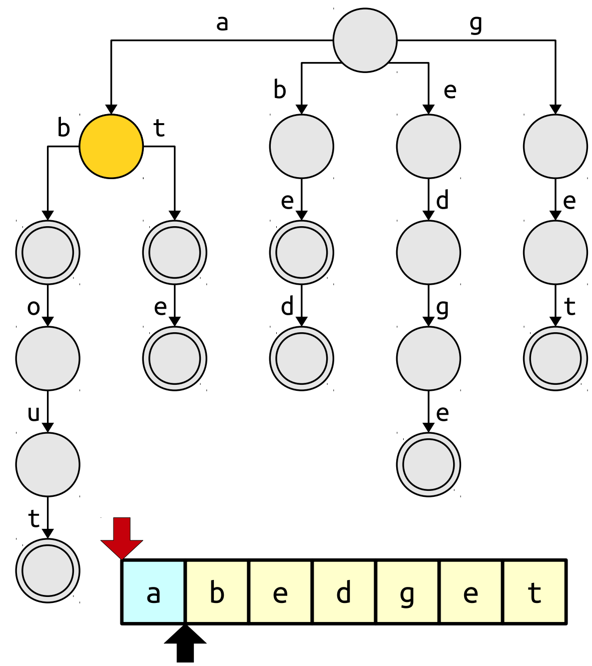
a	b			
a	b	o	u	t
a	t			
a	t	e		
b	e			
b	e	d		
e	d	g	e	
g	e	t		



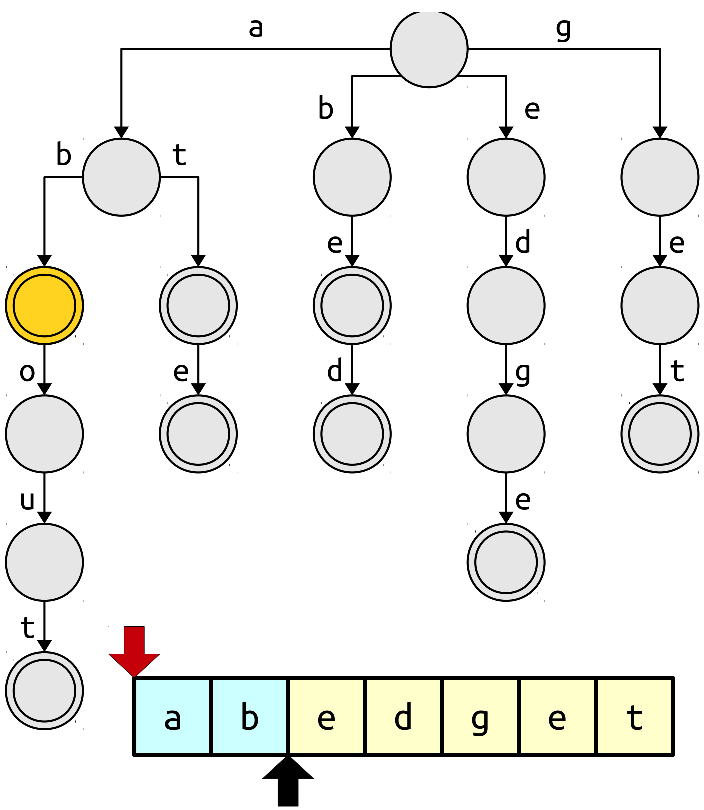
Representing Tries

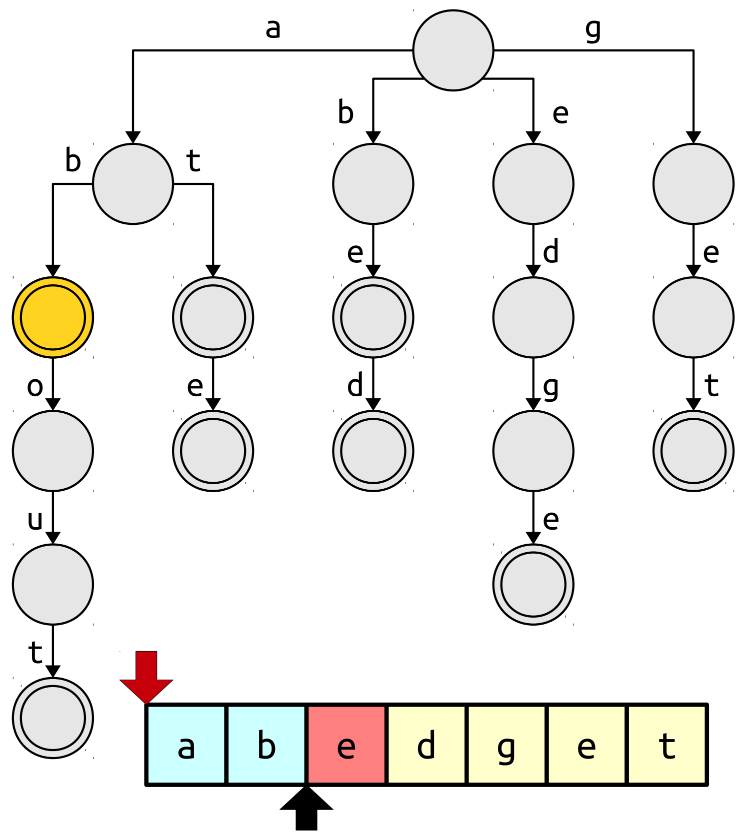
- Each trie node has pointers to its possible children
  - Assume you have an array of pointers at each node, many of which could be null
- Matching



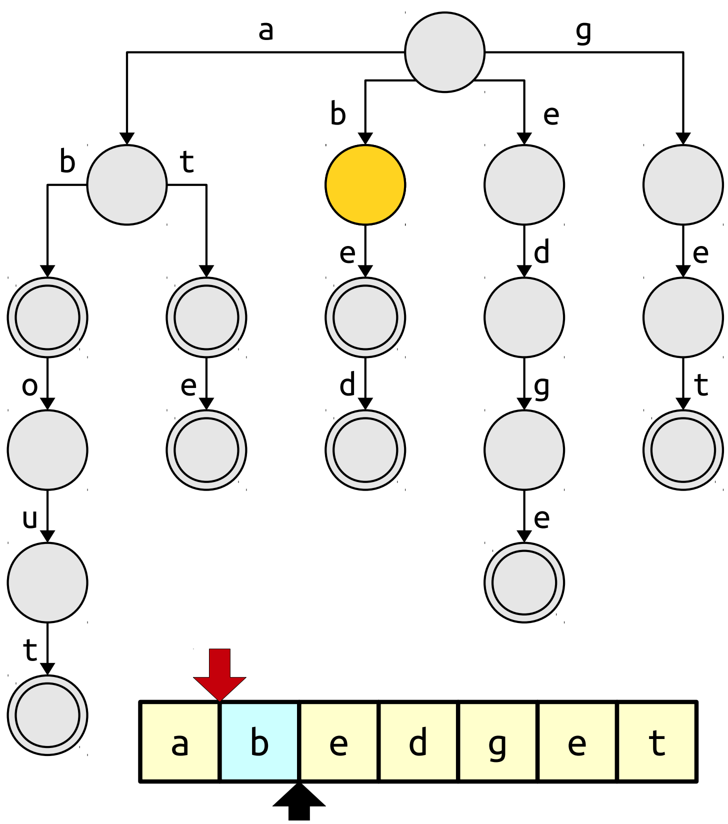


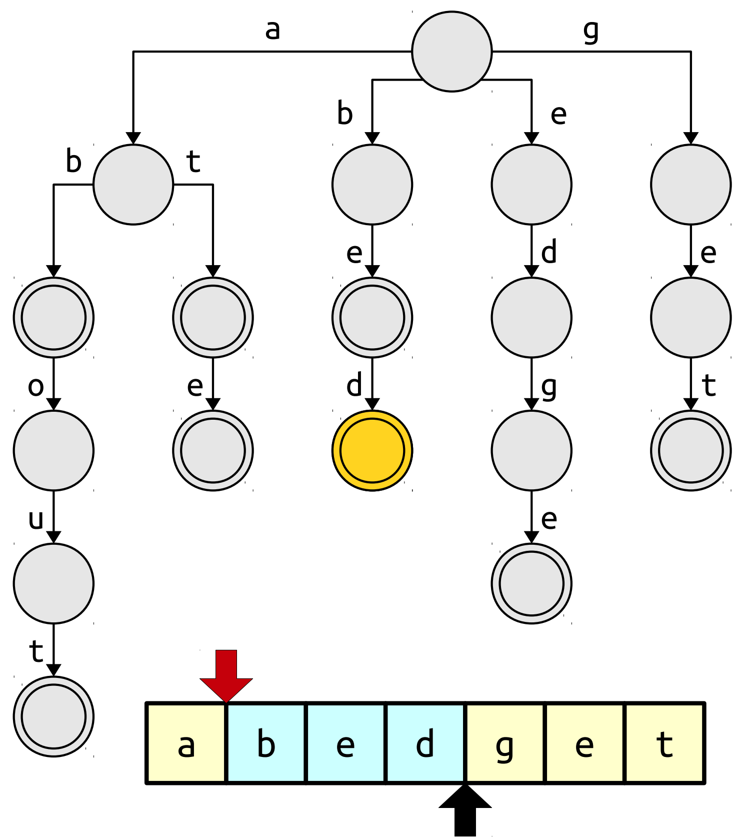
Matching Tries





Matching Tries

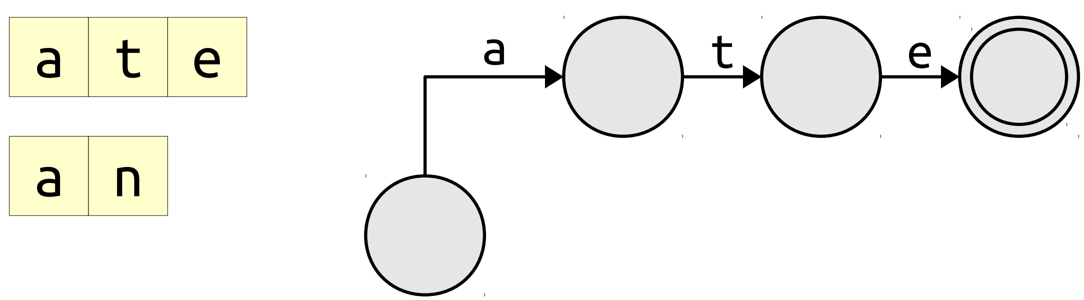




Analysis of algorithm

- Complexity: $\Theta(mL_{max})$
  - Good reduction if n is large (several patterns)
- But: How long to build a trie? Need that preprocessing step

- Claim: Given a set of strings $P_1, P_2, \ldots, P_k$ of total length n, it's possible to build a trie for those strings in time $\Theta(n)$



Building Tries